WORKBENCH 1.3 & 2.0

AmigaDOS® Inside & Out

Revised

An in-depth guide to AmigaDOS and the Shell

by Kerkloh, Tornsdorf and Zoller

Includes ready-to-use companion diskette

Abacus
a Data Becker Book

AmigaDOS Inside & Out

Ruediger Kerkloh
Manfred Tornsdorf
Bernd Zoller

A Data Becker Book
Published by
Abacus

Seventh Printing 1991
Printed in U.S.A.
Copyright © 1988, 1989, 1990, 1991
Abacus
5370 52nd Street SE
Grand Rapids, MI 49512

Copyright © 1988, 1989, 1990, 1991
Data Becker GmbH
Merowingerstrasse 30
4000 Düsseldorf, Germany

This book is copyrighted. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Abacus Software or Data Becker GmbH.

Every effort has been made to ensure complete and accurate information concerning the material presented in this book. However, Abacus Software can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors always appreciate receiving notice of any errors or misprints.

Amiga 500, Amiga 1000, Amiga 2000 and Amiga are trademarks or registered trademarks of Commodore-Amiga Inc.

ISBN 1-55755-041-7

Table of Contents

Preface

The Amiga Workbench, a user-friendly mouse controlled graphic operating system, makes it easy for the beginner to enter the world of computers. The windows and icons which appear on the screen after you start the computer are much more attractive to a new user than a plain cursor waiting for simple keyboard input.

Sooner or later, either by mistake or out of curiosity, you click the `Shell` icon on the Workbench disk. A `Shell` window appears and the boring `CLI` (Command Line Interface) cursor of AmigaDOS makes its appearance. This user interface, although it doesn't use the mouse, is more powerful than the Amiga Workbench. In fact, the Workbench is loaded from `AmigaDOS` when the Amiga is turned on.

You actually can't get by without using AmigaDOS if you wish to do any meaningful work with the Amiga. The Workbench is a powerful graphic interface that makes it easy for the average user to access the Amiga. You can only do so much with the Workbench, while AmigaDOS's capabilities are almost unlimited.

This book will be very helpful to you if you keep it by your side as you work with AmigaDOS. After a simple but necessary introduction, you'll find a lot of information about AmigaDOS. You'll learn solutions to common problems, detailed descriptions of all AmigaDOS commands, programming script files, multitasking, and even an explanation of the internal workings of AmigaDOS and the `Shell`. The last few pages contain a *Quick Reference* of all the commands.

Workbench 1.3 2.0

One final comment: The Amiga is an ever expanding system and the Workbench is constantly being improved. This book covers AmigaDOS in Workbench 1.2, 1.3 and 2.0. These new system disks work much better than the older versions. So, any additions or differences between the Workbench versions are indicated as they appear in this book. This book supports Workbench 1.3/Kickstart 1.2, Workbench 1.3/Kickstart 1.3 and Workbench 2.0/Kickstart 2.0.

R. Kerkloh, M. Tornsdorf, B. Zoller *June 1990*

1.
Introduction

1. Introduction

The first steps in any area of computing usually seem the hardest. For this reason, we have kept the theories in this chapter to a minimum. The following sections are intended to make your first experiences with the `Shell` and AmigaDOS as easy as possible. In fact, the only AmigaDOS commands that appear in the following sections of Chapter 1 are the necessary ones. For those who wish to experiment further, the later chapters contain more background information on AmigaDOS.

For now, however, we recommend that you read this book in sequence and work through the examples as they appear. Whether you've just unpacked your Amiga or are an old hand at the computer, starting from the beginning is always the best way to learn anything. Good luck!

Workbench 1.3 and 2.0 users:

Workbench 1.3 and 2.0 have two AmigaDOS access programs: The `CLI` (contained in the `System` drawer) and the `Shell`, which is in the `Workbench` window. Please use the `Shell` for all of your work with this book. The `Shell` is an upgraded version of the original Amiga command line interface or `CLI`. Once you have become familiar with the `Shell`, you will probably use the `Shell` exclusively, but please use the `Shell` for the examples in this book. In Workbench 2.0 the `Shell` and the `CLI` program both operate in the same manner.

1.1 The Task of AmigaDOS

What is DOS?

Before we begin working with AmigaDOS, we must first briefly explain the function of AmigaDOS. *DOS* is the abbreviation for *Disk Operating System*. You may already know the definition of an *operating system*: The program(s) that controls the computer (tells it what to do). Don't confuse an operating system with an *application* program (e.g., word processors, spreadsheets, etc.). An operating system only provides the computer with basic instructions from which a programmer can construct his programs. It takes over such tasks as memory management, hardware control (keyboard, disk drive, printer, etc.) and coordinating various functions. It also makes operating system functions available to a programmer. A system programmer, for instance, shouldn't worry about which areas of memory in the computer are occupied and which areas are still available. The operating system automatically allocates free memory of the desired size, if enough memory is available.

In AmigaDOS the disk commands that the computer can execute aren't integrated into the operating system itself. On some home computers, you can enter certain commands which the operating system recognizes and immediately executes (such as Load, Save, etc.). AmigaDOS is based on a different principle: AmigaDOS commands are short programs that can be loaded from a disk drive (floppy disk, hard disk or RAM disk) before they execute. Upon execution, AmigaDOS returns to the routines contained in the operating system. This method has certain advantages over an operating system with integrated commands:

- Each command occupies memory only when it executes. After execution, it is removed from memory. AmigaDOS also allows often used commands to remain resident in memory.
- If the authors find that a command contains some kind of bug or error, it can later be fixed with a corrected version.
- An unlimited range of commands exists. New commands can be added to AmigaDOS as needed.

The biggest disadvantage of separate AmigaDOS commands is disk switching; exchanging disks takes time. This frequently occurs on smaller computers with a limited amount of memory space and a single disk drive. By using a hard disk or multiple floppy disk drives in conjunction with a RAM disk, this disadvantage can be avoided.

AmigaDOS 2.0

In AmigaDOS 2.0 all the AmigaDOS commands were rewritten for compactness and speed. This allowed many commands to be made internal commands, thereby allowing them to execute directly, no more loading from diskette. The Amiga designers recognized the flexibility of a system that calls commands from diskette so they built in an internal command override system, keeping the best of both worlds, internal and external commands. The following are the internal commands of AmigaDOS 2.0, these commands will be discussed in greater detail later:

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

AREXX

AREXX is a version of the mainframe computer REXX programming language that has been implemented on the Amiga. AREXX has been integrated in the Workbench 2.0 operating system. AREXX is an application programming language that can be used to extend operating system commands and customize application programs for easy interaction.

The complete power and possibilities that AREXX gives to the Amiga is beyond the scope of this book, so AREXX programming will not be covered in this volume. To do AREXX justice a separate volume would be necessary to describe all the features of this excellent addition to the Amiga.

1.2 The Workbench and the Shell

The previous section gave a rough description of what AmigaDOS does. AmigaDOS contains the tools with which the user can perform functions required for the operation of the computer. For example, how do you tell the computer that you want to format a disk? The Workbench can do this: In Workbench 2.0 there is a menu item in the `Icon` menu named `Format disk` and in Workbench 1.3 there is a menu item in the `Disk` menu named `Initialize`. You insert the blank disk, click once on its icon and select the `Format disk` or `Initialize` item from the menu, depending on which version of the Workbench you are running. This loads the corresponding command from the Workbench disk and any other commands as needed. The Workbench is actually nothing more than a program loaded from the disk when the computer boots up, creating the graphic user interface or GUI.

Command Line Interface

An alternative to the Workbench is the *Shell* or Command Line Interface. AmigaDOS commands entered from the keyboard form the command line interface, instead of icons or pointer. The mouse can only be used to change the size of any window opened for a `Shell` task.

Isn't the `Shell` a step backward in computer technology, then? It may seem that way at first glance, since the Workbench simplifies startup procedures on the Amiga. However, some aspects of the Amiga's operating system, and even the Workbench itself, cannot be accessed without AmigaDOS. The *Startup-sequence*, a file made of commands instructing the Amiga what to do or load as it starts up, can be edited and tested from AmigaDOS. This Startup-sequence is located on the Workbench disk, and the Amiga executes this file every time you turn the Amiga on.

In Workbench 1.3 some of the filenames on a disk are not visible on the Workbench (e.g., an invisible file may have no matching `info` file). As a result, AmigaDOS provides the best way to really look behind the scenes in Amiga disks. Workbench 2.0 solved this problem by allowing the user to view all files on a diskette, but AmigaDOS still gives more flexibility in the displaying of disk information.

1.3 Workbench Backup

Backup copies

Before you begin working with AmigaDOS and the `Shell`, make a copy of your original Workbench disk. Use this backup as your Workbench disk. As time passes, the backup disk may become *corrupt* (unreadable), or important files may be erased accidentally. If this happens, you can make another backup from the original Workbench disk.

It's easy to make a backup copy of the Workbench disk. If you have never backed up a disk before, do the following:

- Take the original Workbench disk. Look for the write protect tab (that sliding piece of plastic set into one corner of the disk. Move the write protect to the write protect position (you should be able to see through the disk in a hole created by the write protect). You cannot overwrite the Workbench disk when the write protect is in this position.

Workbench Backup

- Place the original Workbench disk in the internal disk drive, sometimes called drive **DF0:** (drive floppy 0). Turn on your Amiga. The booting process begins immediately.

- After a while the Workbench screen appears. The loaded Workbench disk is represented by an icon on the screen. Move the mouse pointer onto this icon. Click on this icon by pressing and releasing the left mouse button. Press and hold the right mouse button. Workbench 2.0 users should move the mouse pointer to the `Icon` menu title and select the `Copy` item from this menu. Workbench 1.3 users should move the mouse pointer to the `Workbench` menu title and select the `Duplicate` item from this menu. Release the right mouse button.

- Now the Amiga asks you to insert the SOURCE disk which you would like copied (the FROM disk). You already have that disk in the drive, so click on the `Continue` gadget.

- Have a blank, unformatted disk ready to become your backup Workbench disk. This is the disk that the `Disk Copy` function refers to as the TO disk. Check the write protect of the TO disk; you should not be able to see through the corner of the disk, like you could with the original Workbench disk.

- During the copying process, the Amiga may ask you to exchange the FROM (source) and TO (destination) disks several times, depending on how much memory is available. **Never remove a disk from a disk drive when the drive light is on!!** You

could lose data, and even destroy the disk! A window on the screen tells you when the process is done.

- The difference between the original and the backup disk is that the backup appends the words "Copy of" in front of the original name. Therefore, if the original Workbench disk is named **Workbench x.x**, the new disk has the name **Copy of Workbench x.x**. Remove this extension using the `Rename` item from the `Icon` menu (the `Workbench` menu in 1.3). Use the <Del> and <Backspace> keys to delete the "Copy of" text and press the <Return> key. DOS always distinguishes between the two disks by the date and time of creation assigned to each disk. These details are always stored on the backup.

- Take the original Workbench disk and put it in a safe place. Anywhere far away from moisture and magnetic objects will work (a linen closet, an unused desk drawer, etc.). Use the backup you have made as your Workbench disk.

1.4 Introduction to the Shell

Remove all disks from any disk drives you have connected to the Amiga. Press and hold the <Ctrl> key, the <Commodore> (sometimes called the left <Amiga> key) and the right <Amiga> key to reset the Amiga. Wait until the icon requesting the Workbench disk appears on the screen. Insert your backup copy of the Workbench disk. The Amiga system boots and the Workbench screen appears.

Starting the Shell

Double-click on the `Workbench` disk icon, the `Workbench` disk window opens. Look for the `Shell` icon. Double-click on this icon. The `Shell` loads, and a window named `AmigaShell` (`NewShell` in 1.3) appears on the screen. In our explanations we will refer to both versions as the `Shell` window.

The `AmigaShell` window has some of the attributes of a normal window on an Amiga. It has a drag bar (which allows you to move it around the screen); a sizing gadget (which allows you to change the window's size); a depth gadget (1.3 has a front gadget and a back gadget) for moving the window into the foreground or background of the screen. Workbench 2.0 also has a zoom gadget which toggles the window between full size and last size. Workbench 2.0 AmigaShell windows also have a close gadget, the Workbench 1.3 `NewShell` window has no close gadget: You must use a `Shell` command to close the window in Workbench 1.3 (more on this in Section 1.8).

The only thing displayed in the `NewShell` window is the *DOS prompt*. This consists of a process number (1.), the current drive (SYS:) and a greater-than character (`1.SYS:>`). This character tells you that the computer is ready to receive and execute commands from the keyboard. A cursor waits beside the prompt for your input.

1.5 The First Command

All inputs in the `Shell` must be entered by pressing the <Enter> or <Return> key (some Amigas have <↵> embossed on this key). Since both keys perform the identical function, we refer to the <Return> key for the duration of this book.

<Backspace> and <Return> keys

If you press the <Return> key without entering a command, the prompt appears one line down from its previous location. Unfortunately, you cannot use the four cursor keys to move the cursor to a particular line within the window. All commands must be completely typed out every time they are used. In the input line itself, single characters that have been input can be erased from right to left using the <Backspace> key (some Amigas have <←> embossed on this key) above the <Return> key. An entire line can be erased by holding down the <Ctrl> key and pressing the <X> key (this is called "pressing <Ctrl><X>," and will be used throughout this book to describe key combinations involving the <Ctrl> key).

Only available commands can be executed. Enter the following:

```
files
```

Remember to press the <Return> key at the end of the line. AmigaDOS responds with:

```
Unknown command
```

Only commands available as programs in the disk drive can be executed, AmigaDOS 2.0 can also execute internal commands. This is the special feature of AmigaDOS. The `Shell` receives the command (program name) from the user, searches the disk drive for a file by that name, loads the file into memory and executes it. This means that the `Shell` can execute programs as well as AmigaDOS commands.

Move the mouse pointer to the top of the `Shell` window. Press and hold the left mouse button and drag the window to the top left of the screen. Release the left mouse button. Move the pointer to the sizing gadget at the lower right corner of the `AmigaShell` window. Press and hold the left mouse button and drag the sizing gadget to the bottom right of the screen. Release the left mouse button. Workbench 2.0 owners can simply click on the zoom gadget, next to the depth gadget, to toggle the window full size.

DIR

We'll begin with a relatively simple but very important command, and list other commands as you gain experience in AmigaDOS and the Shell. The name of this first command is DIR (directory). DIR displays a list of the files contained in the specified disk drive (floppy disk, hard disk or RAM disk). Enter the following (remember to press the <Return> key when you're done entering the command):

```
DIR
```

It doesn't matter whether you enter uppercase or lowercase characters in the Shell. Shell commands even accept mixed case letters.

After a while, the Shell displays the contents of the internal disk drive (drive **DF0:**). This list is the directory of the Workbench disk.

The names don't appear on the screen very quickly at first. Soon the names start flying by on the screen. Press any key to stop the display, and press the <Backspace> key to resume the display.

The display should be similar to the following. Your display may differ—don't worry if it does; remember the Amiga is an ever expanding system and new features are continually being added.

```
1.SYS:>dir
    Trashcan (dir)
    Rexxc (dir)
    Expansion (dir)
    Libs (dir)
    WBStartup (dir)
    Prefs (dir)
    Fonts (dir)
    C (dir)
    Devs (dir)
    S (dir)
    L (dir)
    Utilities (dir)
    System (dir)
    .info                   Disk.info
  Expansion.info            Prefs.info
  Shell.info                System.info
  Trashcan.info             Utilities.info
  WBStartup.info
1.SYS:>
```

1.6 Directory Structure

Data files

You may recognize some of the filenames displayed by the `DIR` command; while others may be unfamiliar to you. *Info files* contain icon data, date and time information and comments. You cannot see some files as icons on the Workbench screen because these files don't have matching `.info` files. However, you don't need `.info` files when you work in AmigaDOS.

Some other file entries, shown from the Workbench as drawer icons, have extensions of (`dir`) when you view them using the `DIR` command. The directory (or drawer) structure by which AmigaDOS handles the data files is the same for both the Workbench and AmigaDOS. You can't see all the data files in the `Shell` at once, either. The `DIR` command only displays the *root* (main) directory of a disk for now.

This form of data file management is often referred to as a *tree structure*. The main directory serves as the trunk, and the subdirectories extend from this trunk like the branches of a tree. Each subdirectory can either contain data files or another subdirectory. There is almost no limit to the number of subdirectories you can have.

Subdirectories

How do you reach other subdirectories? From the Workbench it's no problem: a subdirectory appears as soon as you double-click on a drawer. If more drawers appear in this new subdirectory window, you can access their contents in the same way.

If you want to look at the contents of a particular subdirectory from AmigaDOS, you must append a *path* to the `DIR` command. This path describes the "access route" through directories and subdirectories to get to a particular file or directory. The simplest path is to simply provide a directory name. Enter the following (press the <Return> key at the end of the input):

```
DIR System
```

The `DIR SYSTEM` command displays the directory of the `System` drawer on the Workbench disk.

The names shown are actual data files—no (`dir`) extensions appear. Since there are no more (`dir`) names, we cannot go deeper in this branch of the tree. We can only access files in this directory.

Let's look at a directory (drawer) that we normally can't see from the Workbench. The `Devs` directory has no added `.info` file, which is why you can't normally see it in the `Workbench 1.3` window,

Workbench 2.0 does have an option for viewing all diskette files. However, we can view the contents of this directory from AmigaDOS. Enter the following command:

```
DIR Devs
```

This command displays the following directories and files (Workbench 1.3 will also contain a `ramdrive.device` and a clipboard directory):

```
1.SYS:>dir devs
   Printers (dir)
   Keymaps (dir)
 clipboard.device                MountList
 narrator.device                 parallel.device
 printer.device                  serial.device
 system-configuration
1.SYS:>
```

You'll immediately see that there are two more directories contained within this directory. You can easily view one of these directories by adding a slash (/) character and the name of the desired directory. You are still in the main directory; so enter the following to read the `printers` directory inside the `devs` directory:

```
DIR Devs/Printers
```

Don't confuse the slash character (/) with the backslash character (\). The result of this command looks something like this for Workbench 1.2 users (1.3 and 2.0 users will find these printers on the Extras diskette; they should insert the Extras diskette and enter: `DIR "Extras xx:devs/printers"` where `xx` is their version number):

```
1.SYS:>dir devs/printers
  Alphacom_Alphapro_101          Brother_HR-15XL
  CBM_MPS1000                    Diablo_630
  Diablo_Advantage_D25           Diablo_C-150
  Epson                          Epson_JX-80
  generic                        HP_LaserJet
  HP_LaserJet_PLUS               ImagewriterII
  Okidata_292                    Okidata_92
  Okimate_20                     Qume_LetterPro_20
1.SYS:>
```

The underscore character (_) shown above is located on the keyboard by pressing <Shift><->.

The `Preferences` editors retrieve the data needed to drive different types of printers from this directory. No further subdirectories are available from this directory. This directory is one of the deepest subdirectories on the Workbench disk.

Drive specifier A complete path usually contains the name of the disk or the *disk drive specifier*. When you begin, AmigaDOS defaults to **DF0:** (the internal disk drive). This part of the path is optional. If you have two or more disk drives, you can access them with the `DIR` command as well as using the drive specifier. The disk drive specifier must begin the path statement. In the simplest case (no path statement), `DIR DF1:`, for example, displays the main directory of a disk in the first external disk drive. Hard disk users call their device **DH0:**. Statements referring to subdirectories always follow the colon:

```
DIR DH0:TEXT/LETTERT/BILL
```

Unfortunately, if you have only one disk drive connected to your Amiga, you can't just load any disk you want and look at the directory. If you remove the Workbench disk, insert another disk and enter a `DIR` command, the `Shell` requests that you insert the Workbench disk. We'll explain this problem in more detail in Chapter 3. All you need is a single disk drive for this chapter to try out the functions.

The `System` directory you viewed earlier showed some commands that can be accessed as AmigaDOS commands, but aren't necessarily AmigaDOS commands themselves. The actual AmigaDOS commands are in a different directory.

You can view the AmigaDOS commands by looking in directory `C:` of the Workbench disk. Enter the following command to view the commands located on the diskette:

```
DIR C
```

AmigaDOS 2.0 users should enter the following command to view the AmigaDOS internal commands:

```
RESIDENT
```

1.7 Command Templates

Every AmigaDOS command has a built-in help function called the *command template*. Because these commands are so powerful, even an experienced AmigaDOS user can forget the syntax of a command. If the syntax is incorrect, AmigaDOS 1.3 responded with one of these messages:

```
Bad args (or) Bad arguments
```

When AmigaDOS 2.0 was completely rewritten the error messages for many commands were also greatly improved. You could refer to Chapter 10 of this book to find the correct syntax, but it's often much faster to call the command template for the command.

Enter the AmigaDOS command, followed by a space and a question mark, then press the <Return> key. AmigaDOS displays the argument template for the desired command. Enter the following:

```
DIR ?
```

AmigaDOS displays:

```
DIR,OPT/K,ALL/S,DIR/S,FILES/S,INTER/S:
```

The command template is easy to read once you learn the coding. DIR is the *keyword* (command)—this must appear first in the syntax.

A comma separates arguments from each other in the argument template. These shouldn't be entered when you type the command itself. Therefore, `DIR` has five arguments available: OPT/K, ALL/S, DIR/S, FILES/S and INTER/S. Arguments can also contain *qualifiers* (control characters) preceded by a slash (/) character. The second argument of the `DIR` command includes the word `OPT`. `OPT` is an abbreviation for OPTIONAL. This means that `OPT` is a form of input which can be included or omitted.

The final section of the second argument is `/K`. The letter K is an abbreviation for Keyword. This options keyword must be given in the command.

The colon (:) at the end of the argument template is important, but it's not part of the argument template (more on this at the end of this section).

Possible qualifiers that can appear in an argument template in 1.3 and 2.0:

/A (Argument) This qualifier always requires a certain argument. If you omit the argument, the command cannot execute.

/K (Key) The qualifier's name must appear as input (e.g., OPT in the `DIR` example above), and a keyword must appear as well. The parameters allowed and the functions executed depend on the respective `Shell` command (see Chapter Two for more details).

/S (Switch) This qualifier needs no arguments. It acts as a switch (toggle) for a command. In commands switches operate similar to a light switch--they turn a command on or off or switch the command to another mode.

Possible qualifiers that can appear in an argument template only in AmigaDOS 2.0:

/N (Numeric) This qualifier indicates that a numeric argument is expected (AmigaDOS 2.0 only).

/M (Multiple) Multiple arguments can be included. In 1.3 commas were used to signify multiple arguments. Multiple arguments must be separated by spaces. This has been updated in AmigaDOS 2.0, also the number of arguments is unlimited in AmigaDOS 2.0.

/F (Final) The argument is the final argument. This allows using strings without enclosing them in quotation marks (AmigaDOS 2.0 only).

, (comma) The command takes no arguments (AmigaDOS 2.0 only). In AmigaDOS 1.3 the comma was used to show multiple inputs.

If none of the qualifiers appear in an argument, then the parameter accompanying the command (if any) can be identified from its position within the command line. For example, the command below has no qualifiers. It tells AmigaDOS to display directory `C` of drive **DF0:** on the screen:

```
DIR DF0:C
```

AmigaDOS 1.3 In AmigaDOS 1.3 it's possible that an argument can be unnamed. The `DELETE` command (which we'll discuss in detail later) has a number of different arguments. Enter the following in the `Shell` of AmigaDOS 1.3:

```
DELETE ?
```

The `Shell` of AmigaDOS 1.3 responds with:

```
,,,,,,,,,,ALL/S,Q=QUIET/S:
```

The ten commas at the beginning of the argument template imply that you can delete up to ten files at a time.

AmigaDOS 2.0 AmigaDOS 2.0 has greatly improved on this method by allowing unlimited arguments to be separated by spaces in the input line.

The ALL/S argument means that if you precede the word ALL with the name of a directory, the command deletes all the files on the directory and the directory itself. The following input deletes all files from the directory in drive **DF0:** named NORTON, then the NORTON directory itself:

```
DELETE DF0:NORTON ALL
```

If you entered this command and had a set of files inside a directory named NORTON in **DF0:**, AmigaDOS would report the status of the deleted files on the screen. The Q=QUIET/S argument switches on the display of the file deletion process. The equal sign between the Q and the QUIET means that you can either use the word or the letter as the argument. The following command deletes all the files from the NORTON directory in drive **DF0:**, then deletes the NORTON directory. This command and the above DELETE command perform the same function. However, the command below suppresses the list of deleted files on the screen:

```
DELETE DF0: NORTON ALL QUIET
```

This version of the DELETE command does the same thing (notice the use of the letter Q instead of the word QUIET):

```
DELETE DF0: NORTON ALL Q
```

Arguments An argument introduced by the user through its name can be placed anywhere within the input line. For instance, the COPY command includes the arguments FROM and TO/A, among others. Both of the following command sequences perform the same function—copying the letters file from drive **DF0:** to a file on drive **DF1:** named text:

```
COPY FROM DF0:letters TO DF1:text

copy TO DF1:text FROM DF0:letters
```

It doesn't matter whether the command names and arguments are entered in uppercase or lowercase letters.

After command parameters are displayed in a command template, the cursor reappears in the same line as the command template following the colon. You can now enter an argument or set of arguments without re-entering the command keyword.

When working in AmigaDOS, if you're not 100% sure which command uses which arguments, enter the command, a space and a question mark to see the command template.

1.8 Quitting the Shell

We mentioned in Section 1.4 that AmigaDOS 1.3 Shell windows have no close gadget. AmigaDOS uses a command instead of a close gadget to exit and return you to the Workbench.

ENDSHELL The ENDSHELL command closes the Shell window currently active.

Enter the following (again, remember to press the <Return> key at the end of the input):

```
ENDSHELL
```

The Shell window immediately disappears and the Amiga returns you to the Workbench.

This introduction to working with the Shell has made you familiar with its basic operation. The following chapters systematically explain all the currently available commands.

ENDCLI The ENDCLI command also closes the Shell window currently active. This command has been retained for compatibility with earlier versions of AmigaDOS.

AmigaDOS 2.0 AmigaDOS 2.0 users can also click on the close gadget located in the AmigaShell window.

2.
AmigaDOS Commands

2. AmigaDOS Commands

This chapter lists the AmigaDOS commands in detail. The commands appear in order of difficulty and importance and not in alphabetical order. The easier to learn commands appear first, this way you won't immediately confront the relatively difficult commands, which can confuse you if you don't have the background information needed for these commands. The Quick Reference chapter lists the commands in alphabetical order.

Section 2.1 describes all the commands that fall under the general heading of disk drive and file management. Here you'll find the commands which access the floppy or hard disk drives, and files stored on these disk drives.

Section 2.2 describes the commands which access the operating system in some way or another. A typical member of this group is the `Date` command which deals with the system date.

Section 2.3 describes commands used in *script files*. These are similar to batch files on MS-DOS computers. Script files perform multiple commands, saving the user the effort of repeatedly typing in the same command sequences. The `Startup-sequence` is a script file. Script files are one of the most powerful features of AmigaDOS.

Finally, Section 2.4 explains two comprehensive commands. These commands, `ED` and `Edit`, invoke two different *text editors* which are used to create script files.

The following sections contain a great deal of descriptive material. We recommend that you try out commands (when you can) as much as possible while you read. This will help you understand the functions of the commands. Use a backup copy of the Workbench disk, do not use the original disk. If you don't have a backup, go to Chapter 1 and make one. You may also want to keep a blank, unformatted disk around for testing some commands.

Note: AmigaDOS is constantly being improved and updated, the updates usually contain a number of added arguments and commands, making it much more versatile than previous versions. To show the history of AmigaDOS and make this guide usable to all Amiga users we will first present the 1.2 version of the command, then the 1.3 implementation and finally the AmigaDOS 2.0 version. Any differences in the commands will appear after the general description of the command. Each version of the AmigaDOS command will be preceded by **Workbench 1.2 implementation:**, **Workbench 1.3 implementation:** and **Workbench 2.0 implementation:**.

2.1 Disk and File Management

This section lists the commands used for handling files and managing the Amiga disk drives.

2.1.1 FORMAT

Workbench 1.2 implementation:

Syntax: `FORMAT DRIVE <disk> NAME <name> [NOICONS]`

A disk must be *formatted* or *initialized* before you can use it on an Amiga. Formatting prepares a disk so that the Amiga can read data from and write data to the disk. In Workbench 2.0 the `Icons` menu contains an item named `Format Disk...` (in 1.3 the `Workbench` menu contains an item named `Initialize`). AmigaDOS recognizes unformatted disks immediately and places a DFx:??? name under the disk icon on the Workbench (1.3 names the disk as `BAD`).

The AmigaDOS `Format` command requires more information than the Workbench's `Format Disk...` item (1.3 `Initialize`). You must give arguments specifying the disk drive and the additional details about the new disk's name. To format a disk in disk drive **DF0**, you must input:

```
FORMAT DRIVE DF0: NAME Example NOICONS
```

The `NAME` argument can be up to 30 characters long. Names that long can cover up other disk names while in the Workbench, so we recommend that you use shorter disk names. If you include blank spaces in the `NAME` argument, the argument must be enclosed in quotation marks. Incidentally, that applies to all work with the AmigaDOS arguments which cannot contain spaces, or the argument must be enclosed within quotation marks. AmigaDOS 2.0 also includes the /F switch to signify a final argument which can contain spaces. For example:

```
FORMAT DRIVE DF0: NAME "My Text" NOICONS
```

NOICONS The NOICONS argument suppresses the creation of the Trashcan icon which normally appears in any disk window on the Workbench. The Trashcan is unnecessary when using AmigaDOS.

The DRIVE and NAME arguments must be input every time you use the Format command. If the syntax was entered correctly, AmigaDOS loads the Format command and the window displays:

```
Insert disk to be initialized in drive DF0: and press Return
```

Now that the Format command has been completely loaded, those who have only a single disk drive can now remove the Workbench disk, and insert the disk to be formatted. Before you press the <Return> key, however, you should know that any data previously stored on this disk is destroyed when you format the disk. If you wish to cancel the procedure, press <Ctrl><C> (hold down the <Ctrl> key and press the <C> key) and press the <Return> key. AmigaDOS then responds with a *** Break.

If you wish to continue, press the <Return> key alone. The Amiga formats the disk in the drive. AmigaDOS displays which *cylinder* (track set) is currently being formatted. Each cylinder consists of two concentric tracks on the disk, about 0.5 mm wide on opposing sides (surfaces) of the disk. Each track can then be broken down into 11 sectors, each of which can contain 512 bytes of data. Since the disk possesses 80 cylinders (or 160 tracks) altogether, the entire disk capacity amounts to 880K:

```
(80 * 2 * 11 * 512)/1024 Byte = 880K
```

You don't need to format a disk if you plan to use the DISKCOPY command. The DISKCOPY command automatically formats the disk if it has not been formatted.

Workbench 1.3 implementation:

Syntax: FORMAT DRIVE <disk> NAME <name> [NOICONS] [QUICK] [FFS] [NOFFS] [INHIBIT]

The QUICK, FFS and NOFFS arguments are new additions to Version 1.3. The QUICK argument speeds up the formatting operation so that it only takes a few seconds on a pre-formatted disk (a disk that has been formatted once before). Only the tracks that contain the Root block and the Boot blocks are formatted. A standard disk format (without the QUICK argument) takes about two minutes.

Root block
Boot blocks

The Root block (found on cylinder 40, side 0, sector 880) is the block containing the root of the directory structure. The QUICK option writes an empty directory to the disk. This file must not be erased. Formatting of the Boot blocks (cylinder 0, side 0, sectors 0 and 1) renews the boot program so the Amiga can eventually auto boot. This also eliminates any viruses that may have gotten into the boot blocks.

FFS *and* NOFFS

The FFS and NOFFS arguments are interconnected. They create the desired file system for single partitions when formatting a hard disk. Adding the FFS argument puts the new and faster FastFileSystem into use. The slower original FileSystem is used if NOFFS is entered.

A partition must be entered in the MountList if you want to run under the new FastFileSystem. This MountList is found in the Devs drawer on the Workbench disk. Each partition of the hard drive not autoconfigured has an entry here. Before the new FFS partitions can be used, the following lines must be added to each partition entry:

```
FileSystem = 1.FastFileSystem
GlobVec = -1
DosType = 0x4444F5301
StackSize = 4000
```

A requester displays the message Not a DOS Disk... if such a partition is placed inside the Startup-sequence the first time. It can be removed by clicking on the Cancel gadget. The partitions must be re-formatted under the new File system. In case the FastFileSystem is attached to a partition, all you have to do is re-format the partition. The entire hard disk must be re-formatted if you wish to change the size of the partition (LowCyl to HighCyl). Save the contents of the formatted partitions to floppy disks before performing this format.

Inhibit

Inhibits disk access while formatting.

Workbench 1.3.2 implementation:

INHIBIT was made automatic and NOFFS removed since it was not very useful. The error messages were also improved.

Workbench 2.0 implementation:

This operation is identical to Workbench 1.3.2, but the command has been optimized for compactness and speed.

2.1.2 DIR

Workbench 1.2 implementation:

Syntax: `DIR DIR,OPT/K`

This command displays the files and directories on a disk, lists subdirectories and the files within these subdirectories.

You read about this command in Chapter 1 and used it to view AmigaDOS' file structure. In reality, this command does much more. The command template of `Dir` looks like this:

```
DIR DIR,OPT/K
```

The `DIR` argument represents the exact path of the desired directory. This argument initially defaults to the root directory of the Amiga's drive **DF0:** (the internal disk drive). Therefore, if you want to read the directory on another device, you must supply the drive specifier as the `DIR` argument (e.g., **DF1:**, **RAM:**, **DH0:**, **JH0:**). The colon (:) at the end of each drive specifier tells DOS that the name is in fact a device name. The `DIR` argument may be followed by any path to a particular directory (drawer).

Example: You want to view the `Letters` directory. The `Text` directory in the disk in drive **DF1:** contains the `Letters` directory. The following sequence accesses that directory:

```
DIR DF1:Text/Letters
```

Further subdirectories can be accessed at any time simply by adding another slash character and directory name to the `DIR` argument.

If you omit arguments and qualifiers, the `DIR` command displays the current directory. The use of the `CD` command (change directory) dictates the current directory (see Section 2.1.3).

The capabilities of the `DIR` command expand with the use of the `OPT` argument. Four qualifiers can be used with `OPT`: `A`, `D`, `I` and `AI`.

A The `A` (`All`) qualifier displays all of the files and directories in the current disk. You can view every directory and every file: AmigaDOS lists each directory then the directory's contents in indented format. This option is very helpful if you cannot find a certain file. However, this option also creates a stream of data which quickly fills, then scrolls the screen. Pressing any key stops the scrolling; pressing the <Backspace> key continues the directory display. The following

command displays all the files on the disk in drive **DF1:** (the second disk drive):

```
DIR DF1: OPT A
```

D The D (DIR) qualifier lists only the directories of the current disk. This is useful for quick searches for a specific directory, without listing the root directory files in addition to the directories.

I The I (INTER) option runs the DIR command in *interactive mode*. This mode allows the user complete control of the directory output. When DIR is invoked in interactive mode, AmigaDOS prompts with a question mark after it displays each file. The user has the following options for controlling the display:

<Return> Continues interactive output (displays the next file or directory) name.

Del<Return> Typing this word and pressing the <Return> key deletes the file currently displayed on the screen. Notice that you enter the letters D E L, not press the <Del> key (see Delete). You can only delete empty directories (i.e., the directory you want to delete contains no files or subdirectories). If you try to delete an occupied directory using this command, AmigaDOS responds with the error message Error code 216 then displays the error.

<E><Return> Enters a deeper directory level. The directory output resumes upon entry to this directory.

<B><Return> Moves back up to a higher directory level (closer to the main directory). If you try to move to a level higher than the main directory, the DIR command ends.

<T><Return> Types (displays) a plain ASCII text file in the Shell window. If you use the <T> key to display programs or AmigaDOS commands, you'll only get garbage on the screen. Pressing <Ctrl><C> stops the output and returns you to interactive mode. If the output still isn't back to normal, press <Ctrl><O> to restore the Amiga's normal character set.

<?><Return> Displays the command template of commands available in interactive mode. The template for directories appears on the screen as follows:

```
B=BACK/S,DEL=DELETE/S,E=ENTER/S,Q=QUIT/S:
```

<Q><Return> Quits interactive mode and returns you to the Shell prompt.

If you enter an incorrect command in interactive mode, AmigaDOS responds with the message Invalid response-try again?: after which you can re-enter the command.

AI The AI (All Interactive) qualifier displays all directory entries interactively.

Workbench 1.3 implementation:

Syntax: DIR DIR,OPT/K,ALL/S,DIRS/S,FILES/F,INTER/S

This new implementation of the DIR command adds the three arguments ALL, INTER and DIRS which perform the same functions as the A, I, D and AI arguments. The FILES argument is new and allows only files to be displayed, subdirectories will not be displayed. The OPT argument must be left off when using these new arguments.

ALL Displays all directory entries in the current disk.

DIRS Displays the names of the directories only in the current disk. This argument displays the following output of the Workbench 1.3 disk:

```
1> dir dirs
   Trashcan (dir)
   c (dir)
   Prefs
   System (dir)
   l (dir)
   devs (dir)
   s (dir)
   t (dir)
   fonts (dir)
   libs (dir)
   Empty (dir)
   Utilities (dir)
1>
```

FILES Displays the files in the current disk directory, subdirectories will not be listed.

INTER Displays the current disk directory in interactive mode (identical to I qualifier). The INTER argument adds a new option to the command list:

C=COM/s,COMMAND
A new option has been added to the interactive mode which allows the user to execute an AmigaDOS command either directly or through the RUN command. This function can be useful when you are in the directory of a data file you want printed. For example, you're in the S: directory and you want to print out the Startup-sequence file. Enter <C> or COM and press the <Return> key. Interactive mode requests the command. Enter the following to print out the Startup-sequence and continue in interactive mode:

```
Command ?:run type df0:s/startup-sequence to prt:
```

Do not use the diskette FORMAT command while in interactive mode; no confirmation will be allowed.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3.2, but the command has been optimized for compactness and speed.

2.1.3 CD

Workbench 1.2 implementation:

Syntax: `CD DIR`

The CD (Change Directory) command allows you to move to directories either above or below the current directory. Let's review the idea of a tree structure used in disks. CD lets you move from your current directory's location to another branch. Once you use the CD command, the directory to which you move becomes the current directory.

Examples: The following command (CD without arguments) displays the current directory:

```
CD
```

If you start AmigaDOS immediately after startup, this response is the disk drive specifier (e.g., Workbench1.3, Workbench2.0, DF0:).

The following command makes the System directory the current directory (if the System directory is immediately accessible):

```
CD System
```

Entering another CD without arguments displays the new current directory (e.g., SYS:System, Workbench2.0:System).

All AmigaDOS commands now refer to the current directory. For example, if you enter DIR AmigaDOS displays the System directory instead of the main directory.

There are two ways to display the System directory from the main directory. The first method displays the System directory's contents and returns you to the main directory:

```
DIR DF0:system
```

The second method changes the current directory to the System directory and displays the directory's contents:

```
CD DF0:system
DIR
```

The main directory

The second method doesn't automatically return you to the main directory. You must use one of CD's single-character arguments to move up toward the main directory.

/

This character moves you one directory up in the *hierarchy* of directories. Multiple slash characters move you up as many directories as there are slashes. The following command moves you one directory up (notice the space between the CD command and the /):

```
CD /
```

The following sequence moves you up two directory levels toward the main directory (notice the space between the CD and the first / but no space between the two slashes):

```
CD //
```

Note:

If you enter more slash characters than there are directory levels, AmigaDOS responds with the message Can't find and the number of slashes you entered.

:

This character moves you directly to the main directory when used in conjunction with CD (notice the space between the CD and the colon):

```
CD :
```

There are some minor differences between the two arguments. When AmigaDOS searches for a pathname using a disk name as the DIR argument instead of a drive specifier (e.g., using CD Workbench: instead of CD DF0:), AmigaDOS doesn't care which drive the disk is in, as long as the disk is in one of the drives. If AmigaDOS cannot find the disk name, it displays a requester asking you to insert the specified disk in any drive.

Note:

AmigaDOS is extremely choosy about the way that it reads and accepts filenames; it will not accept some characters in directory names or filenames. For example, if you have a directory named Test Drawer and you enter cd Test Drawer, the Amiga responds with the too many arguments (or Bad arguments) error message, even if the directory is available. AmigaDOS will not accept the space character. There are three ways to avoid this problem: Rename the file to a single word filename (e.g., TestDrawer); use the underscore character (_) to separate the two words instead of a space (e.g., Test_Drawer); or enclose the directory name in quotation marks when calling the CD command (e.g., cd "Test Drawer"). The easiest method is to use

one word filenames. The underscore character (<Shifted> minus sign) allows you to separate words, making filenames more readable.

Often you must specify the drive you want to access. For example, if the disk in drive **DF0:** has the name `My_data` and you want to get to the main directory of that disk, all you have to do is enter the following:

```
CD My_data
```

The following gives the same result, and is easier to remember than a disk name:

```
CD DF0:
```

The latter example requires that you have the correct disk inserted in drive **DF0:**—DOS will not look for a disk name unless you specify one. Here lies the basic difference in the `CD` command, because while `CD DF0:` automatically returns you to the main directory of the disk in the internal disk drive, `CD :` always returns you to the main directory of the currently active disk.

Example: Drive **DF0:** contains a Workbench disk and drive **DF1:** contains a disk named `Work_data` which includes a file named `Customers`. Entering the following changes to this directory:

```
CD DF1:Customers
```

Entering the `CD` command without arguments displays the following; `Work_Data:Customers`. When using the complete pathname the disk can be put into any drive without further confusion. It is all the same to the Amiga. Now the difference between `CD :` and `CD DF0:` becomes obvious: a `CD :` makes the main directory the current directory, `CD DF0:` makes drive **DF0:** the main directory.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3.2, but the command has been optimized for compactness and speed. The command has also been made an internal command.

2.1.4 MAKEDIR

Workbench 1.2 implementation:

Syntax: MAKEDIR /A

This command performs the same function as the New Drawer item in the Window menu of Workbench 2.0. There you can make a new drawer, name the new drawer whatever you want and drag files into the new drawer. Workbench 1.3 users duplicated the Empty drawer to create a new directory. An example of this is the Expansion drawer on the Workbench disk. The greater the capacity a disk drive has, the more powerful the MAKEDIR command becomes: The ability to create directories on high-capacity hard disks is vital to keeping disks organized. This command is used to keep a hard disk organized.

The MAKEDIR command is very easy to use. It requires only one path statement, followed by a slash and a name for the new directory:

```
MAKEDIR DF1:System/Monitor
```

It is important that all of the paths specified in the command exist. You cannot create more than one directory at a time.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Syntax: MAKEDIR NAME/M

AmigaDOS 2.0 allows multiple arguments to be passed to the MAKEDIR command. This allows the easy creation of multiple (drawers) subdirectories. Multiple arguments must be separated by spaces. If no arguments are passed to the command an error message, No name given, is displayed.

2.1.5 DELETE

Workbench 1.2 implementation:

Syntax: `DELETE ,,,,,,,,, ALL/S,Q=QUIET/S`

This command removes unnecessary directories or files from a disk or RAM disk. The following command deletes the `Extra_drawer` directory from the `C` directory on the disk in drive **DF0:**:

```
DELETE DF0:C/Extra_drawer
```

AmigaDOS cannot delete a directory which still contains data. If you try to delete a directory that still had files in it, AmigaDOS displays the message `Not Deleted: directory not empty`. You must move or delete these files before you can delete the directory.

Note: Be very, very careful with the `DELETE` command; it's easy to delete the wrong file. Unlike the `Trashcan` on the Workbench, once you delete a file you can't get it back.

Wildcards The Amiga *wildcard* is very useful with the `DELETE` command (see Chapter 5 for more information). Like a wildcard in poker, the file wildcard acts as a match for many files. This wildcard is made of two characters—a number sign (#) and a question mark (?). The following command deletes all the files beginning with `test`:

```
DELETE DF0:test#?
```

There's a second way of deleting more than one file. The `DELETE` command evaluates a maximum of ten files separated from one another by a single space:

```
DELETE DF0:Utilities/Notepad DF1:system/say.info
```

How can you squeeze ten path specifiers onto one line? You don't have to. The cursor can move up to three screen lines for one command line. You press the <Return> key when you're done entering data.

The `QUIET` argument keeps the file deletion process from appearing on the screen. The following command deletes all the files and directories from the `Utilities` directory in drive **DF0:**, then deletes the directory itself without telling the user what it's doing:

```
DELETE DF0:Utilities ALL QUIET
```

Workbench 1.3 implementation:

Syntax: Version 1.2 and Version 1.3 syntaxes of this command are identical.

The new version of DELTE doesn't stop when an entry cannot be found. The following command deletes the file test3 from drive **DF0:**, even if AmigaDOS cannot find the file test2 on the disk:

```
delete df0:test1 df0:test2 df0:test3
```

The old version of the command would have deleted test1 and then displayed an error message.

Workbench 2.0 implementation:

Syntax: DELETE FILE/M/A,ALL/S,QUIET/S,FORCE/S:

FILE/M allows multiple arguments to be included. In 1.3 commas were used to signify multiple argument. Multiple arguments must be separated by spaces. This has been updated in DOS 2.0, also the number of arguments is unlimited in DOS 2.0.

FORCE /S allows the deletion of a file, even though it's in use; use this with extreme caution.

2.1.6 COPY

Workbench 1.2 implementation:

Syntax: COPY FROM,TO/A,ALL/S,QUIET/S

The COPY command is one of the most important and flexible commands for manipulating files using AmigaDOS. This command can copy a single piece of file or a complete directory on any device of your choice that can receive data. Naturally it can also copy within a disk drive. The command template reads:

```
FROM,TO/A,ALL/S,QUIET/S
```

The FROM argument represents a path description for the source data or source file. Because an /A qualifier doesn't exist, there is no input obligation. If the FROM description is wrong, then the actual directory becomes the source file. The TO argument represents the destination path for the copy operation. The description depends on the source data:

a) FROM refers to a single file

In this case the destination path can be any subdirectory you choose within the device, or a device that you specify. It treats the destination device as a drive, so the data is put in the desired directory under the same name. The following example takes the file `test` from directory `C:` of drive **DF0:** and creates a duplicate of the same name in the `C:` directory of the RAM disk:

```
COPY DF0:C/test RAM:C
```

The `C:` directory must already exist in the RAM disk (see the description of the `MAKEDIR` command for details on making directories). If there is already a file in the destination directory named `test`, AmigaDOS overwrites the file. AmigaDOS is consistent in this: It overwrites an existing file without warning.

The following command copies the `Startup-sequence` script file to a printer:

```
COPY DF0:S/Startup-sequence PRT:
```

If you want the copy to have a name other than the one already stated, you have to specify that filename.

If a subdirectory with the same name already exists in this drawer, the copy is placed under the old name, because in the input there isn't a difference between drawers and data names. Here is an example:

```
COPY DF0:C/MAKEDIR RAM:MD
```

This copies the `MAKEDIR` command in the `C:` directory to the RAM disk under the name `MD`. There cannot be an existing subdirectory in the RAM disk named `MD`. If such a subdirectory already exists, then the `MAKEDIR` command is stored under its default name in that directory.

Now we come to the second option of the `FROM` argument.

b) FROM refers to an entire drawer

The destination path must point to a directory onto which you want to copy files. Unfortunately you cannot specify the printer as a destination device. The `COPY` command cannot send multiple files to a printer.

Usually only the data in the drawer itself is copied. Subdirectories are ignored. The command should include the subdirectories you want copied as well. The following command copies the contents of drive **DF0:** onto the hard drive (**DH0:**) into an existing directory named `Games:`

```
COPY DF0: DH0:Games all
```

The DISKCOPY command copies entire disks more efficiently than the COPY command. However, using COPY brings a little order to the disk. When files are edited they may become fragmented on a disk, this means they are scattered over many different tracks. When copied with the COPY command they are copied to the destination disk so they are on tracks that are close to one another. Now the read head of the disk drive does not have to move as far to access the file.

Workbench 1.3 implementation:

Syntax: `COPY FROM,TO/A,ALL/S,QUIET/S,BUF=BUFFER/K,CLONE/S,DATE/S, NOPRO/S,COM/S`

When you want to copy data to a directory that doesn't exist on the destination disk, the new version of the command creates a directory of the same name on the destination disk. The source files are then copied into this directory.

The new COPY command also allows you to print the contents of a directory to a printer. This output may be distorted if the directory does not contain only true ASCII data files.

Now we come to the added arguments mentioned in the command template above:

The BUFFER (or BUF) argument allows the user to allocate a number of 512 byte buffers to be used in the copying process.

The CLONE, DATE, NOPRO and COM arguments represent additional information passed to the copy. The additional information that AmigaDOS prepares for all files and directories state the date in which the file was created, and the protection bits listed under the description of the PROTECT command. Up to 80 characters of comments can be added to a file.

The LIST command allows you to see this information. This is explained in the next section.

The CLONE argument copies the original file's creation date, protection bits and comments to the new file.

The DATE argument copies the original file's creation date to the new file.

The COM argument copies the original file's comments to the new file.

The NOPRO argument suppresses the protection bit information when copying the new file.

Example: The following command copies a data file named `Test` to the RAM disk using the original file's creation date and comments. No protection bits are passed to the new file:

```
COPY Test RAM: DATE COM NOPRO
```

Workbench 2.0 implementation:

Syntax:

```
COPY FROM/A/M,TO/A,ALL/S,QUIET/S,BUF=BUFFER/K/N,CLONE/S,
DATE/S,NOPRO/S,COM/S:
```

FROM/M allows multiple arguments to be included. Multiple arguments must be separated by spaces. The number of arguments is unlimited in AmigaDOS 2.0.

2.1.7 LIST

Workbench 1.2 implementation:

Syntax:

```
LIST DIR,P=PAT/K,KEYS/S,DATES/S,NODATES/S,TO/K,S/K,
SINCE/K,UPTO/K,QUICK/S
```

The `LIST` command lists important file information that the `DIR` command doesn't show.

The `LIST` command displays the following information, filename or directory name, size of file or directory, protection bits, date and time of creation.

Names The filenames and directory names appear on the screen in their order on the disk. `LIST` makes no distinction in names between files and directories.

Size/Dir The next entry in the listing distinguishes files from directories. Filenames list their file sizes in bytes; directories display the word `Dir` in the location reserved for file sizes.

Protection bits The next entry displays the *protection bit* status of each file. All the file entries listed above contain four protection bits. Each protection bit letter represents the following:

```
r    (read)     should allow reading of the file
w    (write)    should allow writing to the file
e    (execute)  should allow execution of the file
d    (delete)   allows entry to be deleted
```

If one or more of the options is suppressed a dash appears in place of that option. A file with the combination rwe- therefore cannot be deleted. The remaining flags (rwe) aren't implemented at the time of this writing. DOS leaves these flags alone.

The PROTECT command described later lets you change the status of these flags.

Time & date The next two entries list the time and date when the file was first created. These date entries always appear if you enter the correct date with the Preferences editor, or if you have an Amiga with a battery-backup real-time clock.

Bottom line At the bottom of the list the number of files and the number of directories on the disk appear, as well as the number of *blocks* (1 block=512 bytes=0.5K) free on the disk.

The following command displays a list of files and directories contained in the current directory of drive **DF0:** (the internal disk drive):

```
LIST DF0:
```

If the Workbench disk is in drive **DF0:** text similar to the following appears on the screen.

```
Trashcan.info              1144 ----rwed 20-Jun-90 17:22:48
Trashcan                    Dir ----rwed 20-Jun-90 04:35:07
Rexxc                       Dir ----rwed 20-Jun-90 04:35:18
Wbstartup.info              824 ----rwed 20-Jun-90 17:22:47
Utilities.info              824 ----rwed 20-Jun-90 17:22:46
System.info                 824 ----rwed 20-Jun-90 17:22:47
Shell.info                  722 ----rwed 20-Jun-90 17:22:47
Prefs.info                 1144 ----rwed 20-Jun-90 17:22:46
Expansion.info              824 ----rwed 20-Jun-90 17:22:46
.info                        87 ----rwed 22-Mar-78 03:43:47
disk.info                   388 ----rwed 20-Jun-90 17:36:37
Expansion                   Dir ----rwed 22-Mar-78 03:44:36
Libs                        Dir ----rwed 20-Jun-90 04:35:53
WBStartup                   Dir ----rwed 20-Jun-90 04:35:57
Prefs                       Dir ----rwed 22-Mar-78 03:49:29
Fonts                       Dir ----rwed 20-Jun-90 04:36:45
C                           Dir ----rwed 20-Jun-90 04:33:37
Devs                        Dir ----rwed 20-Jun-90 04:33:54
S                           Dir ----rwed 20-Jun-90 04:34:01
L                           Dir ----rwed 20-Jun-90 04:34:07
Utilities                   Dir ----rwed 22-Mar-78 03:44:59
System                      Dir ----rwed 23-Mar-78 04:43:38
9 files - 13 directories - 40 blocks used
```

There's more to LIST than you might think. Invoking the command template (LIS ?) displays the following:

```
DIR,P=PAT/K,KEYS/S,DATES/S,NODATES/S,TO/K,
S/K,SINCE/K,UPTO/K,QUICK/S
```

Don't panic! Most of the time all you'll ever need is the LIST command without arguments. Here's an overview of each argument:

DIR The DIR argument lets you specify another directory (e.g., LIST RASM:C).

PAT The PAT argument allows you to use patterns or wildcards. The wildcard (#?) is extremely useful for finding selected entries (e.g., LIST PAT A#? displays only the entries beginning with the letter A).

KEYS This argument returns the starting blocks of the selected programs on the disk (only AmigaDOS "power users" will use the KEYS argument).

DATES The DATES argument enables date display in the format DD-MMM-YY (this output is the default for the List command).

NODATES The NODATES argument disables date display.

TO The TO argument specifies the file or device that should receive the output (e.g., LIST DF0: TO PRT: sends the listing of **DF0:** to the printer).

S The S (substring) argument enables you to search for entries that are arranged according to their *substring*. A substring is part of a name. Chapter 3 lists details about the input #?subname#? in the Pat option.

SINCE The SINCE argument displays all entries created since the specified date. The specified date must be in DD-MMM-YY format, or stated as the words YESTERDAY or TODAY.

UPTO The UPTO argument displays all entries created before the specified date. The specified date must be in DD-MMM-YY format, or stated as the words Yesterday or Today. The following example of SINCE and UPTO includes the YESTERDAY specifier:

```
LIST SINCE 09-JUN-87 UPTO YESTERDAY
```

QUICK The QUICK argument lists only the entry names, as well as the number of blocks remaining.

The following command, which would be entered on one line in the Amiga, searches for the C subdirectory in drive **DF0:** and looks for all the commands that begin with C. The command then looks for all entries created after September 10, 1986 and sends all these entries to the printer (no dates appear on the printout):

```
LIST DF0:C PAT C#? NODATES TO PRT: SINCE
10-SEP-89 UPTO TODAY
```

Workbench 1.3 implementation:

Syntax: `LIST DIR,P=PAT/K,KEYS/S,DATES/S,NODATES/S,TO/K,SUB/K, SINCE/K,UPTO/K,QUICK/S,BLOCK/S,NOHEAD/S,FILES/S, DIRS/S,LFORMAT/K`

There are some very useful arguments added to this version. This version also displayed any comments attached with the FILENOTE command.

BLOCK The `BLOCK` argument displays file sizes in disk blocks instead of bytes.

NOHEAD The `NOHEAD` argument suppresses the display of directory names and creation date. This argument always appears when the `List` command is entered with a directory name (e.g., `LIST DF0:`). In addition, `NOHEAD` disables the display of the closing line (`xx files -yy directories-zz blocks are used`).

FILES The `FILES` argument displays the filenames only.

DIRS The `DIRS` argument displays the directory names only.

LFORMAT The `LFORMAT` argument allows the formatting of List text for use as script files. The output format specification follows the `LFORMAT` argument enclosed in quotation marks:

```
LIST DF0: LFORMAT="..."
```

Any text can be used in the output format specification. When the character string `%s` appears as the output format specification, AmigaDOS inserts the current filename at that point. The following example inserts the filenames listed in directory `C:` in the resulting output:

```
Input:
LIST DF0:c LFORMAT="This is the %s command"
Output:
This is the Run command
This is the Fault command
This is the Install command
This is the Stack command
This is the Prompt command
This is the Else command
This is the Status command
This is the Ed command
This is the BindDrivers command
(...)
```

The following use of the LFORMAT argument can be used to create a script file that removes all of the d (delete) protection bits in drive DF0:'s Text directory:

```
list >Script_file df0:Text LFORMAT="protect %s -d"
```

The result could be something like this:

```
PROTECT Text_1 -d
PROTECT Text_2 -d
PROTECT Text_3 -d
PROTECT Letter_1 -d
PROTECT Letter_2 -d
```

This file can be executed directly using the EXECUTE Script_file command (see the description of the EXECUTE command).

The %s string can appear more than once in the output format specification. If two %s are used the current filename appears in both locations. When three of these strings are used, the second and third occurrences display the filename while the first occurrence displays the path of the specified directory. The following example creates a script file that will copy a backup of the commands in the C: directory to the directory named Directory:

```
Input:
LIST >Script_file c: LFORMAT="COPY %s%s TO
directory/%s.BAK"
Output:
COPY c:RUN to directory/Run.BAK
COPY c:FAULT to directory/Fault.BAK
COPY c:INSTALL to directory/Install.BAK
COPY c:STACK to directory/Stack.BAK
COPY c:PROMPT to directory/Prompt.BAK
COPY c:ELSE to directory/Else.BAK
(...)
```

When four %s are used, the occurrences alternate between the specified path description and filename.

The Version 1.3 List command still has more functions. The wildcard features increased flexibility. It's now possible to use the wildcard with the path description. The following example lists all the files in the C directory beginning with the letter m:

```
LIST DF0:C/m#?
```

The Version 1.2 LIST command would display this message:

```
Can't examine "df0:c/m#?": object not found
```

Protection bits In addition to the existing `rwed` protection bits, Workbench 1.3 adds three new protection bits: `h` (not implemented), `s` (Script), `p` (Pure) and `a` (Archive). See the description of the `PROTECT` command for details about these protection bits.

Workbench 2.0 implementation:

Syntax: `LIST DIR,P=PAT/K,KEYS/S,DATES/S,NODATES/S,TO/K,SUB/K, SINCE/K,UPTO/K,QUICK/S,BLOCK/S,NOHEAD/S,FILES/S, DIRS/S,LFORMAT/K,All/S`

The ALL argument was added to this version:

ALL The `ALL` argument displays all the directories and files on a disk. This is very useful for creating printed listings of your disks contents.

2.1.8 RENAME

Syntax: `RENAME FROM/A,TO=AS/A:`

This command assigns new names to a file. The command is useless without arguments. It must have two paths:

1. the complete path description of the object to be renamed
2. the new pathname

This command appears to be very simple. The following changes the filename `My-text` to the name `Essay`, keeping the file in the same directory as before:

```
RENAME text/My-text text/Essay
```

Actually the `RENAME` command is much more flexible. The example above is only a special case where the path stays the same. You can also transfer data or directories within the disk data structure. For this you must make another distinction:

1. The renamed object is a single data object

In this case the destination path out of the directory description must be followed by a new name for the file. The following example places the `Format` command located in the `System` directory into the `C:` directory under the name `Formatting`:

```
RENAME DF0:System/Format DF0:c/Formatting
```

If there wasn't a C: drawer or if the command could not find the FORMAT command, the following error message would appear:

```
Can't rename system/Format as c/Formatting
```

2. *The renamed object is a drawer*

If you only want to change a drawer name, use a simple RENAME. For example:

```
RENAME DF0:Expansion DF0:Expan
```

You can even move a drawer to a different place in the disk data structure.

Example: Suppose you have a directory on disk named BASIC which contains the subdirectory PROGRAMS. In addition, a directory named GAMES which contains a subdirectory named ADVENTURES exists in the main directory. The following command places the entire GAMES directory and its contents in the PROGRAMS directory:

```
RENAME DF0:GAMES to DF0:BASIC/PROGRAMS/GAMES
```

You must specify the new directory's name as well as the source directory's name. The TO argument can be omitted.

Note: You cannot move a file or drawer from one disk drive to another using RENAME. The following input is not permitted:

```
RENAME DF0:C/type RAM:type
```

Only the COPY command can perform this task.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

There are no changes to the RENAME command. It is no longer possible to have two files in the RAM disk with the same name due to a better RAM handler.

Workbench 2.0 implementation:

Syntax: `RENAME FROM/A/M,TO=AS/A,QUIET/S:`

FROM FROM/M allows multiple arguments to be included. Multiple arguments must be separated by spaces. The number of arguments is unlimited in AmigaDOS 2.0.

QUIET The name of the file being renamed is displayed on the screen, unless you specify the QUIET option, which can be useful in script files.

2.1.9 DISKCOPY

Workbench 1.2 implementation:

Syntax: `DISKCOPY FROM/A,TO/A,NAME`

The `DISKCOPY` command is the AmigaDOS equivalent of the `Copy` item in the `Icons` pulldown menu (in 1.3 the `Duplicate` item in the `Workbench` pulldown menu).

Unlike the `COPY` command, this produces a complete copy of the entire disk. The following example copies a disk using only one drive:

```
DISKCOPY FROM DF0: TO DF0:
```

The `TO` argument is required; the `FROM` argument may be omitted.

If you are certain that the data on the destination disk is no longer needed, press the <Return> key to begin the copy operation. You can abort the copying process by pressing <Ctrl><C>:

```
*** BREAK
Disk Copy Abandoned.
Remember to insert original disk

Disk Copy Terminated
```

If you press <Ctrl><C> while the Amiga is writing to the destination disk, not all of the information will be contained on the disk. Remember to put the original disk in the drive after aborting.

In the Workbench a message may appear telling you the number of disk changes you'll have to make during the copy process. It looks like this:

```
The Disk Copy will take 4 swaps. .
```

An Amiga 500 with 512K could copy a disk with just three disk changes. The waiting time between disk changes can be bothersome.

This problem doesn't exist if you own an Amiga with two disk drives.

There are two differences between the Workbench Copy (1.3 `Duplicate`) item and the `DISKCOPY` command. First, the `NAME` argument isn't always needed. This argument lets you assign a different name to the destination disk from that of the source disk. The following example copies the contents of drive **DF0:** into drive **DF1:** then assigns the name Work 1.2 to the new disk (note the use of quotation marks around the name because of the space between Work and 1.2):

```
DISKCOPY DRIVE DF0: TO DRIVE DF1: NAME "Work 1.2"
```

Second, DOS can tell the copy from the original every time from the date and time of the copy operation.

Workbench 1.3 implementation:

Version 1.3 added the `Multi` option to make multiple copies if enough memory is available. `NoVerify` option was added since verify is now automatically on. It also does single disk copies on 1MEG chip ram Amigas. The error messages have been improved.

Workbench 2.0 implementation:

Version 2.0 and Version 1.3 of this command operate identically, the new version has been optimized to operate faster.

2.1.10 RELABEL

Workbench 1.2 implementation:

Syntax: `RELABEL DRIVE/A,NAME/A`

This command assigns a new name to a disk. The following line changes the name of the disk in drive **DF1:** to `Games`:

```
RELABEL DF1: Games
```

There must be a space after the drive specifier. If the filename itself contains a space (e.g., `Test disk`), you must enclose the filename within quotation marks. The following example renames the disk in drive **DF2:** to `Test disk`:

```
RELABEL DF2: "Test disk"
```

The maximum length allowed for disk names is 30 characters. Longer names can pose problems for the Workbench.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Version 2.0 and Version 1.3 of this command operate identically, the new version has been optimized for compactness and the error message has been improved.

2.1.11 INFO

Workbench 1.2 implementation:

Syntax: INFO

The INFO command appears twice in this book: here and in Section 2.2. This version of the INFO command displays disk drive information.

Entering this command without arguments displays information about the currently connected drives. An example of this output follows:

```
Mounted disks:
Unit     Size   Used   Free  Full  Errs  Status      Name
RAM:      30K     30      0   100%   0    Read/Write  Ram Disk
DF0:     880K   1645    113    93%   0    Read        Workbench 2.0
DF1:     880K    534    224    30%   0    Read/Write  TextPro

Volumes Available:
Ram Disk [Mounted]
Workbench 2.0 [Mounted]
TextPro [Mounted]
```

The first section contains information about all the *mounted* (connected) disk drives. The Unit category lists the drive specifier. The Size category lists the disk capacity as specified in the Format command. The Used and Free categories display the number of blocks (1 block=0.5K; 2 blocks=1K) used and the number of blocks still available. The Full category lists the percentage of the disk used. A zero under the Err category means that no defective blocks (errors) exist.

The `Status` category gives the position of the write protect on the disk. The disk in drive 0 can only be read. The last category (`Name`) displays the names of the respective disks.

The second section (`Volumes Available`) lists the names of the disks so that you can check disk names without removing the disks from the drives.

Workbench 1.3 implementation:

Syntax: `INFO DEVICE`

The new `INFO` command includes the `DEVICE` argument. You can receive information about the specified device only. The `Info` command automatically reformats data for easily reading longer names using a `Tab` function.

Workbench 2.0 implementation:

Version 2.0 and Version 1.3 of this command operate identically, the new version has been optimized for compactness and speed.

2.1.12 INSTALL

Syntax: `INSTALL DRIVE/A`

The `INSTALL` command converts Amiga format disks to bootable disks (i.e., an installed disk can be used to boot up when you turn the Amiga on). The Workbench disk is an installed disk.

The following example makes the formatted disk in drive **DF0:** into a bootable disk by placing the *boot block* onto the disk:

```
INSTALL DF0:
```

Note: You cannot make a hard disk drive into a bootable disk. KickStart 1.3 and KickStart 2.0, located in ROM on the Amiga 500 and 2000, lets you boot from a hard disk without using an `INSTALL` command on the hard disk (**never** use the `INSTALL` command on a hard disk).

If you install a newly formatted disk then reset the Amiga immediately, the system resets, stops and enters the `CLI`. There are a number of reasons for this. A bootable disk looks for AmigaDOS commands—it needs these commands to function. The trouble is, it doesn't know where to search for these commands. You have to copy the essential

directories on the Workbench disk onto the new disk. These directories are:

```
C
L
System
Devs
S
Pref
Fonts
Libs
```

In addition, you would have to write a *Startup_sequence* (see Chapter 6 for detailed information about the `Startup-sequence` and script files) to assign system directories within the disk.

The simplest solution to having a bootable disk is to copy the Workbench disk using the `DISKCOPY` command. This copies the boot block and all the necessary directories to the new disk. Then if you need memory for other applications, delete the directories and files not needed by the booting procedure.

Workbench 1.3 implementation:

Syntax: `INSTALL DRIVE/A,NOBOOT/S,CHECK/S`

The added arguments are `NOBOOT` and `CHECK`.

NOBOOT The `NOBOOT` argument makes a bootable disk non-bootable.

CHECK The `CHECK` argument examines the boot block and tells the user whether the boot block has been damaged. This damage may have been done by a *computer virus*. This virus is a program that loads into the computer when the disk is accessed and copies itself onto any disks placed in that drive while the computer is turned on. The virus can cause extensive damage if the disk is used further.

A boot block virus cannot do anything to a non-system disk because it has nothing to do with controlling the computer. The `CHECK` argument displays the following message for non-bootable disks:

```
No bootblock installed
```

When the `CHECK` argument examines a boot disk with an intact boot block, the message reads:

```
Appears to be normal V1.2/V1.3 bootblock
```

The `CHECK` argument displays the following message if the boot block is corrupt or abnormal:

```
May not be standard V1.2/V1.3 bootblock
```

There is a good possibility your computer has been infected by a virus if the disk is one that you formatted. The results of viruses vary from a message on the screen, to a Guru Meditation, to completely formatting the hard disk. There are as many remedies as there are viruses.

We'll briefly describe one method to remove a virus from an infected disk. Turn off the computer for at least five seconds using the main power switch. Boot it with a disk that you know is not infected with a virus. Because most users make a backup copy the first time they use the new Workbench disk, the original disk will almost always work. Start the Amiga with this disk and open a `Shell` window. Enter the following command:

```
DIR >NIL: RAM:
COPY C:INSTALL RAM:
PATH RAM: ADD
```

Put the Workbench disk back in a safe place. Now check out all of your disks for viruses, even if you only have one drive, using the `INSTALL DF0: CHECK` command. The boot block can be installed by using `INSTALL DF0:`, DON'T use this command on any commercial software. When you have done this to all of your disks, you should again have control of the boot blocks. Unfortunately this only takes care of the simple viruses hiding in the boot blocks. Smart viruses infect other parts of the disk (such as `trackdisk.device`). If you think you have a smart virus or any of your commercial software disks are infected contact your local dealer or a user's group as quickly as possible—they may be able to help you.

Workbench 2.0 implementation:

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

The added argument is FFS/S. The AmigaDOS version has also been optimized for compactness and speed.

NOBOOT The FFS/S option is used when you want to use the disk with the FastFileSystem.

2.1.13 TYPE

Workbench 1.2 implementation:

Syntax: `TYPE FROM/A,TO.OPT/K`

The `TYPE` command displays ASCII files on the screen, to a device or to a file. The following command displays the `Startup-sequence` script file in the `S` subdirectory of the Workbench disk on the screen:

```
TYPE DF0:S/Startup-sequence
```

The output can be stopped temporarily by pressing any key. Pressing the <Backspace> key continues the display. Pressing the <Ctrl> and <c> keys aborts the display and returns to the DOS prompt (>1).

Adding `to PRT:` sends the output to the printer. The following example performs the same function as above except it sends the output to a printer:

```
TYPE DF0:S/Startup-sequence TO PRT:
```

The data can also be redirected to other output devices. The following example sends the `Startup-sequence` file to the `T` directory and stores it under the name `mytext`:

```
TYPE :S/Startup-sequence T/mytext
```

Adding the `OPT N` argument displays text with line numbers. This is useful for viewing a BASIC program stored in ASCII format.

The `OPT H` argument displays each word of the file being typed as a hexadecimal number. `OPT H` is intended mainly for the true hacker. The `TYPE` command is perfect for text output when the data doesn't contain any control characters. If you try to `TYPE` a DOS command (e.g., `TYPE C/TYPE`) you'll get garbage on the screen. However, the `TYPE C/TYPE OPT H` command organizes the screen into a table like this:

```
0000: 000003F3 00000000 00000002 00000000    ................
0010: 00000001 0000004F 000001C4 000003E9    .......o........
0020: 0000004F 286A0164 700C4E95 2401223C    ...o(j.dp.N.$."<
0030: 00000095 49FAFFEE 286CFFFC 2F0C2F02    ....I...(l.././.
```

On the far right we have our text displayed in ASCII. Each period stands for a non-displayable character that AmigaDOS handles by displaying a period.

The first column lists the hexadecimal line numbers. The middle column displays the contents of the file using four long words. Each long word is made up of four bytes, and each byte represents one character, so each byte corresponds to a character on the right margin.

The I in the last line stands at the 52nd byte position (=3*16 +4). The ASCII code that is associated with the text for an I reads: $49 ($=hexadecimal) or 73 decimal (4*16 +9).

Workbench 1.3 implementation:

Syntax: `TYPE FROM/A,TO,OPT/K,HEX/S,NUMBER/S`

The options `OPT H` and `OPT N` arguments can also be accessed using the `HEX` and `NUMBER` arguments without the `opt` argument. For example:

Version 1.2:
```
TYPE S:Startup-sequence OPT N
```

Version 1.3:
```
TYPE S:Startup-sequence NUMBER
```

Workbench 2.0 implementation:

Version 2.0 and Version 1.3 of this command operate identically, the new version has been optimized for compactness and speed.

2.1.14 JOIN

Syntax: `JOIN ,,,,,,,,,,,,,,,AS/A/K`

The `JOIN` command lets you *concatenate* (join) up to fifteen files to create one new file.

The fifteen commas in the command template represent the maximum fifteen source files. The `AS` argument must follow. Then follows the path description for the concatenated file (`/A`). The simplest form of the `JOIN` command can simulate the basic function of the `Type` command. By placing an asterisk behind the command you specify the source data and `JOIN` displays it on the screen. The following demonstration displays the text of the startup sequence on the screen:

```
JOIN DF0:S/Startup-sequence AS *
```

There is no argument available to let us print multiple files at one time. The `COPY` command accepts the wildcard, but that really doesn't allow more data to be accessed. The `JOIN` command makes it possible

to print out 15 data files right after each other. The following prints text files `text1` through `text5`:

```
JOIN text1 text2 text3 text4 as prt:
```

The `JOIN` command also has something to offer the compiled language programmer. If you run out of room using your editor, this command allows you to concatenate separate files into one file before compiling.

Workbench 1.3 implementation:

Syntax: `JOIN ,,,,,,,,,,,,,,,AS=TO/K`

The `JOIN` command now understands the `TO` argument as well as the `AS` argument.

Workbench 2.0 implementation:

Syntax: `JOIN FILE/M,AS=TO/K/A`

FILE/M allows multiple arguments to be included. Multiple arguments must be separated by spaces. The number of arguments is unlimited in AmigaDOS 2.0. The new version has been optimized for compactness and speed.

2.1.15 SEARCH

Workbench 1.2 implementation:

Syntax: `SEARCH FROM,SEARCH/A,ALL/S`

The `SEARCH` command lets you look for data using a character string. If AmigaDOS finds the character string it displays the name of the file in which the string is located, followed by the line number and the line that contains the string.

FROM The `FROM` argument represents the complete path specification of a directory and a single data item. If the `FROM` argument is omitted, the command looks in the current directory.

SEARCH The `SEARCH` argument must precede the search string:

```
SEARCH SEARCH "Goodness gracious"
```

The SEARCH command searches the current directory for the words "Goodness gracious". Quotation marks must surround any string containing a space. SEARCH makes no distinction between uppercase letters and lowercase letters. If you want to search all subdirectories you can direct the SEARCH command to do so.

Wildcards

Like the LIST command, this command allows you to complete the pathname using wildcards. The following command searches for all files in a subdirectory starting with three letters:

```
SEARCH DF0:C?/#? SEARCH window
```

This would find all the files starting with C in any directories containing one letter names, containing the word "window."

Like all of the AmigaDOS commands, the SEARCH command can be stopped by pressing <Ctrl><C>. When searching all the directories, pressing <Ctrl><D> moves AmigaDOS to the next file.

When AmigaDOS returns the message `Line x truncated`, the lines in the file being searched are too long (this happens often).

The SEARCH command is very helpful to the C programmer. The command can quickly find the desired include directories.

Workbench 1.3 implementation:

Syntax:

```
SEARCH FROM,SEARCH/A,ALL/S,NONUM/S,QUIET/S,QUICK/S,FILE/S
```

The 1.3 SEARCH command replaces the message `Line xx truncated` with `Warning: line xx too long`. In case the search operation comes up empty (null), AmigaDOS returns error code 5. The error code can be analyzed in a script file (see Chapter 6).

There are four new arguments:

NONUM The NONUM argument suppresses line number output when the search finds multiple items. The text found appears at the left margin of the screen for easy readability.

QUIET The QUIET argument searches files without output.

QUICK The QUICK argument displays the filenames being searched next to one another instead of under one another. A new directory begins a new line.

FILE The FILE searches for a specific filename instead of a string.

Workbench 2.0 implementation:

Syntax: `SEARCH FROM/M/A,SEARCH/A,ALL/S,NONUM/S,QUIET/S, QUICK/S,FILE/S,PATTERN/S`

The new version has been optimized for compactness and speed. Two new arguments have been added, FILE/M and PATTERN/S

FILE FILE/M allows multiple arguments to be included. Multiple arguments must be separated by spaces. The number of arguments is unlimited in AmigaDOS 2.0.

PATTERN PATTERN/S allows pattern matching to be used in searches.

2.1.16 SORT

Workbench 1.2 implementation:

Syntax: `SORT FROM/A,TO/A,COLSTART/K`

The `SORT` command sorts (alphabetizes) text files.

The arguments are as follows:

FROM The `FROM` argument specifies the pathname of the file to be sorted. Because this cannot be a directory, an additional input is necessary (`/A`).

TO The `TO` argument specifies the destination of the sorted data. Here a pathname or device name must be given. The `FROM` data isn't really changed. If you want output on the screen, for example, you must enter the * character. Using the `PRT:` device directs the sorted output to the printer.

COLSTART The `COLSTART` argument specifies the column at which the sorted output should start. For example, if you reserve 10 places for first names and a certain number of places for last names the following sorts the last names starting at the tenth column:

```
SORT FROM fred TO ned COLSTART 11
```

If you omit the `COLSTART` argument the sorting begins at the first column.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Syntax: `SORT FROM/A,TO/A,COLSTART/K,CASE/S,NUMERIC/S:`

The new version has been optimized for compactness and speed. Two new arguments have been added, `CASE/S` and `NUMERIC/S`.

CASE When `CASE/S` is specified the sort is case sensitive.

NUMERIC NUMERIC/S allows numeric sorts.

2.1.17 PROTECT

Workbench 1.2 implementation:

Syntax: `PROTECT FILE/A,FLAGS:`

The `PROTECT` command lets you set a single protection bit (see Section 2.1.8 for a detailed description of the four protection bits).

```
r     Read-the file can be read
w     Write-the file can be written to
e     Execute-an 'execute' is allowed
d     Delete-an entry can be deleted
```

The delete bit can be activated from DOS. This bit acts like the write protect on disks, except the delete bit guards an individual file from deletion instead of the entire disk. The following command sets the delete bit on the `Letters` directory in drive **DF1:**:

```
PROTECT DF1:Letters
```

Files inside directories can be protected by activating their own delete bits. The following example sets the delete bit in the `Invitations` file contained in the `Letters` directory:

```
PROTECT DF1:Letters/Invitations rwe
```

If you view a protected file using the `LIST` command, the protection bits appear as four hyphens. These hyphens indicate that the file can no longer be accessed. Any attempt to erase the file returns an error code. The protection can be removed using the `FLAGS` argument. The

following command enables all four protection bits in the Invitations file:

```
PROTECT DF1:Letters/Invitations rwed
```

Workbench 1.3 implementation:

Syntax: `PROTECT FILE/A,FLAGS,ADD/S,SUB/S`

Workbench 1.3 adds four new protection bits to the `PROTECT` command:

```
h     (Hidden)—controls visibility of certain file entries
s     (Script)—controls starting script files w/o Execute
p     (Pure)—controls program loading using Resident
a     (Archived)—controls file copying (Kickstart 1.3)
```

When using Workbench 1.3/Kickstart 1.2 to start your Amiga you must pay particular attention to the `p` and `s` flags.

`h(idden)` The `hidden` protection bit suppresses the entry of the respective files in the directory. For example, the `.info` files responsible for the icon on the Workbench disk can be made invisible in the directory list. Larger directories can be made more readable using this method.

`s(cript)` The `script` protection bit deals with script files. When the script flag is positive (set), the script file can be started from a shell. It is not necessary to enter the `Execute` command to invoke a script file anymore. A set `script` flag automatically calls an `Execute` command.

`p(ure)` The `pure` protection bit allows the associated program to be loaded using the `Resident` command. By doing this it is always ready for the user and it also doesn't have to be loaded from the drive anymore.

The `pure` protection bit is necessary because not every program has the qualities needed for using the `Resident` command. More information about the `Resident` command can be found in Chapter 4.

`a(rchive)` The `archive` protection bit controls the option of copying files under Kickstart 1.3. The `Copy` command only copies files that have negative (unset) `archive` protection bits. A file with a *positive* (set) `archive` protection bit is said to be *archived*. The `archive` protection bit changes to negative when you write to the file. A new `archive` protection bit must be set.

One practical application: When you work with the RAM disk, you can activate a script file as a background process that can save all modified data on a disk. When you place the commands `Copy`, `Wait` and `Execute` in working memory, the disk drive eventually performs a save operation. The following file acts as a script file to do just this:

```
wait 5 min
copy ram:#? to df0:
execute BACKUP_SCRIPT
```

This script file also functions under Kickstart 1.2. The complete contents are saved whether the RAM disk has been written to in the last five minutes or not.

The ADD and SUB arguments make individual protection bits positive or negative. These are the equivalents of adding + and - to change protection bit status. The following examples show how ADD and SUB work:

```
SUB
Status before       ----rwed
Input        protect file d sub
Status after        ----rwe-

ADD
Status before       ----rwe-
Input        protect file d add
Status after        ----rwed
```

The ADD and SUB options can be replaced by plus and minus signs. The input is simplified this way:

```
-
Status before       ----rwed
Input        protect file -w
Status after        ----r-ed

+
Status before       ----r-ed
Input        protect file +w
Status after        ----rwed
```

Workbench 2.0 implementation:

Syntax: `PROTECT FILE,FLAGS,ADD/S,SUB/S,ALL/S,QUIET/S`

The new version has been optimized for compactness and speed. Two new arguments have been added, ALL/S and QUIET/S.

ALL When All/S is used all protection bits can be cleared.

QUIET The QUIET argument does not display the filename of the files being accessed. This can be useful in script files.

2.1.18 FILENOTE

Syntax: FILENOTE FILE/A,COMMENT/A

The FILENOTE command allows you to place up to 79 characters of comments in a file or place a comment about the version number in a program. You can read the comments later using the LIST command. The text appears on a separate line. A colon at the beginning of the line indicates that it is a comment. For example:

```
FILENOTE C/FILENOTE "This command lets you add 80 characters to files!"
```

The quotation marks must surround any text containing spaces. If the LIST command is used on the C/FILENOTE file, this is the result:

```
c/filenote                    700 rwed 20-Jun-90 23:30:19
: This command lets you add 80 characters to files!
```

Two final observations about the FILENOTE command: Comments inserted using FILENOTE don't copy using the COPY command. In addition, if the destination file already exists, the comments in the destination file remain intact.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Syntax: FILENOTE FILE/A,COMMENT/A,All/S,QUITE/S

The new version has been optimized for compactness and speed. Two new arguments have been added, ALL/S and QUIET/S.

ALL When All/S is used all comments can be cleared.

QUIET The QUIET argument does not display the filename of the files being accessed. This can be useful in script files.

2.1.19 SETDATE

Syntax: `SETDATE FILE/A,DATE,TIME`

This command makes it possible to store the correct date entry of files. This is useful for Amiga users who have battery-powered real-time clocks in their Amigas; the time is set without using the `Preferences` editor.

FILE The `FILE` argument represents the path description of the directory/file. The specified file must be found and in the same format as it appears in the `LIST` command.

DATE The `DATE` argument represents the current date. If the old date is only within a week of the current date, then you can enter the current day's name for the `DATE` argument, as shown in the following example:

```
SETDATE text/Letter Saturday
```

The command sets the date correctly by itself. The correct date can even be set by using the word `yesterday` as the `DATE` argument. These words appear in the listing executed by the `LIST` command. If you want to pre-date something (assign a future date), the `LIST` command shows the word `future` for any future datings.

TIME The `TIME` argument sets the current time. When the time setting is correct, then the entire date description appears. If you do not set the time, the time automatically sets to 00:00.

Note: Date settings before January 2, 1978 are usually not shown. When this occurs, two empty spaces appear in the `LIST` display.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Syntax: `SETDATE FILE/A,DATE,TIME,All/S:`

The new version has been optimized for compactness and speed. A new argument has been added, `ALL/S`.

ALL When `ALL/S` is used all dates can be set to TODAY.

2.1.20 DISKDOCTOR

Workbench 1.2 implementation:

Syntax: `DISKDOCTOR DRIVE/A`

The `DISKDOCTOR` attempts to save data on disks that have read/write errors or possible data corruption in general.

The `DRIVE` argument represents the disk drive specifier (e.g., **DF0:**, **DF1:**, etc.). The following example invokes the `DISKDOCTOR` and examines the disk in **DF1:** (the first external disk drive on an Amiga 1000 or 500):

```
DISKDOCTOR DF1:
```

Error messages The following messages that are displayed by `DISKDOCTOR` during execution are documented in the following section.

`DiskDoctor cannot run in the background`

This is displayed when you try starting `DISKDOCTOR` as a background process using the `Run` command. `DISKDOCTOR` can only be executed directly (without `Run`).

`Unknown device xxx`

This occurs when the description of a device name that DOS doesn't know is entered (`xxx` stands for the device name).

`Not enough memory`

`DISKDOCTOR` needs more memory than the system can allocate. Hint: Close all unnecessary windows and/or end all other running programs. This message also appears when you try to use `DISKDOCTOR` on a device other than a disk drive (printer, serial device, etc.).

`Device xxx not found`

`DiskDoctor` cannot find the desired device. This error almost never occurs with normal 3.5" drives because of the `trackdisk.device` found in ROM (in WOM for the Amiga 1000). This error is usually a result of a device name entry error in the Mount list for unusual drives (e.g., 5.25"). By using a special disk drive the error message appears when the device is not found in the Mount list.

```
Unable to open disk device
The disk device was found, but it cannot be opened
Unexpected end of file
```

DOS handles the file with a great amount of redundancy. The advantage of this redundancy is that it's easier to reconstruct this file if the file somehow becomes damaged. This error message occurs when the file is shorter than is declared in the file header.

```
Error: Unable to access disk
```

This occurs when the drive is unable to respond (e.g., no disk in the drive).

```
Disk must be write enabled
```

Write protects prevent writing to the disk. Because `DISKDOCTOR` wants to write to the disk, write protects must be set to write enable (no hole in the write protect area).

```
Unable to read disk type - formatting track zero
```

`DISKDOCTOR` cannot read the disk type from track zero, sector zero. It reformats that track and sector.

```
Track zero failed to format - Sorry!
```

There is a good chance of a defect on track zero of the disk when this message appears. There may be a problem with the drive itself (read/write head is incorrectly positioned) if this happens frequently with other disks.

```
Unable to write to root - formatting root track
```

`DISKDOCTOR` cannot rewrite the track on which the root block appears. This root block acts as the reference point of all the disk directories. `DISKDOCTOR` tries to format the track (track 40, side 0) and install the disk. Because the name of the disk is found on this track, `DiskDoctor` assigns the name `Lazarus` to the disk.

```
Root track failed to format - Sorry!
```

The root track cannot be formatted. The disk cannot be rescued.

```
Cannot write root block - Sorry!
```

The root block cannot be written. `DISKDOCTOR` can't do anything about it.

`Warning: File xxx contains unreadable data`

The specified file (`xxx`) cannot be reconstructed fully and doesn't contain any readable data. You may be able to salvage some of this data using a disk monitor. In most cases, the file must be erased by answering `Yes` to the "`Delete corrupt files in directory yyy?`" prompt.

`Attention: Some file in directory xx is unreadable and has been deleted`

`DISKDOCTOR` has taken the initiative and erased a file because too much information was missing for reconstruction.

```
Failed to read key
A block cannot be read
Failed to rewrite key
A block cannot be rewritten
Warning: Loop detected at file xx
```

Normally, a file stands at a single block together with a block pointer that connects it to the rest. This error message means that the given file has a loop in the connection. A file block loops back to a block that has already been read. The read operation of the file may never have ended because the same data was being read all the time.

`Parent of key xx is yy which is invalid`

A block exists which is not connected to the list because the operating block is useless.

`Hard error Track xx`

Track number `xx` cannot be read either because it was incorrectly formatted or because of mechanical failure. The problem may be caused by the reconstruction of some files or directories.

`Key xx now unreadable`

The block with the number `xx` is no longer readable.

`Replacing dir xx`

The given directory can be reconstructed and is now being integrated into the directory structure of the disk.

`Inserting dir xx`

The given directory can be reconstructed and is now being entered in the main directory of the disk.

`Replacing file xx`

The given file can be reconstructed and is now being entered into the original directory.

`Inserting file xx`

The given file can be reconstructed and is now being entered into the main directory of the disk.

`Now copy files to a new disk and reformat this disk`

This is the closing message of DISKDOCTOR. All rescued files and directories can now be copied to a new disk. Then the defective disk should be reformatted.

Workbench 1.3 implementation:

DISKDOCTOR can also be used for reconstructing the recoverable RAM disk.

The use of the Version 1.2 and 1.3 DISKDOCTOR are identical. The 1.3 program has been enhanced and updated.

Workbench 1.3.2 implementation:

Diskdoctor V1.3.4 corrected the out of memory error message and now uses BufMemType so it will work with large hard drives.

Workbench 2.0 implementation:

Syntax: DISKDOCTOR DRIVE/A

The new version has been optimized for compactness and speed. You can use DISKDOCTOR on standard and FastFileSystem diskettes. The DosType keyword in the MountList must be set to 0x444f5301 to use DISKDOCTOR on FastFileSystem diskette! Never use DISKDOCTOR on a FastFileSystem diskette if the DosType keyword is not set correctly.

2.1.21 DISKCHANGE

Workbench 1.2 implementation:

Syntax: DISKCHANGE DEV/A

This command deals only with material for Amiga owners who use 5 1/4" disk drives or removable media drives. These drives, unlike the 3.5" drives, don't come with DOS already on them. In this case, the DISKCHANGE command is given, followed by the name of the given device (the DEV argument). After that the new disk can be selected.

This command can also be used to inform WorkBench of a name change to a floppy diskette using the RELABEL command.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Syntax: DISKCHANGE DRIVE/A

Version 2.0 and Version 1.3 of this command are identical. The new version has been optimized for compactness and speed.

2.2 System Commands

The following section describes system commands, including the commands that are related to the AmigaDOS `Shell` itself.

2.2.1 NEWCLI

Workbench 1.2 implementation:

Syntax: `NEWCLI WINDOW, FROM`

Multitasking The `NEWCLI` command gives the Amiga user access to multitasking. CLI stands for Command Line Interface. Multitasking allows different programs to run at almost the same time. For example, you can print a letter while formatting a new disk.

The command runs alternating tasks, not parallel tasks (that's why the word *almost*). This is something like the digital readout on a clock radio. The numerals on a clock light up one after another, not all at once. The rapid rate at which they change fools the eye into thinking the numbers are lit simultaneously.

The `NEWCLI` command makes it possible to add a running task. After entering the command, another AmigaDOS window appears. In 1.2 it is named after the current task (e.g., `NEWCLI task 2`), in 2.0 it is named AmigaShell. The Amiga can have more than one `CLI` window open at a time.

However, work can only be done in one window at a time. You can, for example, enter `FORMAT DRIVE DF0: NAME Empty` in the original `CLI` window, then click on the new window and enter `DIR DF1:` to see the contents of the disk in the external drive.

There is a disadvantage to multitasking: Each additional task increases the risk of errors. See the chapter on AmigaDOS and Multitasking for more information about multitasking.

Finally, we'll describe the parameters allowed in the `NEWCLI` command. First, `NEWCLI` can open a window in the size and title specified by the user. The following command creates a window named `Amiga` with a width of 250 pixels and height of 100 pixels, with the

upper left corner of the window starting at X-coordinate 50 and Y-coordinate 70:

```
NEW CON:50/70/250/100/Amiga
```

This option works best when using the command in conjunction with a `Startup-sequence` script file.

If the size input is missing, the `CLI` creates a window the full width and half the height of the normal screen or one quarter of an interlaced screen.

FROM

With the addition of the `FROM` argument and the name of a script file, the `NEWCLI` command can automatically call a new `CLI` and execute a script file. If the script file is in a drawer the complete pathname must be specified. An example:

```
NEWCLI FROM S/Copies
```

In this example the script file named `Copies` in the S: directory executes, before you can work with the new `CLI`.

Workbench 1.3 implementation:

Every time the `NEWCLI` command is called it executes a script file named `CLI-Startup`, which is in `S:` directory on the Workbench disk. The only command contained in this file is the `PROMPT` command, which creates the DOS prompt for the new `CLI`.

NEWSHELL

The `NEWCLI` command has become obsolete with AmigaDOS 1.3. In the `C:` directory on the new Workbench disk there is a new command called `NEWSHELL`. This command creates a window port to AmigaDOS that has many advantages over the `CLI`.

Many of these additions can only be used when the shell segment is resident in Amiga RAM before calling the `NEWSHELL` command. The command reads:

```
RESIDENT CLI 1:SHELL-SEG SYSTEM pure
```

This command is automatically executed when the computer is first turned on so that you don't have to bother with it. The `Shell` window has the following advantages:

Resident commands supported

AmigaDOS now contains a `RESIDENT` command that can load most of the commands into working memory so they do not have to be loaded from disk. These commands are then ready for use by the user. It is covered in detail in the More AmigaDOS Commands chapter. Calling resident commands is only possible through the `Shell`. In a typical AmigaDOS window such a command is loaded from the disk.

Command synonyms allowed

It's often a good idea to give your AmigaDOS commands shorter names using Rename. There are disadvantages to this. Rename the Fault command, which is found in the C directory, to FT (RENAME C:FAULT AS C:FT). The FAULT command can be used to view the text of an error message. For example, if FT 103 is entered, Fault 103: insufficient free store is displayed (in 2.0 Fault 103: not enough memory available is displayed).

Try to erase your AmigaDOS command directory using DELETE C. This can't be done because the directory is not empty. The error message Not Deleted—directory not empty is displayed, which is FAULT 216. AmigaDOS also makes use of the AmigaDOS commands.

NEWSHELL allows you to call any command by another name. The syntax for this reads:

```
ALIAS Newname originalname
```

Newname stands for any character string without spaces that can be used to call that command. originalname is the name of the command that should be executed by using the new name. When AmigaDOS finds a name at the beginning of a line for which such a relationship exists, this name is replaced by the related command. All other input remains unchanged. For example:

```
ALIAS D DIR
```

The DIR command can be called by entering a D followed by a <Return>. The relationship between the shortened version and the normal command is not stored on disk but in a table that is controlled from AmigaDOS.

The description of the original command is not limited to a single word. You can build your own command using ALIAS if you use the same options with a command all of the time:

```
ALIAS S-UP RUN ED S:Startup-sequence
```

Now you can load the Startup-sequence into ED for editing by entering S-UP.

Unfortunately the relationships are lost when the computer is turned off. For this reason a script file can be created so that any number of ALIAS relationships can automatically be established. This file is found in directory S: of the Workbench disk and is called Shell-Startup. All entered relationships are valid in each new AmigaDOS window.

A list of the current relationships can be obtained by entering just the word ALIAS.

Output of current path

In the AmigaDOS Shell, the prompt represents the actual directory path. This indicates at which branch of the directory tree you stand. The current path can be read by entering CD. Making your own prompt is discussed under the description of the PROMPT command.

Direct calling of script files

Usually only object programs can be started directly from AmigaDOS. For example, if you try to start a script file by entering its name, the error message Unable to load xxx: file is not an object module (xxx stands for the filename) appears. Script files can be started using the EXECUTE command.

Script flags allow access to a script file without the EXECUTE command. DOS recognizes the flag, knows it's dealing with a script file and automatically calls EXECUTE. The command for setting the flags reads: PROTECT Filename +s (see PROTECT).

When script files are started in this manner a script file with the name Shell-Startup is called by NEWSHELL. This file is found in the S: directory of the Workbench disk.

Workbench 2.0 implementation:

The NEWSHELL windows of AmigaDOS 2.0 now contains a close gadget. The NEWCLI command is now the same as the NEWSHELL command; a Shell window will be opened, not a simple CLI window. The command was also optimized for compactness and speed. It was also made an internal command.

Further information on the NEWSHELL command can be found in the More AmigaDOS Commands chapter.

2.2.2 ENDCLI

Workbench 1.2 implementation:

Syntax: ENDCLI

This command closes the current CLI window task started from the Workbench or with NEWCLI. A second CLI cannot be closed from the first CLI window. If a CLI which was started by using RUN ends, the CLI ends before the process is ended; the window remains open for output from the currently running task. When the last task ends; the window closes.

Note: If the Workbench is not already loaded and you enter ENDCLI, the Workbench screen appears without icons or a menu bar (you won't have access to the Workbench). Enter the LOADWB command, then enter ENDCLI to exit to the Workbench.

Workbench 1.3 implementation:

There is no such command as ENDSHELL; Shell can be ended using ENDCLI. Some versions of the Shell-Startup script file contain the statement ALIAS ENDCLI so that Shell will accept the ENDSHELL command.

Workbench 2.0 implementation:

ENDCLI is synonymous with ENDSHELL; this was done to keep compatibility with earlier versions of AmigaDOS.

2.2.3 RUN

Workbench 1.2 implementation:

Syntax: RUN PROGRAM_NAME

This command executes a program or AmigaDOS command while allowing access to a program running in the background *and* the current AmigaDOS task. Any output from the RUN command appears in the AmigaDOS window which started the task. The example below prints three files named letter1, letter2 and letter3 and then displays the RAM disk directory:

```
RUN C/JOIN Letter1 Letter2 Letter3 TO PRT:
DIR RAM:
```

The JOIN command sends the multiple letters to the printer. The RUN command starts the first task and immediately frees up the computer to display the RAM disk contents.

There is an alternative to using JOIN to print the three letters. AmigaDOS accepts the plus sign (+) character followed by the <Return> key as a specifier for multiple commands. The following example performs the same task as the example listed above:

```
RUN TYPE Letter1 to PRT: +
TYPR Letter2 to PRT: +
TYPE Letter3 to PRT:
```

The entire command group executes as a background process as soon as you press the <Return> key following the last line (the line without "+").

Workbench 1.3 implementation:

It should be possible to leave the AmigaDOS `Shell` used to start a task by using `ENDCLI`, but also without closing the window eventually used for output.

The following command creates a background process that writes the entire contents of the disk in drive **DF0:** to a file named `List`:

```
RUN >List DIR DF0: OPT A
```

It should be theoretically possible to leave the `Shell` using `ENDCLI` and close the window while the `DIR` command continues to work. It doesn't work that way; the device receives an EOF (end of file) command character. The number of the task (e.g., `CLI [2]`) is given to the device.

To allow the closing of the `Shell` while the running process continues, redirect the output to NIL: using the > redirection symbol.

Our sample file displays the task number instead of the disk directory, if the command `TYPE List` is used.

Workbench 2.0 implementation:

The RUN command was optimized for compactness and speed. It was also made an internal AmigaDOS command.

2.2.4 STATUS

Workbench 1.2 implementation:

Syntax: `STATUS PROCESS,FULL/S,TCB/S,CLI=ALL/S`

This command displays all the information available about the AmigaDOS tasks running at that particular time. If you enter `STATUS` without parameters, or if you enter `STATUS all`, AmigaDOS displays the names of the individual tasks. The following example is a response to `STATUS ALL`:

```
Task 1: Loaded as command: status
Task 2: Loaded as command: beckertext
```

In this case, because the BeckerText program was started from AmigaDOS using RUN, it is assigned task number two.

PROCESS The PROCESS argument specifies the correct task number for additional information about the task. Entering Status 2 would only show the second line of the above output.

TCB The TCB argument produces more information about the individual Tasks Control Block. Entering Status TCB for the above data would return the following:

```
Task 1: stk 1600, gv 150, pri 0
Task 2: stk 3200, gv 150, pri 0
```

The information following the task number has the following meaning:

stk Processor stack size of this task

gv Global vector table width

pri Specified task's priority (values range from -128 to +127)

FULL The FULL argument displays complete information about tasks. Status full displays the following for the above example:

```
Task 1: stk 1600, gv 150, pri 0 Loaded as command: status
Task 2: stk 3200, gv 150, pri 0 Loaded as command: textpro
```

Workbench 1.3 implementation:

Syntax: PROCESS,FULL/S,TCB/S,CLI=ALL/S,COM=COMMAND/K

The 1.3 STATUS command gives negative priorities correctly. In addition, the new STATUS includes the COM=COMMAND/K argument. This argument helps the user determine if a specific program exists in the current task. The user must enter STATUS COM and the name of the task. The following example searches for a task named TextPro and displays the corresponding process number:

```
STATUS COM TEXTPRO
```

No other output occurs. If the process doesn't exist, the condition flag is set to 5 (=WARN). If the task is found, the shell number is output and the condition flag is set to 0. This argument is especially helpful in script files for seeing if a background task is running.

Workbench 2.0 implementation:

Syntax: `PROCESS/N,FULL/S,TCB/S,CLI=ALL/S,COM=COMMAND/K`

Operation is identical to Workbench 1.3, but the `PROCESS` argument is specified as numeric. The command has also been optimized for compactness and speed.

2.2.5 CHANGETASKPRI

Syntax: `CHANGETASKPRI PRI/A`

This command changes the current `CLI` task's priority. Each task in the Amiga has a given priority. This value can range from -128 to +127. The following example sets the priority of the current task to 5:

```
CHANGETASKPRI 5
```

Entering `STATUS FULL` after the above `CHANGETASKPRI` command will display a message similar to the following, depending on which tasks are running your computer:

```
Task 1: stk 1600, gv 150, pri 5 Loaded as command: status
Task 2: stk 3200, gv 150, pri 0 Loaded as command: textpro
```

If the input is out of the allowed range, the following appears:

```
Priority out of range (-128 to +127)
```

Workbench 1.3 implementation:

Syntax: `CHANGETASKPRI PRI/A,PROCESS/K`

The `PROCESS/K` argument allows the user to change the priority of any process. You must enter the process number following the `PROCESS` argument. The following example changes process number 4 to a priority of -5:

```
changetaskpri Pri -5 Process 4
```

This option is very useful in case you have started a printing operation as a background process and want to slow down this task so your other tasks are done more quickly. `CHANGETASKPRI` lets you lower the priority of the printing task, freeing up time for other tasks to execute.

Workbench 2.0 implementation:

Syntax: `CHANGETASKPRI PRI=PRIORITY/A/N,PROCESS/K/N`

Operation is identical to Workbench 1.3, but the PRIORITY and PROCESS arguments are specified as numeric. The command has also been optimized for compactness and speed.

2.2.6 BREAK

Workbench 1.2 implementation:

Syntax: `BREAK PROCESS/A,ALL/S,C/S,D/S,E/S,F/S`

This command halts execution of a DOS command from any AmigaDOS window. For example, if the first task window contains the `DIR OPT A` command, the complete output of this command can be stopped by entering `BREAK 1` from a second window.

You can achieve the same result by activating the first window and pressing the <Ctrl> and <C> keys. But there is another use for this command. In the description of the `RUN` command we mentioned a way to print out more than one letter when it's started. What would you do if you wanted to stop the printing process? Turning the printer off is not the correct way. Pressing <Ctrl><C> in the window from which the process was started doesn't work because the process is running without a window. However, the `BREAK` command will stop output to the printer.

PROCESS The `PROCESS` argument tells the system which task to interrupt.

C,D,E,F The `BREAK` command without arguments defaults to <Ctrl><C>. The `C, D, E, F` arguments allow you to change the control character to <Ctrl><C>, <Ctrl><D>, <Ctrl><E> or <Ctrl><F>. The following example transmits a <Ctrl><D> to task number 3:

```
BREAK 3 d
```

A multiple file operation will stop at the beginning of the next file when `BREAK` is sent. The operating system's response to <Ctrl><C> varies from case to case and depends on the respective AmigaDOS command. In most cases, nothing happens.

ALL The ALL argument sets all four <Ctrl> codes simultaneously. The following example sends all the <Ctrl> codes to task 3:

```
BREAK 3 ALL
```

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed.

2.2.7 PATH

Workbench 1.2 implementation:

Syntax: `PATH ,,,,,,,,,ADD/S,SHOW/S,RESET/S`

This command displays the current directory and disk path. If the PATH command is entered without parameters or is followed by SHOW, a disk path appears on the screen. Here's an example of the output of the path command:

```
Current directory
RAM:c
A500 WB 1.2 D:System
C:
```

This list shows the order and directories used when searching for a file. If the name of a program is entered (e.g., an AmigaDOS command), DOS first searches the current directory for the file. The current directory can be specified using the CD command.

If DOS doesn't find the file in the current directory, it searches RAM:C and the System drawer on the Workbench disk. If the file is not in any of these places, DOS finally looks in a *virtual* (i.e., it exists only within the computer) device named C:. This pseudo device ensures that the AmigaDOS command searches for the correct directory. See the description of the ASSIGN command for more information about virtual device C:.

The PATH command allows the user to add or remove paths. For example, if you use the calculator in the Utilities drawer of the

Workbench disk often, the following command to load the program is entered:

```
Utilities/Calculator.
```

However, if you enter the command `Path SYS:Utilities ADD` beforehand, the path list will have the following added:

```
A500 WB 1.2 D:Utilities
```

Now AmigaDOS automatically looks in the `Utilities` drawer.

The `PATH` command is especially useful when used in conjunction with the RAM disk and Workbench 1.2. Because additional paths in the list are always searched before the `C:` device, several DOS commands can be placed in the RAM disk. This saves the floppy disk user quite a bit of work because the operating system looks in the RAM disk first for the desired command. Next it calls for the Workbench disk to be inserted because the command was not found (see Chapter 3 for more information on this subject). The Workbench 1.3 and 2.0 RESIDENT command is a much better solution.

ADD The `ADD` argument must appear at the end of the list to add up to ten new path specifications.

RESET The `RESET` argument removes all of the paths up to a maximum of 10 paths. All paths except the current directory and the `C:` are deleted.

Workbench 1.3 implementation:

The `PATH` command's function remains unchanged but the search order is different in the 1.3 version. When you omit a specific path for a command, AmigaDOS first searches the resident commands. If it cannot find the command in residence, the search operation continues as described above.

Workbench 2.0 implementation:

Syntax: `PATH PATH/M,ADD/S,SHOW/S,RESET/S,REMOVE/S`

PATH AmigaDOS 2.0 allows multiple arguments to be passed to the PATH command. Multiple arguments must be separated by spaces; this replaces the multiple comma method used in 1.3.

REMOVE The `REMOVE/S` argument allows you to remove a segment of the path without resetting the entire path.

The `PATH` command has improved its compactness and speed; it has also been made an internal AmigaDOS command.

2.2.8 ASSIGN

Workbench 1.2 implementation:

Syntax: `ASSIGN NAME,DIR,LIST/S`

Before we describe this command in detail, look at the Amiga's response when you enter `ASSIGN LIST`:

```
Volumes:
RAM disk [Mounted]
A500 WB 1.2 D [Mounted]

Directories:
S               A500 WB 1.2 D:s
L               A500 WB 1.2 D:l
C               A500 WB 1.2 D:c
FONTS           A500 WB 1.2 D:fonts
DEVS            A500 WB 1.2 D:devs
LIBS            A500 WB 1.2 D:libs
SYS             A500 WB 1.2 D:

Devices:
DF0 DF1 PRT PAR SER
RAW CON RAM
```

`Volumes` lists the names of the disks currently recognized by AmigaDOS. The word `[Mounted]` means that the disk is currently in the drive (this doesn't literally apply to the RAM disk).

Look at the entries beneath the `Directories` category. The left margin lists the known devices. You read some information about the `C:` virtual device under the description of the `PATH` command. Each virtual device is a real path related to a currently existing directory. A device name can also be labeled for the path on the right. The `C:` device is related to the `C:` drawer on the Workbench disk. The `C:` drawer contains all the AmigaDOS commands. The device name doesn't have to be the same name as the drawer name, in some cases. A program in the device named `Fonts:` can be assigned a drawer named `Character_sets:`. Naturally, the `ASSIGN` command allows these assignments to be changed.

```
ASSIGN NAME,DIR,LIST/S
```

NAME The `NAME` argument represents a device name (AmigaDOS recognizes this from the ending colon).

DIR The DIR argument represents a complete pathname. Entries under DIR can be assigned to this path. If this argument is omitted, the command deletes the specified device from the list.

LIST The LIST argument changes the display format of the current list. If no changes are desired, the LIST argument may be omitted from the ASSIGN command.

The end of the output lists the devices that can be accessed from AmigaDOS. These devices are described in detail in Chapter 3. Devices are separated from one another in the list by spaces. Device names more than three characters in length are not yet implemented.

Workbench 1.3 implementation:

Syntax: `ASSIGN NAME,DIR,LIST/S,EXISTS/S,REMOVE/S`

EXISTS The command must include the device name and the EXISTS argument. ASSIGN will search the assign list for the device and display the directory and device assigned. The following occurs when you use ASSIGN in conjunction with the Devs: directory:

Input: `ASSIGN DEVS: EXISTS`

Output:
```
Devs:       SYS:Devs
```

The ASSIGN command sets the condition flag to WARN if the device is not found. This error status can be used in a script file (see the chapter on Script Files for more information). The following script file tests for the existence of the Extras disk. The user is asked to insert the Extras disk if it isn't in the drive:

```
ASSIGN >NIL: Extras: exists
IF WARN
ECHO "Please insert the Extras disk in a disk drive"
ENDIF
```

The >NIL command directs all output to the NIL: device. This device acts as a trash can—the redirected data doesn't come out. Unwanted output can easily be suppressed this way. Error status can be read using the IF WARN command. The script file executes the IF WARN command if the Extras disk isn't found (warn = 5) and displays the specified text.

REMOVE A volume or device can be removed from the mount list with the REMOVE option. This option should only be used by software developers since it does not free up resources, it only removes the name from the list.

Workbench 2.0 implementation:

Syntax: `ASSIGN NAME,TARGET/M,LIST/S,EXISTS/S,DISMOUNT/S, DEFER/S,PATH/S,ADD/S,REMOVE/S,VOLS/S,DIRS/S,DEVICES/S:`

The ASSIGN command has eight new arguments and two new versions of ASSIGN have been added to 2.0: *non-binding* and *late-binding*. The command has also been optimized for size and speed.

TARGET The `TARGET/M` argument allows you to make multiple assignments to a single device. This could be used to store your own custom fonts in a separate directory from the `Fonts` directory. For example, the following command will allow two `Fonts` directories: the standard `Fonts` directory and your `Custom_Fonts` directory located on the RAM: disk.

```
ASSIGN FONTS: SYS:FONTS RAM:Custom_Fonts
```

The standard `Fonts` directory will be search first and then your `Custom_Fonts` directory. Once the assignment has been made the `ASSIGN` command will display the following:

```
Volumes:Ram Disk [Mounted]
Workbench2.0 [Mounted]

Directories:
CLIPS          Ram Disk:clipboards
ENV            Ram Disk:env
T              Ram Disk:t
ENVARC         Workbench2.0:Prefs/Env-Archive
SYS            Workbench2.0:
C              Workbench2.0:C
S              Workbench2.0:S
LIBS           Workbench2.0:Libs
DEVS           Workbench2.0:Devs
FONTS          Workbench2.0:Fonts
             + Ram Disk:Custom_Fonts
L              Workbench2.0:L

Devices:
PIPE AUX SPEAK RAM CON
RAW SER PAR PRT DF0
DF2
```

DISMOUNT The `DISMOUNT/S` argument allows devices and directories to be removed from the assignment list. This option should only be used by software developers since it does not free up resources, it only removes the name from the list.

DEFER The `DEFER/S` argument creates a late-binding assignment. This assignment only takes effect when the assigned object is accessed. This can be used to avoid constantly having to switch disks, since the

assigned object is only required when it is actually needed. The assignment remains in effect until explicitly changed.

PATH The PATH/S argument creates a non-binding assignment. It does not take effect until it's referenced and only remains in effect while it's needed. This can be very useful to avoid unwanted disk swapping if the disk in the drive contains the necessary directories.

VOL The VOL/S argument will only display information on the current volume assignments.

DIRS The DIRS/S argument will only display information on the current directory assignments.

DEVICES The DEVICES/S argument will only display information on the current device assignments.

2.2.9 ADDBUFFERS

Workbench 1.2 implementation:

Syntax: `ADDBUFFERS DRIVE/A,BUFFERS/A`

This command assigns a large buffer to a specified disk drive. When working with AmigaDOS, sometimes a command can be loaded from the drive before it's used for the first time, and then the command remains in memory for subsequent command calls. The reason for this is found in the disk drive buffer memory. The operating system loads all data into the disk buffer before it can be used elsewhere. If a program is small enough to fit in the buffer, it doesn't need to be recalled from the disk or hard disk again. This speeds up execution time.

DRIVE The DRIVE argument is the drive specifier to which the buffer should be assigned.

BUFFERS The BUFFERS argument specifies the number of blocks allocated for the additional buffer (1 block = 512 bytes).

The following example assigns 11 blocks of RAM to drive **DF0:**

```
ADDBUFFERS DF0: 11
```

Drive **DF0** is given an additional 11 blocks for working memory (1 block = 512 bytes). Through this addition one of the 160 tracks of a disk can be loaded into memory.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the ADDBUFFERS command has been optimized for compactness and speed.

2.2.10 WHY

Syntax: WHY

This command displays a response from the Amiga describing the reason a command could not be executed. In most cases AmigaDOS can be asked WHY the function did not work.

For example, you would like to read the startup sequence. You enter:

```
TYPE S/Startup-sequenze
```

The computer responds:

```
Can't open S/Startup-sequenze
```

You enter:

```
WHY
```

The computer responds:

```
Last command failed because Error code 205
```

Entering the command `Fault 205` explains the error: `Object not found`. We purposely misspelled `startup-sequence` above.

Workbench 1.3 implementation:

Version 1.3 of this command improved the error messages by passing the error number to the FAULT command and then displaying the message. If you rename the FAULT command the error number and not the message will be displayed.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The WHY command was also made an internal command.

2.2.11 FAULT

Workbench 1.2 implementation:

Syntax: `FAULT ,,,,,,,,,`

This command converts error numbers into descriptive text. Only some errors have texts. If a specific text doesn't exist, the word `Error` appears, followed by the error number. Two examples:

Input: `fault 10`
Output: `Fault 10: Error 10`

Input: `fault 120`
Output: `Fault 120: argument line invalid or too long`

Workbench 1.3 implementation:

Version 1.3 of this command improved the error messages.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The FAULT command was also made an internal command.

2.2.12 DATE

Workbench 1.2 implementation:

Syntax: `DATE TIME,DATE,TO=VER/K`

This command sets and reads the current time and date on the Amiga, independent of the `Preferences` editor.

TIME The TIME argument represents the clock time in HH:MM:SS format (H = hours, M = minutes, S = seconds) or just HH:MM format.

DATE The DATE argument must have the format DD-MMM-YY (D = day, M = month, Y = year). If the old date is less than a week old, you can enter the day of the week itself instead of the date format. Even if the old date is within a day of the present date, you can enter Yesterday. Either case installs the correct date.

TO=VER The TO=VER argument directs the date setting to a file. The following example sends the current date to the file DOUG:

```
DATE TO DOUG
```

Entering DATE without parameters displays the current day of the week, date and time:

```
Thursday 19-Jul-90 10:17:48
```

The calendar begins at January 2, 1978. The first of January is shown as unset. Time periods before that time are invalid.

Workbench 1.3 implementation:

The new DATE command now accepts one digit date input as well as two digit input. For example, in addition to the input date 01-Jul-90, you can also enter 1-Jul-90.

Workbench 2.0 implementation:

The new DATE command now accepts digit month input as well as month name abbreviation. For example, in addition to the input date 01-Jul-90, you can also enter 1-7-90.

The DATE command has been optimized for compactness and speed.

2.2.13 SETCLOCK

Workbench 1.2 implementation:

Syntax: SETCLOCK OPT LOAD|SAVE

This command places the time and date set by DATE into the Amiga battery-powered real-time clock (this is an option for early Amigas).

The real-time clock and the data entered in `Date` are independent of one another.

OPT LOAD The `OPT LOAD` argument transfers the real-time clock date and time to the system.

OPT SAVE The `OPT SAVE` argument transfers the system date and time to the real-time clock.

In most cases the command is used in the startup sequence of the boot disk to set the time. The command sequence `SETCLOCK >NIL: OPT LOAD` can be found on the `Startup-sequence`. The command sends a message to the `NIL:` device. This virtual device ensures that the output does not appear on the screen.

If you enter `SETCLOCK` without parameters and no real-time clock exists, the computer replies:

```
Internal clock not functioning
```

You will receive this message if you don't have a real-time clock in your Amiga. The entire procedure takes about six seconds to load. The command can also be erased from the `Startup-sequence`.

Workbench 1.3 implementation:

Syntax: `SETCLOCK LOAD|SAVE|RESET`

Version 1.3 of this command added the RESET argument. This command also had a minor bug in that the argument template was not displayed with the input of a "?". Instead the error message showing the correct usage was displayed.

Workbench 2.0 implementation:

Syntax: `SETCLOCK LOAD|SAVE|RESET`

Version 2.0 of this command has been optimized for compactness and speed. The 2.0 version of this command displays the argument template correctly with the input of a "?".

2.2.14 PROMPT

Syntax: `PROMPT TEXT`

This command changes the appearance of the DOS prompt. When the prompt appears, the computer is ready to receive input. The Amiga default prompt is the AmigaDOS task number followed by a greater-than character (1>). This can confuse the new user.

The PROMPT command lets you change the prompt display. If the text contains spaces, it should be placed in quotation marks. Example:

```
PROMPT "What do you want?"
```

If you enter PROMPT without any parameters, the prompt defaults to a greater-than character. If you want the number of the respective AmigaDOS task displayed, the combination %n must be entered. Example:

Input: `PROMPT "I am number %n !"`
Output: `I am number 1 !`

The old AmigaDOS 1.2 prompt can be restored by entering:

```
PROMPT %n>
```

Workbench 1.3 implementation:

Syntax: `PROMPT PROMPT:`

The new PROMPT command allows you to display the current drive and directory path as part of the prompt text. In addition to the command string %n, which shows the number of the actual AmigaDOS task, the command characters %s lets you display the last position of the CD command. For example:

```
prompt "%n.%s> "
```

The new prompt could look like the following:

```
3.Workbench 1.3:System>
```

You are in the third AmigaDOS task. The actual directory is the System: directory of the Workbench 1.3 disk.

Workbench 2.0 implementation:

Syntax: `PROMPT PROMPT:`

Version 2.0 of this command has been optimized for compactness and speed. The 2.0 version of this command has been made an internal command.

2.2.15 STACK

Workbench 1.2 implementation:

Syntax: `STACK SIZE`

This command specifies the amount of memory allocated for the stack. Each AmigaDOS task places DOS commands in a special memory location accessible from a machine language stack. Normally the size of the location is 4000 bytes per `CLI`. The amount of stack memory can be specified from 1600 bytes on up. However, if a large amount of memory is needed, the memory given to the `Shell` could be overwritten and a system crash could occur. The `DIR` command is especially susceptible to crashing. Try this on the Workbench disk when there is nothing important in working memory (this will usually crash the computer in AmigaDOS 1.2 and 1.3; in AmigaDOS 2.0 use `LIST ALL` in place of `DIR OPT A`):

```
stack 1600
dir opt a
```

The `SORT` command is also fussy about stack memory. This depends on the starting point of the data to be sorted. Unfortunately, there are no given values to avoid. Only trial and error help here.

Another interesting fact is that a new task always receives as much memory allocation as the `CLI` from which it was started. Remember this, or else memory can be used up very quickly.

If you are uncertain about the size of the stack, use the `STACK` command without any parameters.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Syntax: `STACK STACK/N:`

Version 2.0 of this command has been optimized for compactness and speed. The STACK/N argument is now specified as numeric. The 2.0 version of this command has also been made an internal command.

2.2.16 BINDDRIVERS

Syntax: BINDDRIVERS

This command integrates the device drivers (hard disk, plotter, etc.) found in the Expansion drawer into the system. You'll find this command used primarily in the startup-sequence of a boot disk. You must have the driver to operate the hardware. If you don't need the drivers, then you can delete this command from the startup-sequence, and the Expansion drawer from the Workbench. By doing this the system booting time shortens by a couple of seconds.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Version 2.0 of this command has been optimized for compactness and speed.

2.2.17 MOUNT

Workbench 1.2 implementation:

Syntax: MOUNT DEVICE/A

This command can add new devices to AmigaDOS. The basic configuration of the Amiga recognizes the following devices:

DF0:	Internal disk drive
PRT:	Printer
PAR:	Parallel port
SER:	Serial port
RAW:	RAW: window
CON:	CON: window
RAM:	RAM disk

These devices can be addressed immediately. New devices (e.g., hard disk partitions) can be installed using the Mount command. Mount waits for the name of the new device as a parameter. Information about

this device can be found in the text file `MountList`, contained in the `Devs` directory on the Workbench disk.

Here's some sample information about the 5-1/4" floppy disk drive device (installed as **DF2:** on some systems):

```
DF2:  Device = trackdisk.device
      Unit = 2
      Flags = 1
      Surfaces = 2
      BlocksPerTrack = 11
      Reserved = 2
      PreAlloc = 11
      Interleave = 0
      LowCyl = 0
      HighCyl = 39
      Buffers = 5
      BufMemType = 3
#
```

Any device can be entered in the `MountList`. Each entry must begin with the device name (in this case, **DF2:**) and must end with a number sign (#). The data between these strings specifies the device's characteristics. `Mount` accepts the following keywords:

Disk drives:

Keyword	Function
`Device`	Name of the device driver
`Unit`	Device number (e.g., 0 for **df0:**)
`FileSystem`	Label of a special `FileSystem`
`Priority`	Task priority (mostly 10)
`Flags`	Parameter for Open device (usually 0)
`Surfaces`	Number of sides of drive (for disks: 2)
`BlocksPerTrack`	Number of blocks per track
`Reserved`	Number of boot blocks (usually 2)
`PreAlloc`	(no function)
`InterLeave`	Device-specific (usually 0)
`LowCyl`	Number of small tracks
`HighCyl`	Number of large tracks
`Buffers`	Size of buffer memory in blocks
`BufMemType`	Type of memory: 0,1 = Chip or Fast RAM 2,3 = Only Chip RAM 4,5 = Only Fast RAM
`Mount`	1 = Device connected -1 = Device connected on first access

Other devices:

Keyword	Function
`Handler`	Path description of the device driver
`Stack`	Size of the processor stacks for the task
`Mount`	See above

Workbench 1.3 implementation:

Syntax: `MOUNT DEVICE/A,FROM/K`

The `MountList` can receive any name that follows the `FROM` argument:

```
MOUNT DF2: FROM Devs:Devicelist_1
```

The `MOUNT` command searches in the `Devs` directory for the file `MountList` if you omit the `FROM` argument.

Workbench 1.3 allows you to install new devices. A few of these new devices are briefly described here (see Chapter 3 for detailed information).

AUX: A serial port connection that doesn't store the data in a buffer. The important entries are already in the `MountList`, so the connection can be installed using the command sequence `mount aux:`.

PIPE: The device enables different tasks to exchange information. For example, if you want to send information from one `CLI` to another, this sequence allows you to make the exchange easily:

Input to 1st `CLI`: `COPY S/Startup-sequence TO PIPE:`

This information can be read in the second `CLI` from the `pipe:`

Input to 2nd `CLI`: `TYPE PIPE:`
Output: The Startup-sequence is listed

The statement for installing the `pipe:` is already in the `MountList`.

RAD: A recoverable RAM disk. Unlike the device `ram:`, data remains in memory even after the computer is reset. Not even a Guru Meditation can reset the `RAD:` device. Unfortunately, memory management is not dynamic, so `RAD:` takes up all of its allotted memory even when it is empty. The capacity of `RAD:` is included in the `MountList`.

NEWCON: A new window port that expands on the usual `CON:` window. The `NEWCON:` device manages a 2K buffer for temporary storage of the last input. The old input can be recalled and edited with the help of the cursor keys. The `NEWCON:` device can be used in conjunction with the `NEWCLI` command.

SPEAK: Controls Amiga speech output.

The new `MOUNT` command reads the keywords described above in addition to the following statements:

Keyword	Function
`MaxTransfer`	Maximum number of blocks that can be transferred
`Mask`	Address area that can be addressed by the DMA
`Handler`	Path description of the device driver
`GlobVec`	Global vector for the process, 0 sets up a private global vector, -1 is no global vector and if the keyword is absent the shared global vector is used.
`StartUp`	A string passed to the filesystem, handler or device on startup as a BPTR to a BSTR (see Chapter 8).
`BootPri`	Sets boot priority of a device, used with the recoverable RAM disk.
`DOSType`	Indicates the filesystem. For the FastFileSystem it should be 0x44F5301 otherwise 0x44F5300.

These statements are only evaluated in conjunction with the FastFileSystem.

Workbench 2.0 implementation:

Version 1.3 and Version 2.0 of the MOUNT command are identical.

2.3 Script File Commands

This section contains information about the commands used in conjunction with script files. Script files (called batch files in the MS-DOS world) are simple text files containing any number of AmigaDOS commands, written using ED or a word processor. The EXECUTE command runs these commands in sequence. The Script Files chapter contains detailed explanations and several practical uses for script files.

2.3.1 EXECUTE

Workbench 1.2 implementation:

Syntax: `EXECUTE NAME`

This command executes script files. Because script files are text files, They cannot be directly accessed like programs. If this is attempted, the computer responds with the `Error code 121: file is not an object module` error message.

The EXECUTE command needs the name of the file to be executed. For example, a script file named `printer` might contain the following line:

```
TYPE Text/Letter TO PRT:
DATE TO PRT:
```

The EXECUTE `printer` command works the same as if both of the above lines had been typed in from the AmigaDOS Shell. The script file prints the letter, followed by the current date and time.

As with most AmigaDOS commands, the filename and additional parameters may be added—these are transferred to the script file. The script file, in this case, must have a predetermined variable to which the parameters can be assigned.

The above script file should serve as an example of this. Instead of printing out the given text (`Text/Letter`), a variable can now be inserted, which can be assigned any name. The variable is declared in the example below using the .KEY *directive*:

```
.KEY name
TYPE <name> to PRT:
DATE TO PRT:
```

The `Printer` script file is now called using:

```
EXECUTE Printer Text/Letter
```

There are a few rules about using variables. They are as follows:

1. The .KEY directive, with which the variables are declared, must always be at the beginning of the script file.

2. If the assignment allows multiple parameters, they must be separated by a comma. The .KEY directive can only be used once, otherwise the error message `Execute: More than one K directive` is displayed. Example of correct usage:

```
KEY dataname, destinationdevice
COPY <dataname> TO <destinationdevice)
```

3. Replace the text between the greater-than and less-than characters with your own contents. There should be nothing about the variable name in their place, but instead the statement of what was assigned by the EXECUTE command.

Normally, these three points are all you need to know when working with variables in conjunction with script files. There are a few additional functions that should not be overlooked.

In addition to .KEYS, there are a number of directives beginning with a period that can be put in a script file:

`.DEF` This directive assigns given contents to a variable. This instruction can emerge anywhere in the text. A use for this is to give a firm name to a variable in case the `Execute` command is not given a definite name (default name). Such a script file can look like the following:

```
.KEY datafile,devicename
.DEF devicename PRT:
TYPE <datafile> TO <devicename>
```

When the `devicename` parameter is omitted, EXECUTE defaults to the printer.

For this use there is a special but very simple procedure. The variable name in the greater-than and less-than characters must be expanded by adding a dollar sign and the text that is to take the place of the variable, on the chance that the EXECUTE command isn't given any parameters. The above example would then look like this:

```
.KEY datafile,devicename
TYPE datafile TO <devicename$prt:>
```

In case a filename is entered but not a device name, the output automatically goes to the printer (PRT:).

.DOLLAR This directive changes the dollar sign ($) placed at the beginning of a text to any other character. For example:

```
.DOLLAR #
```

The corresponding line under .DEF would now have to read:

```
TYPE datafile TO <devicename#prt:>
```

.BRA This directive has a task similar to .dol. This allows the less-than character (<) to be replaced by another character.

.KET This directive is similar to .BRA, except it changes the greater-than (>) sign.

. A period followed by at least a space allows the user to insert a comment line. BASIC programmers use a REM statement for this.

.DOT This directive changes the period preceding each instruction to another character.

Script files should be used without any other control characters, otherwise it becomes too confusing.

Workbench 1.3 implementation:

Script files are still called through EXECUTE. By adding the S (Script) flag, it's now possible to start script files by entering their names. The Script flag must be set first using the PROTECT command. The following sequence sets the flags for a script file named Test_Batch:

```
PROTECT Test_Batch +s
```

Workbench 2.0 implementation:

The current shell number can now be accessed with <$$>. A blank comment line can be inserted with ./ in the script file. Version 2.0 of the PROTECT command has been optimized for compactness and speed.

2.3.2 ECHO

Workbench 1.2 implementation:

Syntax: `ECHO TEXT`

This command makes it possible to direct a character string to any output device. The default device is the screen:

```
ECHO "Hello, Doug!"
```

You must add a greater-than character and another output device name to send the output to another device:

```
ECHO >PRT: "One more beer and I'll go home."
```

The text must be enclosed in quotation marks if any spaces exist in the text.

The ECHO command features an optional parameter of the `*n` character combination. This combination forces a linefeed in the text output:

```
ECHO "Careful*n        Stairs!"
```

The output on the screen looks like this:

```
Careful
       Stairs!
```

In the rare instance that you wanted to use `*n` as an actual entry in the text, use the character string `**n`.

Workbench 1.3 implementation:

Syntax: `ECHO ,NOLINE/S,FIRST/S,LEN/S`

Three arguments were added, one for line suppression and two that allow the echoing of a substring.

NOLINE The `NOLINE/S` argument suppresses the linefeed that usually follows after the output.

FIRST The `FIRST/S` indicates the beginning position in the string to echo.

LEN The `LEN/S` indicates the length of the string, beginning at the FIRST position, to echo.

Workbench 2.0 implementation:

Syntax: `ECHO ,NOLINE/S,FIRST/K/N,LEN/K/N`

The FIRST and LEN arguments are specified as numeric keywords. Operation is identical to Workbench 1.3.2, but the command has been optimized for compactness and speed. The `ECHO` command has also been made an internal command.

2.3.3 FAILAT

Workbench 1.2 implementation:

Syntax: `FAILAT RCLIM`

This command halts a command sequence if the Amiga reaches a specified error return code limit. Each AmigaDOS command and many other programs return an error number if an error occurs during execution. In AmigaDOS most numbers are assigned a related error text so that by using `FAULT`, followed by the respective number, the explanation can be read. For example, the number 216 means that you tried to delete a drawer that still contained entries.

If the error number for an AmigaDOS command inside of a script file is greater than or equal to ten, the script file stops working and returns control to the main program (for example, back to the Amiga `Shell`). This error limit can be read by using `FAILAT`. This is very useful because the limit could be anywhere. In some cases it's desirable when a script file reports a warning (an error number less than 10).

An example: When compiling, the difference between warnings and errors is most obvious. Warnings can usually be ignored because they are caused by a poor programming style. If the error limit is set in a script file of a compiler, the work is stopped as soon as it encounters incorrect data. This prevents the calling of other work operations (assembling, linking).

To set a new error limit in a script file, `FAILAT` requires an argument of the new error number at which the operation should be stopped. This new limit is valid only while work is being done in the script file. After that it is automatically reset to 10.

If a new error limit is given directly from the `Shell`, the limit is also valid in the script file called from the `Shell`. If the limit is undefined, then the error limit returns to 10. If the `FAILAT` command does not emerge in the data file the given error limit remains unchanged.

Each new `Shell` called automatically supersedes the error limit of the `Shell` that it was called from. After it is called the limits can be changed independent of one another.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The `FAILAT` command has also been made an internal command.

2.3.4 QUIT

Workbench 1.2 implementation:

Syntax: `QUIT RC`

This command exits a script file at any point. The `QUIT` command is unnecessary at the end of a script file. If you want the script file to tell you what went wrong, `QUIT` can display the desired error number. Control returns to the calling script file and the following text appears if the error number is greater than or equal to 10:

```
Quit failed returncode xx
```

The `xx` represents the error number.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The `Quit` command has also been made an internal command.

2.3.5 IF/ELSE/ENDIF

These commands execute certain parts of a script file if specific conditions are met. These three commands must be handled as one: ELSE and ENDIF are only allowed to be used in conjunction with IF.

IF/ENDIF The simplest case only requires IF and ENDIF:

```
IF EXISTS Text/Letter
TYPE Text/Letter TO PRT:
ENDIF
ECHO "Have I printed the letter yet or not ?"
...
...
```

In this example the TYPE command executes because the data file Letter really existed in the subdirectory Text. In this case it doesn't matter about the rest of the script file directly under the ENDIF.

It can be determined whether data files are contained in a disk drive or on the RAM disk. Another set of codes that are allowed to follow the IF:

EQ EQ compares two texts for the same contents:

```
IF "That is the text" EQ "That is the text"
ECHO "Yes, the two texts are equal"
ENDIF
```

Some inquiries naturally do nothing because the interrogation can also be omitted. With just the EQ command, the text remains unchanged. Using EQ in conjunction with batch variables is interesting (see the description of the EXECUTE command). Two examples:

```
.KEY input
IF <input> EQ Letter
ECHO "You entered the word 'letter' !"
ENDIF

.KEY input
IF <input> EQ ""
ECHO "You didn't enter anything !!!"
ENDIF
```

Here you must differentiate between the variable input, the contents of input and the text letter. It is important to note that when comparing text, it does not matter whether it is in capital letters or not. If letter EQ LETTER returns the same result.

IF FAIL Using IF FAIL determines whether the last command had an error number greater than or equal to 20. This evaluation is useful when, before the use of the command, the error limit has been changed from 10 to a larger value than 20. If not, the execution of the script file is interrupted.

ERROR IF ERROR is the same as IF FAIL. In this case, however, the error limit stays at 10.

IF WARN The error limit for IF WARN is set at five. It is not necessary to set the error limit higher than 10 with FAILAT.

The labels IF WARN and ERROR should not be confused: If IF WARN traps error number 225, for example, this is a fail error instead of a warning. It recommends that a higher error limit be set with FAILAT.

NOT If NOT is added before any of the above conditions, the opposite of the declaration is done. For example:

```
.KEY text
IF NOT EXISTS <text>
ECHO "I don't have any such data file"
ENDIF
IF EXISTS <text>
ECHO "Here it goes !"
TYPE <text> TO PRT:
ENDIF
```

The script file needs the name of a text file contained in the variable text. In the first section it tests to see if the data file doesn't exist. If it does not, the first ECHO message appears.

After that, IF EXISTS is used again to see if the file actually exists. If it does, the script file prints it on the printer (this only works with true text files).

ELSE The ELSE command can easily be built into a script file as an alternative to the IF NOT statement. The above example looks like the following when you're done using the ELSE command:

```
.KEY text
IF NOT EXISTS <text>
 ECHO "I don't have that data !"
ELSE
 ECHO "Here it is !"
 TYPE <text> to PRT:
ENDIF
```

This example delivers the same result as before, except faster and easier.

Finally, a few comments about the three commands. Each IF/ELSE/ ENDIF block is allowed to have any number of lines. The block must

end with either an ELSE or an ENDIF. A block can also have any number of interlocking IF commands. An ELSE or ENDIF must always be associated with the last IF in a block. The example below evaluates how many parameters the EXECUTE command is given (a maximum of three). It becomes easier to see the function of the program through the structured indenting of the program:

```
.KEY text1,text2,text3

IF NOT <text1> EQ ""
   IF NOT <text2> EQ ""
      IF NOT <text3> EQ ""
         ECHO "All three inputs exist"
      ELSE
         ECHO "The three inputs are missing"
      ENDIF
   ELSE
      ECHO "The second and third inputs are missing"
   ENDIF
ELSE
   ECHO "No input has been made"
ENDIF
```

Workbench 1.3 implementation:

GT
GE
VAL
EXISTS

GT is for greater than comparisons. GE is for greater than or equal to comparisons. The VAL option may be specified to compare numbers. EXISTS checks for the existence of a file.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The IF/ELSE/ENDIF commands have also been made internal commands.

2.3.6 ASK

Workbench 1.2 implementation:

Syntax: ASK PROMPT/A

This command has the computer wait for a response from the user before continuing with the script file.

The ASK command can either be given without arguments or with text that displays a question. The computer waits for an answer, either Yes or No (Y or N), followed by the <Return> key. If something else is entered, the ASK command waits until a correct answer is given. The evaluation occurs through error code number five so the command can confirm the input. The following example demonstrates the first reaction to different input:

```
FAILAT 5
ASK "Should I stop? (y/n)"
ECHO "Good, then I'll go further"
...
```

Because the error limit is usually set at 10 for stopping script file execution, it must be set to five here so that entering Yes would return you to the AmigaShell.

This solution hardly satisfies everyone. It would be better for the user if two different program lines could work at once. What you can do is use the IF WARN command from Section 2.3.5. That adds the nuisance of lowering the error limit to a value smaller than six. Example:

```
ASK "Do you know the ask command ? (y/n)"
IF WARN
   ECHO "Very good, go on !"
ELSE
   ECHO "Set at six !"
ENDIF
```

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The Ask command has also been made an internal command.

2.3.7 SKIP/LAB

Syntax: SKIP LAB

Skip The SKIP command is the script file equivalent of the BASIC/C GOTO command, or assembly language's jmp instruction.

Lab

If a script file encounters a SKIP command, the text file is searched for the LAB (label) command. The file executes at the routine specified by the label. If you add a name to the SKIP command, the script file jumps to the label of the same name. For example:

```
ASK "Can you go around with Skip and Lab ? (y/n)"
IF WARN
SKIP mark
ENDIF
     ECHO "Not too bad"
QUIT

LAB mark
     ECHO "Use only in moderation"
```

The program text cannot be re-entered with the SKIP command. As with other programming languages that use a jump command, SKIP should be reserved for cases where there is no alternative, since it detracts from structured programming. In the above example it makes for a sloppy program because it exits an IF/ENDIF construction. That should be prevented whenever using the SKIP command. In almost all cases an IF/ELSE/ENDIF construction is the best solution.

Workbench 1.3 implementation:

The BACK option was added; this allows you to skip back to the beginning of a script file before searching for a label.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The SKIP/LAB commands have also been made internal commands.

2.3.8 WAIT

Workbench 1.2 implementation:

Syntax: `WAIT SEC=SECS/S,MIN=MINS/S,UNTIL/K`

This command delays script file execution for a specified amount of time. A typical example of a needed pause is the execution of two tasks that access the same disk drive at the same time (excepting the RAM disk). If the directory of the disk in drive **DF0:** is listed in one AmigaDOS window using the DIR command, and the LIST DF0: command lists the directory of the same disk in another AmigaDOS

window, the two commands are executed parallel to each other. The net effect is that it takes longer for the commands than if they had been entered one after the other. Because both processes must access the disk, each command can only access a few tracks during execution time. A lot of time must be allotted because the drive head must always be changing its position.

If you wish to load two programs with the Startup-sequence, we recommend that you wait for the first program to load using the WAIT command. The time needed to wait is entered as the argument. The time can be entered in seconds, minutes or in system time format. WAIT without any parameters waits one second. Some examples:

```
WAIT                 waits 1 second
WAIT 5               waits 5 seconds
WAIT 5 sec           (same as 2)
WAIT min             waits 1 minute
WAIT 5 min           waits 5 minutes
WAIT UNTIL 14:30     waits until 14:30 (2:30 pm)
```

You can interrupt the WAIT command by pressing <Ctrl> <C>. In Chapter 5 we'll show you how WAIT can be used to make an AmigaDOS alarm clock.

Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed.

2.3.9 VERSION

Syntax: `VERSION [<library name>] [version] [revision]`

This command returns the version and revision number of the Workbench from a device or library. When the VERSION command is entered without arguments, you receive statements about the Kickstart and Workbench versions. For example:

```
Kickstart version 33.180. Workbench version 34.4
```

VERSION can have a special library of device names attached:

Input: `VERSION trackdisk.device`
Output: `trackdisk.device version 33.127`

It is possible to test the version number. Error code 5 is returned if the given version number is greater than the one tested. The error status can be evaluated from within a script file with the help of a `IF/ELSE` construction. The following script file calls the `Math` program if the fast math library Version 34.44 or less is present. Otherwise it returns an appropriate message.

```
VERSION >NIL: mathieeedoubbas.library 33.44
IF WARN
     ECHO "the fast Math library is not there !"
ELSE
     RUN Math
ENDIF
```

Workbench 1.3 implementation:

Syntax: `VERSION [<library name>] [version] [revision] [unit]`

The [Unit] option allows you to specify a unit number other than 0, used when accessing multi-unit devices. This command also had a minor bug in which the argument template was not displayed with the input of a "?". Instead the error message showing correct usage was displayed.

Workbench 2.0 implementation:

Syntax: `VERSION NAME,VERSION,REVISION,UNIT:`

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed. The 2.0 version of this command displays the argument template correctly with the input of a "?".

2.4 The Editors

The ED and Edit programs are two large programs that make it possible to edit text files. MEMACS (MicroEmacs), which is available with Workbench 1.3 and 2.0, is a screen-orientated editor with drop-down menus. It can be started from AmigaDOS and is an excellent editor. Since it has drop-down menus and this book is about AmigaDOS, we will not discuss this editor in this book.

ED

ED is a *full screen editor*. It can load the entire text into the working memory of the Amiga and display an entire screen of that file. With the help of the four cursor keys the cursor can be placed at any position inside the window. You can then edit the text at that position. You can scroll the text to see data above and below the screen window. If the text to be edited has more columns than the ED window shows, the window scrolls left and right when the cursor moves beyond either margin. Anyone that has worked with AmigaBASIC is at least familiar with the principle of screen editors.

ED 2.0

ED 2.0 has been updated by John Toebes III, and the improved version is excellent. The new 2.0 operating system allows programmers to add menus to their programs. The menus for the latest version of ED are contained in the ED-Startup script file in the S: directory. They allow you to use the mouse when using ED to access often used commands. The normal ED commands are still available. The new version includes menus and the new standard Amiga file selector.

EDIT

The EDIT program is a *line* editor. The basic difference from ED is that you edit text line-by-line through different commands. You cannot manipulate text within a window, similar to the CLI.

There are many reasons for using two editors. These two editors allow the user to edit script files or enter source codes for compiled languages. In most cases ED is much easier to use, and gives a better overview of the text (very important when programming).

EDIT needs relatively smaller amounts of memory than ED, because the entire text does not need to be loaded into memory. In cases involving large files that ED cannot load, EDIT can help. EDIT also lets the user open more than one source file at a time. Overall, EDIT has more flexibility than ED.

It would take an entire book to describe EDIT's capabilities. Only its basic functions are supplied in this section.

2.4.1 Reading text with ED

ED is usually accessed through the ED command and a file/path:

```
ED Text/Prog
```

The Text/Prog argument is the pathname of the text file you want edited. Two different conditions can exist when invoking ED in this way.

Existing filename

This case loads the text file into the working memory. If this is successful, the ED window appears, displaying all or part of the text file (depending on the file's size). If the loading operation is interrupted, the Amiga displays the message Unable to open window... on the screen. This error could occur if the filename given is in fact a directory, or if the file doesn't contain text characters. These are indicated by the messages x is a directory and cannot be edited or File contains binary.

Nonexistent filename

If the file doesn't exist, an empty ED window appears. When starting ED, you can specify the SIZE argument, which sets the working memory size in bytes. The following allocates 60,000 bytes to the test file in the text directory:

```
ED Text/Test SIZE 60000
```

If the SIZE is missing, ED defaults to a size of 40,000 bytes. The SIZE should be increased if you want to load a larger text file. Now work can continue with ED. After the size and position of the window has been set, it is possible to move all over the screen and manipulate data and even add new text. The <Backspace> and <Del> keys function as usual: <Backspace> erases the character immediately to the left of the cursor, and <Del> erases the character that the cursor is on. The mouse cannot be used with ED.

There are two different ways to control ED (ED 2.0 has been updated to include menu control):

1. Direct mode by pressing the <Ctrl> key and another key. The respective command is executed.

2. In the command line. Pressing the <Esc> key displays an asterisk at the bottom left of the screen. As long as the asterisk is visible, you are in command mode (i.e., you can't edit the text). Entering and executing a command, or by pressing <Return>, you can exit this mode.

Basic ED commands The end of this book shows a complete listing of the commands that can be executed from ED. Here we are only presenting the most important commands, but they are sufficient in most cases.

Direct mode

<Ctrl><A> Insert line.

<Ctrl><B> Delete line.

<Ctrl><G> Displays the last <Esc> command (important for searching and replacing).

<Ctrl><Y> Delete from cursor to end of line.

<Esc> mode

<Esc><X> (eXit) Saves text to disk and exits ED. A copy of the file named `ed-backup` is placed in subdirectory `t` of the disk.

<Esc><Q> (Quit) Exits ED without saving the text. If you have entered any data, the program will ask for confirmation from you before quitting.

<Esc><SA> (SaveAs) Saves the text without exiting ED. If you want the text saved under a different name, the name can be changed to a new name or an already existing name.

Note: The old contents of a previously existing file are lost forever.

It is possible to send the data directly to a peripheral device: <Esc><SA> "PRT:" sends the text to the printer.

<Esc><J> (Join) Combines two lines into one. This is very useful when a line has been accidentally separated by the user by pressing the <Return> key in the middle of a line. The cursor must be placed at the end of the top line.

<Esc><BS> (BlockStart) Marks the beginning of a block of text for different block operations. The line in which the cursor currently stands is the top line of the block.

<Esc><BE> (BlockEnd) Marks the end of a block of text for different block operations.

<Esc><IB> (InsertBlock) Places a copy of the block marked out by <BS> and <BE> at the current cursor position.

<Esc><DB> (DeleteBlock) Deletes the block marked by <BS> and <BE>. The line is removed from the text.

2.4.2 Text handling with EDIT

Forget everything that we just talked about regarding ED. EDIT works on a completely different principle:

The working memory buffer of EDIT can only hold a few lines of text at a time. Under normal circumstances, the user edits these one after another. When editing is complete for the last line of the buffer, EDIT automatically loads the next line from the data file and writes the previous lines to a destination file. EDIT requires both files (this is a major difference from ED).

The user can only edit the lines currently in the buffer. It is also possible to scroll up a limited number of text lines. If a line has left the buffer and been written to the destination file, it is no longer accessible by EDIT.

When the session with EDIT ends, the complete contents of the buffer are saved to the destination file. The rest of the source file must eventually be saved so that data isn't lost. You can exit EDIT without read or write operations taking place.

EDIT also lets you open and read different data files while editing. Each new data file is superimposed over the beginning of the original source data file. When you return to the original file, it reopens and assumes the original position.

Finally, EDIT has another feature: It can read EDIT commands from any properly configured data file as well as from the keyboard.

2.4.3 Parameters of EDIT

Workbench 1.2 & 1.3 implementation:

Syntax: `EDIT FROM/A,TO,WITH/K,VER/K,OPT/K`

The EDIT command itself is started with EDIT. The arguments are as follows:

FROM The FROM argument specifies the name of the file to be edited. This data file must already exist (completely new text cannot be created using EDIT).

TO The TO argument specifies the name of the destination file to which the data are written. If this name is missing, EDIT creates a work file in the t subdirectory and places file data in it. When you quit EDIT, this work file receives the complete pathname of the source file as given in the FROM argument. The original source file is placed in the t subdirectory under the name EDIT-BACKUP until it's overwritten by further work with EDIT.

WITH The WITH argument loads a file which specifies commands. This file can give commands just as the user can give commands from the keyboard.

VER The VER argument directs EDIT's output to a device other than the screen. VER Data_File would put the input into a file named Data_File. Using VER con:10/10/300/100/VerWindow places the contents of such a file in a window.

OPT P The OPT P argument specifies the number of lines allowed in the buffer. Example: OPT P100 configures the buffer to hold 100 lines (40 lines is the default). This is very practical when more system memory is needed.

OPT W The OPT W argument changes the maximum line length to a value other than 120. Example: Opt W81 sets line length to 81 characters.

OPT PxWy The OPT PxWy argument is a combination of OPT P and OPT W. The x and y arguments are the values for these arguments.

Workbench 2.0 implementation:

Syntax: EDIT FROM/A,TO,WITH/K,VER/K,OPT/K,WIDTH/N,PREVIOUS/N:

Two new arguments have been added to the Version 2.0 of EDIT. EDIT determines the amount of memory required by multiplying the WIDTH and PREVIOUS values.

WIDTH The WIDTH argument specifies the maximum line width. The default value is 120.

PREVIOUS The PREVIOUS argument specifies the maximum number of previous lines. The default value is 40.

2.4.4 Starting EDIT

In most cases, you enter the name of the file to be edited when you start EDIT. As was explained above, the edited lines are placed in a help file named EDIT-BACKUP.

The EDIT prompt (a colon) appears after you invoke EDIT. It waits for user commands, much like AmigaDOS. Because of the line orientation, you must search for the next line to edit. EDIT automatically numbers all the lines of the source file internally. Immediately after you start EDIT, line 1 of the source file is the first line to be edited. Unfortunately the contents are not automatically displayed. To reach another line, there are different methods:

a) Entering <N> (Next) places the user at the next line.
b) Entering <P> (Previous) places the user at the previous line.
c) Entering <M><x> (Move) places the user at line number x.

To display the contents of line 1, for example, it is sufficient to enter <N> followed by <P>. Multiple commands can be entered at once, but as in ED, they must be separated by semicolons. EDIT does not distinguish between lowercase and uppercase letters.

The <P> and <M> commands let you return to the line of the buffer not written to the destination file. As soon as a line with a number greater than 40 is reached, many of the previous lines are placed in the destination file. If the user tries to go back to line number 1, for example, the error message `Line number 1 too small` appears.

Preceding the <P> and <N> commands with a number executes the command multiple times. For example, `10N` advances EDIT 10 lines in text.

The <F> (Find) command lets you find a specific string within the data file. The command must be followed by the search text enclosed by any characters. For example:

```
F ?Key?
```

EDIT searches for the current line number containing the word "Key."

If you omit input following <F>, the command searches for the last text string searched for. This is very practical when looking for more text that contains a certain search string. You must advance to the next line after a successful search using <N>, so that the same line doesn't get returned constantly.

2.4.5 Editing Text

After finding the designated text, you can make changes to it. These changes cannot be made directly to the line (as opposed to ED), but must be made by using certain commands. The important commands in EDIT are:

e (exchange) Substitutes one character string with another. Example: The line reads: "Edit is difficult to use."

```
Input:   e/difficult/easy
Result:  Edit is easy to use.
```

or:

```
Input:   e/difficult//
Result:  Edit is to use.
```

a (after) Inserts a given text behind a certain character string. Example: The line reads "Edit is a program."

```
Input:   a/a/flexible /
Result:  Edit is a flexible program.
```

b (before) This command inserts a given text before a certain character string. Example: The line reads "Edit can do more !"

```
Input:   b/more/much /
Result:  Edit can do much more !
```

d (delete) Deletes the current line. The line number disappears; the text is not re-numbered. A line can be deleted by entering the line number in the command line. Entire text passages can be removed if the start and end line numbers are given. For example:

```
"d 10 100"
```

Lines 10 to 100 are erased.

i (insert) Inserts the following lines preceding the current line. Entering a <Z> in a separate line ends insert mode. The buffer contents are renumbered starting with the first newly entered line. For example: The texts read:

```
20. "The input mode"
21. "makes everything too complicated."
```

Line 21 is the current line, and the following input is made:

```
i
and working reasonably
with Edit are possible.
It should not be thought impossible.
```

The result looks like this:

```
20. "The input mode"
21. "and working reasonably"
22. "with Edit are possible."
23. "It should not be thought impossible"
24. "makes everything too complicated."
```

2.4.6 Multiple Files

It's possible to open more than one source file from EDIT. The command for this reads:

```
FROM .datafile
```

After this command new lines are called only from the data file with the name datafile. As with the original source file, input in the new file begins with the first line of the text.

Using FROM without parameters returns to the start of the original source file. The program basically leaves all channels open, and marks how many lines of each data file have been read already. If after closing a data file using the command CF .datafile (CloseFile) the file is opened again with FROM .datafile, the lines that were already read can be called into the buffer one more time.

2.4.7 Command Macros

Edit can receive command *macros* (program information) from a data file that contains all of the normal Edit commands. The name of this file is given either at the start of the program after the addition of the WITH argument or the C command can be used when working with EDIT. A macro file can look like the following:

```
i
****************************************
*  Program       :                     *
*  Author        :                     *
*  Date          :                     *
*  Language      : "C"                 *
*  Assembler     : Aztec c68/am-c v3.4 *
****************************************
z
```

If this introduction is inserted before the active line in EDIT, entering C followed by the filename between two periods is all that is needed. Example:

```
C .introduction.
```

The insertion is not ordered by the C command, which just calls the file. The I command emerges here, through which the following text, up until the Z, is inserted before the active line. As soon as the end of this file is reached, or a line with the Q command occurs, EDIT returns to command level. This does not necessarily have to be the keyboard again because a command file is also allowed. In such cases a C command is contained in the command data file.

A macro file can be constructed for any imaginable case. If you know the situation well, working with EDIT can be much faster than ED.

2.4.8 Quitting Edit

Normally, the <W> (windup) command exits EDIT. The contents of the buffer and the rest of the source data file are copied to the destination file in subdirectory t with the name EDIT_BACKUP. If a name for the destination file wasn't specified at the beginning of the work with EDIT, the work file in subdirectory t receives the name of the source file. After that all the channels close and the program ends.

EDIT can also be exited using the STOP command (copy procedures are not executed). This leaves the destination file incomplete. If a destination file isn't given at the start of the editing session, no renaming is done on the work file. The source data file is also unchanged and remains under the same name.

See the Appendix for complete descriptions of EDIT's commands.

3.
Devices

3. Devices

A *device* is simply a piece of hardware with which the computer can exchange information. The disk drive is a typical device.

This data exchange between computer and device doesn't always have to go in both directions. A printer only accepts data, while a mouse only conveys information to the computer.

The description of the ASSIGN command includes a list of devices that can be accessed from AmigaDOS. The standard devices of the Amiga are listed below:

```
PIPE AUX SPEAK CON RAW
SER PAR PRT DF0
```

A colon (:) must always follow the device name, so that AmigaDOS can tell devices apart from directories or filenames.

This chapter describes the individual device names and what you can do with these devices.

Handlers

Handlers are found in the L: directory. Handers are treated as if they are actual physical devices even though no hardware is required for their operation. The SPEAK:, PIPE: and AUX: devices are handlers. Handlers must be MOUNTed before they can be used. This is usually done in the Startup-sequence or the StartupII script file. They must also be described in the MountList located in the DEVS: directory.

3.1 Floppy Disk Devices (DFx:)

All devices beginning with the letters **DF** are Amiga floppy disk drives. A total of four disk drives can be connected at one time (**DF0:-DF3:**). The drive specifier **DF0:** represents the internal disk drive on any Amiga; **DF1:** represents the first external disk drive (Amiga 500 and 1000) or the second internal disk drive (Amiga 2000); and so on. The `Devices` section of the `ASSIGN` list contains many references to the letters **DF**.

All AmigaDOS commands default to drive **DF0:** in the basic Amiga configuration (with only one disk drive). If you enter an AmigaDOS command without a disk drive specifier, AmigaDOS automatically accesses drive **DF0:**. The following command accesses the directory in drive **DF0:**

```
DIR
```

The following command accesses the directory in drive **DF2:** (the second external disk drive in some units, the second internal disk drive on the Amiga 2000):

```
DIR DF2:
```

See the descriptions of the `LIST` and `CD` commands in Chapter 2 for the problems that can occur in disk directory handling.

Workbench 2.0 implementation:

The device driver has been optimized for compactness and speed in AmigaDOS 2.0.

3.2 The RAM-Handler (RAM:)

The RAM-handler simulates a disk drive device (RAM:) with the Amiga's working memory. Handlers are treated as if they are physical devices. The word *RAM* is short for Random Access Memory, a type of memory that allows free access (both reading and writing). With few exceptions the RAM disk can be used like any other disk drive. The RAM disk's biggest advantage over floppy disk drives is high speed data exchange.

There is a disadvantage to using a RAM disk for storage: The contents of the RAM disk are temporary; they vanish when you turn the computer off, or when it crashes. Because of this, important data should be saved from the RAM disk to a "real" disk drive from time to time.

Another disadvantage is the memory requirement of the RAM disk. The memory capacity for the RAM disk is *dynamic*. The more data you store in the RAM disk, the less memory you have available for applications and user memory. The more system memory you have available, the more data you can store on the RAM disk. The RAM disk doesn't give useful information about its capacity (it's always 100% full according to the disk gauge on the left border of the RAM Disk window). This makes sense because the system only supplies the memory it needs and no more.

You must create a RAM disk before you can work with it. The following command opens a RAM disk:

```
DIR RAM:
```

You'll find the command in the Startup-sequence of 1.3, so that the RAM disk is immediately accessible. If the Startup-sequence installs a RAM disk and the user didn't want it present, he's out of luck—there is no DeleteRamDisk command. Section 5.3 explains how to use the RAM disk to decrease disk swaps when using only one drive.

Note: The following disk commands do not work with the RAM disk. Chapter 2 supplies detailed information about each command.

ADDBUFFERS This command produces the error message:

```
Warning: Insufficient memory for buffers
```

It would be a waste of memory to assign both a RAM disk and buffer memory to RAM, if the operating system did let you do this.

DISKCOPY The data files or directories of a RAM disk can only be copied one at a time using the COPY command. Copying the entire disk using the DISKCOPY command is impossible.

FORMAT The RAM disk doesn't need to be formatted before using it for the first time. If you click on the RAM DISK icon and select the Format disk.. item in the Icon menu (in Workbench 1.3 you would select the Initialize item from the Disk menu) the Amiga lets you get as far as the OK to format... requester. If you click on the OK gadget, the Amiga displays the following requester:

```
Format  failed - (1.3 Initialization failed)
cannot find handler
```

If you try to format the RAM disk using AmigaDOS Format command, AmigaDOS displays:

```
Format failed - (1.3 Initialization failed)
cannot find handler
```

RELABEL You shouldn't assign another name to the RAM DISK in AmigaDOS 1.3. If you do you will not be able to access the RAM DISK from the Workbench. Version 2.0 has solved this problem. Version 1.2 would not allow you to rename the RAM DISK.

INSTALL This command turns normal disks into boot disks. The Amiga can be started using these disks. The RAM DISK cannot be used as a boot disk.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the handler has been made an internal handler and is not located in the L: directory.

3.3 The Parallel Device (PAR:)

This device allows the Amiga to access Centronics interfaced hardware. The device works through the *parallel port* on the case of the Amiga and must first be connected before you access it. The PAR: device is *parallel* because all eight bits of a byte are transferred at once. It is also possible to transfer information one byte after another or bit for bit (see the next section for a description of *serial* transfer).

The connection can be used for more than one device. For example, an analog/digital video converter can be connected, and the video and audio signals will be converted to a format that can be understood by the computer. In this case the data from outside is sent to the computer through the connection. Other data flow directions are possible. A typical application for PAR: is a printer connected to the Amiga. The actual information runs through the connection to the printer. The reason for using a printer with a parallel connection is found in the handling of this device.

The speed at which the transfer of data takes place depends on how fast the data from the device can be processed. In addition to the eight lines used for transferring the data, there is an additional line used for *handshaking*. Over this line the data receiving device (printer) informs the transmitting device (Amiga) that it is ready to receive new data. This maintains optimal data transfer speed.

Workbench 2.0 implementation:

The device driver has been optimized for compactness and speed in AmigaDOS 2.0.

3.4 The Serial Device (SER:)

The serial device is also known as the RS-232C interface. The connection point on the Amiga is called the *serial port* and can be used for a wide variety of functions (modem, MIDI, etc.). The serial port transfers individual bits one after another, and not at the same time like the PAR: device. Each data direction also requires one signal. Parity is chosen before using the connection (even, odd or none). This eighth bit is automatically set so that either all set bits are always even or all set bits make an odd number. The receiver must be set at the same parity. In a few cases, parity can discover transfer errors. Only the bottom seven bits of the byte can be transferred when the parity bit is active. The remaining bit is sent as the *parity bit* (transfer control bit).

Each transfer is synchronized by one start bit and two stop bits. The speed at which the single bits are transferred must be identical at both the sending and receiving devices. This speed is traditionally measured in Baud (after the French inventor Baudot). One baud is equal to the transfer rate of about one bit per second.

The RS-232 connection also has handshake protocols. There are three ways to achieve correct data transfer:

Using command bytes

This method is usually called xON/xOFF protocol. This method assumes that the connection is bidirectional. As soon as a device cannot receive any more data, it sends a special message (xOFF) through its return line. The sending device stops data transfer until it receives the xON message from the receiving device. The advantage of this method is that only three lines are needed, and that's why a three-wire handshake is frequently used in telecommunications. The operating system automatically looks in the active program for correct utilization of the command characters.

Using control lines

This method is similar to the handshake used with the PAR: device. It requires additional wires, Ready TO Send/Clear To Send and Data Set Ready/Data Terminal Ready (RTS/CTS and DSR/DTR), over which additional information can be exchanged. A data channel used only for return messages can be established. The advantage of this method is the faster transfer speed because the control codes don't go through the relatively slow data channel.

No Handshaking

If the user is 100% certain that the data-receiving device can process the incoming bytes faster than the sender is sending them, then you can conceivably do without the handshaking. This method is most useful when a fast transfer of data from one computer to another is desired, with the least amount of expense (2 lines). As a permanent solution this method is ineffective because it takes too much time to configure.

All parameters must be set with the `Preferences` editor before using the serial port. In addition, there is a gadget called `Buffer Size` that can be used to change the size of the transfer memory for the receiving data. This buffer holds the receiving data in case the receiving program is not ready. If it takes too long to read the data and no handshake takes place, this buffer can be overwritten. Data that was in the buffer are lost.

Null modem cables are ready made cables that have the correct wiring to allow two computers to easily exchange data thru the serial ports. The software used is the same software used with a modem, only the cable without a modem connects the two computers together. Hence the name null modem cable. Most computer dealers carry null modem cables, in case you need to transfer data from a laptop into your Amiga.

Workbench 2.0 implementation:

The device driver has been optimized for compactness and speed in AmigaDOS 2.0.

3.5 The Printer Device (PRT:)

The printer device is intended specifically for output on the printer. If the PRT: device is addressed, it uses the printer driver and selections set by the Preferences editor. By using printer drivers Commodore has attempted to standardized printer output. In this manner all programs use the same printer functions and command characters. When new printers become available, only the printer driver and not the program will have to be re-written. Different printers require different drivers because different printers use different codes for activating the same function. Despite the many printer drivers on the Workbench disk, there are always difficulties with a few printers.

We believe that these methods are easy to use with completed printer drivers. However, it would be much better if there were a program that made it possible to put together custom drivers for any available printer. Please see the Abacus book *Amiga Printers - Inside and Out* for a description of an excellent shareware program that does just that.

Workbench 2.0 implementation:

The device driver has been optimized for compactness and speed in AmigaDOS 2.0.

3.6 The Console Device (CON:)

The CON: device refers to both the keyboard and monitor of the Amiga (i.e., the *console*). Because the keyboard and monitor screen of the Amiga are normal input and output devices, they can be addressed like any other device. Both input and output can take place in any window. The CON: device is accessed as follows:

```
CON:X/Y/WIDTH/HEIGHT/NAME
```

The arguments following CON: have the following meaning:

X/Y	Coordinates of upper left screen corner
WIDTH	Screen width in pixels
HEIGHT	Screen height in pixels
NAME	Window's name

Example 1:

```
DIR >CON:10/10/300/100/Testwindow
```

The directory output appears in a window with the given dimensions. As soon as the output ends, the window disappears again.

Example 2:

```
COPY CON:10/10/300/75/input CON:10/100/300/75/output
```

This displays two CON: windows on the screen at the same time. The input entered in the input window appears in the output window after pressing the <Return> key. Pressing <Ctrl><Backslash> (<Ctrl><\>) removes both windows.

Workbench 2.0 implementation:

The device driver now includes the NEWCON device which allows the enhanced Shell operations in AmigaDOS 2.0.

3.7 The RAW Device (RAW:)

This device is closely associated with the CON: device. At first glance it looks exactly the same. The first difference between the two is established when entering input. The RAW: device doesn't display any characters. The following example is a good demonstration of the function:

```
COPY RAW:10/10/300/75/input CON:10/100/300/75/output
```

Enter any characters in the top RAW: window. All of the characters are transferred to the CON: window without waiting for the <Return> key to be pressed. If it is pressed, the cursor appears at the beginning of the line.

Another nice feature of RAW: is that control characters for cursor movement, <Delete> and <Backspace> can be transferred. The receiving device (CON:) removes these characters when executing them. In our example only the cursor keys function as usual. Pressing <Ctrl><C> ends the entire process.

If the output doesn't function, there is a possibility that input was first entered in the bottom window. As in the AmigaDOS window, the output can be suppressed by other data. In this case, the <Return> key should be pressed in the bottom window.

Workbench 2.0 implementation:

The device driver has been optimized for compactness and speed in AmigaDOS 2.0.

3.8 NEWCON-handler (NEWCON:)

Workbench 1.3 implementation:

The Shell uses this new window interface for output and input in 1.3. The NEWCON: device is similar to the old CON: device from the original CLI. Before it can be used it must be mounted, like all other handler devices, using the MOUNT command:

```
MOUNT NEWCOM:
```

The important entry in the MountList file found in the Devs directory on the Workbench disk should look like the following:

```
NEWCON:     Handler = L:Newcon Handler
            Priority = 5
            StackSize = 1000
#
```

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the device has been made the internal CON: device and does not have to be mounted.

3.9 The RAD device (RAD:)

Workbench 1.3 implementation:

The abbreviation RAD stands for Recoverable rAm Disk. This is a reset-resistant RAM disk for the Amiga. The RAM disk device named `ram:` loses all of its information after a reset. A normal reset does not affect the `RAD:` device. In most cases the data can even survive a Guru Meditation. The `ramdrive.device` is located in the `Devs:` directory.

The `RAD:` device has at least one disadvantage. RAD doesn't have a dynamic memory system. RAD uses the same amount of memory whether it contains any data in it or not. A typical entry for the `RAD:` device in the `MountList` looks like the following:

```
RAD:  Device = ramdrive.device
      Unit = 0
      Flags = 0
      Surfaces = 2
      BlocksPerTrack = 11
      Reserved = 2
      Interleave = 0
      LowCyl = 0
      HighCyl = 21
      Buffers = 5
      BufMemType = 1
#
```

You must specify RAD's capacity in the `HighCyl` parameter before you can mount RAD with `MOUNT RAD:`. Each cylinder has a capacity of 11K. RAD would have a capacity of (21+1) * 11K = 242K if it were mounted using the above entry. In 1.3 RAD must be formatted before it can be used with the `FastFileSystem`.

After a reset, all you have to do is enter `MOUNT RAD:` and the contents of RAD are restored. If you discover that some data is lost, use the `DISKDOCTOR` to restore RAD.

When RAD is no longer needed, the largest section of its memory can be freed by using the `REMRAD` command. It can be removed by using the `ASSIGN` command:

```
ASSIGN RAD: REMOVE
```

The entire memory area that was occupied by RAD is then free after the next boot operation.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the device has been made an internal device and is not located in the DEVS: directory.

3.10 The PIPE Handler (PIPE:)

This device opens a communication channel for data exchange between different tasks. This communication channel consists of a 4K data buffer that can be written to and read at the same time by a task.

Before a PIPE: device can be used, the device must be mounted using the following command:

```
MOUNT PIPE:
```

Any number of communication channels can theoretically be open at once. For this reason the PIPE: device name has a channel name added to it. In the following example the actual contents of the directory are loaded into ED with PIPE: "test":

```
DIR >PIPE:test
ED PIPE:test
```

This was only possible through an intermediate file in Workbench 1.2:

```
DIR >DF0:helpfile
ED DF0:helpfile
```

The output process waits until another process is finished if the channel capacity is not large enough. For example, if the output of the entire directory contents is directed to the Workbench disk using a pipe: device (DIR >PIPE:test opt a), the process waits a while because the buffer cannot take any more characters. In this situation a second Shell can help read out of the PIPE.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the device driver has been optimized.

3.11 The SPEAK handler (SPEAK:)

Workbench 1.3 implementation:

This device controls the Amiga's speech output. The SPEAK: device must be mounted using the MOUNT command, this is usually done in the Startup-sequence.

```
MOUNT SPEAK:
```

Example for speech output:

```
ECHO > SPEAK: "nice to see you"
DIR > SPEAK DF: opt a
TYPE S:Startup-sequence TO SPEAK:
```

It is possible to choose different output modes using OPT. The options must be separated by a slash(/) with no spaces between them or the colon. The following options are available:

p###	Pitch (### is 65-320)
s###	Speed (333 is 30-400)
m	Male voice
f	Female voice
r	Robot voice
n	Natural voice
o0	No option in input stream
o1	options allowed input stream
a0	turn off direct phoneme mode
a1	turn on direct phoneme mode
d0	break sentences on punctuation alone
d1	break sentences on punctuation, RETURN and LINEFEED

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3.2, but the device driver has been optimized.

3.12 The AUX Handler (AUX:)

This handler supports the serial interface of the Amiga. The data is no longer stored in a temporary buffer. The AUX: device must be mounted using the MOUNT command, which is usually done in the Startup-sequence:

```
MOUNT AUX:
```

Multi-user operation can be easily realized with the Amiga (see Chapter 5) by using this device. With a simple NEWSHELL AUX: command an inexpensive used terminal can be attached to the Amiga for all your AmigaDOS work. This frees up the Amiga keyboard and Workbench screen and gives you a true multi-user computer very inexpensively. The transfer parameters are set from the Preferences programs.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3.2, but the device has been optimized.

3.13 The FastFileSystem

Workbench 1.3 implementation:

You'll find this new handler in directory L: of the Workbench 1.3 disk.

Generally, a file system can be viewed as an enlarged handler. It handles the organization of data on the disk differently. A file system also does not access the device directly, but deals with device handlers.

So like the SPEAK handler uses the speak.device, a file system can address the trackdisk.device to read data from a disk drive. A file system is not fixed to any special device. To address a hard disk, make an entry in the MountList and the file system accesses the hard drive.

Until now communication with the connected drives (floppy or hard drive) took place over the file system found in the Kickstart operating system. The new FastFileSystem (abbreviated FFS) was invented for drives whose memory medium does not change (RAD, hard disk). The name reflects the improvement over FileSystem: It is faster than the old file system.

Because a disk change is not allowed, the FFS functions only with a hard drive or the new RAM disk RAD:. Only partitions that are not automatically mounted on the hard drive work with FFS.

To take advantage of the new FFS, the following lines must be entered for the device and partition in the MountList:

```
Globvec = -1
FileSystem = L:FastFileSystem
DosType = 0x444F5301
```

These changes can be made easily using ED. The call for ED looks like this:

```
ed devs:mountlist
```

Then the device can be connected using mount <Devicename>. This command should be entered in the startup sequence following the BINDDRIVERS command.

Before the first access the hard drive or RAD disk must be formatted for the FastFileSystem. The Format command must have the added argument FFS. For example:

```
FORMAT DRIVE DZH1: name Part_1 FFS
```

You only have to format the partitions if the hard drive is already divided into partitions. The entire hard disk must be reformatted if a partition in the `MountList` changes the statement after the `LowCyl` or `HighCyl` parameter.

ADDBUFFERS Special attention should be given to the `ADDBUFFERS` command (see Chapter 2) when using the FFS. In opposition to the old file system, increasing the buffer memory using `ADDBUFFERS` also increases the speed. This increase is especially evident in the output of the directory when using the `DIR` command.

Workbench 2.0 implementation:

The new ROM filing system of 2.0 is based on the FastFileSystem, but has been expanded to support the original filing system for compatibility reasons. Since the new file system is incorporated into ROM, the handler is no longer necessary in the `L:` directory. For floppy diskettes the new FastFileSystem can be created using the FFS option of the format command. Floppy diskettes formatted in the FastFileSystem format will not be accessible to AmigaDOS 1.3 users, the `Not A DOS disk` error message will be displayed and the diskette icon will be labeled DOSA:.

4.
More AmigaDOS Commands

4. More AmigaDOS Commands

So far we have covered the AmigaDOS commands common to Workbench 1.2, 1.3 and 2.0. The Amiga is a very flexible computer system and its operating system is constantly being improved. This chapter will cover the AmigaDOS commands added to Version 1.3 and 2.0. Any differences in the commands will appear after the general description of the command. Each version of the AmigaDOS command will be preceded by **Workbench 1.3 implementation:** or **Workbench 2.0 implementation:**.

Version 1.3

Version 1.3 of the Workbench and Kickstart, featured many added improvements over 1.2. A few of the improvements are:

- New and improved AmigaDOS commands
- A comfortable `Shell` in addition to the `CLI`
- AmigaDOS commands can be loaded and remain resident commands
- A `FastFileSystem` for disk drives without changing storage media (hard disks, RAM disk)
- A faster math library
- New device handlers (`AUX:`, `SPEAK:`, `PIPE:`, `NEWCON:`)
- Reset-resistant RAM disk that boots with Kickstart 1.3
- Booting from special devices (Kickstart 1.3 only)

Much of this list functions with Kickstart 1.2. This is good news especially for Amiga 500 and Amiga 2000 users, because these computers have Kickstart resident in ROM (Read Only Memory).

RAD:

Workbench 1.3 in conjunction with Kickstart 1.2 can boot from special devices. The new RAM disk `rad:` belongs to these devices from which the Amiga can be booted in seconds using Kickstart 1.3. Kickstart 1.2 users can only boot from drive **DF:** (the internal disk drive). There is greater value in the compatibility of the old Workbench and Kickstart versions. If you switch from 1.2 to 1.3 there could be problems with the existing software. You should go back to the old Workbench if you encounter difficulties (for example, we had problems with the debugger `db` of the Aztec compiler).

Version 2.0

Version 2.0 of the Workbench and Kickstart, featured many improvements over 1.3. The main improvements to AmigaDOS were that every AmigaDOS command was rewritten in the C programming language. This optimized each command for compactness and speed. Many of the AmigaDOS commands were made internal commands and no longer have to be loaded from the `C:` directory, greatly improving the Amiga's already astounding performance.

The Amiga designers recognized the flexibility of a system that calls commands from diskette so they built in an internal command override system, keeping the best of both worlds, internal and external commands. Here are the internal commands of AmigaDOS 2.0:

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

The Shell window also has the new Workbench 2.0 improvements. It now contains a close gadget, a zoom gadget and a depth gadget. It also automatically displays as much data as will fit in the window when the window is resized.

AREXX

AREXX is a version of the mainframe computer REXX programming language that has been implemented on the Amiga. AREXX has been integrated in the Workbench 2.0 operating system. AREXX is an application programming language that can be used to extend operating system commands and customize applications program for easy interaction.

4.1 AmigaDOS 1.3 Commands

Some new, very useful commands exist in the C: directory on the Workbench 1.3 disk and internally in AmigaDOS 2.0. We want to examine these new commands more closely. Each version of the AmigaDOS command will be preceded by **Workbench 1.3 implementation:** and **Workbench 2.0 implementation:**.

4.1.1 AVAIL

Workbench 1.3 implementation:

Syntax: `AVAIL [CHIP|FAST|TOTAL]`

This command displays the amount and types of memory available. The Workbench screen title bar displays the amount of free working memory. `AVAIL` displays much more, as the following example illustrates:

Type	Available	In-Use	Maximum	Largest
chip	77472	445760	523232	42712
fast	226200	290688	516888	219008
total	303672	736448	1040120	219008

Descriptions for the chip memory, fast memory (only present in memory expansions) and for the entire memory region (chip memory+ fast memory) appear in each column. The amount of memory not in use is displayed under `Available` and the size of the memory being used is shown under `In-Use`. Both values add up to the value found under `Maximum`.

A program can be loaded into small memory sections but these memory sections must be the smallest allowable size. When a program won't load anymore, even though there is still enough memory, there is a good possibility that the program segments are larger than the largest segment of available memory.

Workbench 2.0 implementation:

Syntax: `AVAIL CHIP/S,FAST/S,TOTAL/S,FLUSH/S`

The argument template now displays correctly when a "?" is input. The FLUSH/S argument attempts to flush available memory as much as possible to recover as much memory as possible.

4.1.2 FF

Workbench 1.3 implementation:

Syntax: `FF -0/S,-N/S, FONTNAME`

This command activates a program named `FastFonts`, developed by Microsmiths®. `FastFonts` accelerates text output on the Amiga screen. The output increases in speed by a maximum of 20 percent.

The user enters `FF -0` to enable `FastFonts` (this command can usually be found in the Startup-sequence of many boot disks). The following message appears on the screen:

```
FastFonts V1.1 Copyright—1987 by C.Heath of Microsmiths, Inc
Turning on FastText
```

The message can be suppressed by redirecting it to the `NIL:` device with:

```
FF >NIL: -0.
```

The `-N` argument can be entered if the normal output mode is required. The starting message will then appear with the `Turning off FastText` message.

FONTNAME can be used to replace the system font with an 8X8 pixel font.

Workbench 2.0 implementation:

The command has been optimized for compactness and speed and made and internal part of AmigaDOS.

4.1.3 LOCK

Workbench 1.3 implementation:

Syntax: `LOCK DRIVE/A,ON/S,OFF/S,PASSKEY`

This command write protects a diskette or any partitions of a hard disk drive. The diskette or hard disk partitions must function under `FastFileSystem` (FFS) from Workbench 1.3. A `LOCK`ed partition behaves exactly like a disk on which the write protect clip is in the write protect position.

The `LOCK` command can also secure the write protect condition using a 4 character password. You can then only remove the write protection when you know the password. The following command sequence protects drive **DH1:** until you unlock the drive using the password open:

```
LOCK DH1: ON open
```

Any attempt at writing to partition **DH1:** is greeted with the message `Volume xxx is write protected` (xxx represents the name of partition **DH1:**).

The following command restores write access to the partition protected by the above `LOCK` command:

```
LOCK DH1: OFF open
```

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3, but the command has been optimized for compactness and speed.

4.1.4 NEWSHELL

Workbench 1.3 implementation:

Syntax: `NEWSHELL WINDOW, FROM`

This command allows you to open another `Shell` window for DOS command entry. The `NEWCLI` command also performs this function.

NEWSHELL can open a window in the size and title specified by the user. The following command creates a window named `Amiga` with a width of 250 pixels and height of 100 pixels, with the upper left corner of the window starting at X-coordinate 50 and Y-coordinate 70:

```
NEWSHELL:50/70/250/100/Amiga
```

This option works best when using the command in conjunction with a Startup-sequence script file.

If the size input is missing, AmigaDOS creates a window the full width and half the height of the normal screen or one quarter of an interlaced screen.

FROM

With the addition of the `FROM` argument and the name of a script file, the `NEWSHELL` command can automatically execute a script file. If the script file is in a drawer the complete pathname must be specified. An example:

```
NEWSHELL FROM S/Copies
```

In this example the script file named `Copies` in the S directory executes, before you can work with the new AmigaDOS window.

A `Shell` has the following advantages over the `CLI`:

- The input line can be edited by using the cursor keys. The cursor can be placed anywhere on the input line by using the <Cursor left> and <Cursor right> keys.
- The `Shell` uses the `NEWCON:` device for input and output. This new window interface is responsible for many of the new `Shell` features. The <Del> and <Backspace> keys function as usual. Additional text is entered from the current cursor position. When the <Return> key is pressed, the `Shell` accepts the entered line.

The following key combinations can be used to edit a command line:

<Ctrl><A> Places the cursor at the beginning of the line (also <Shift><Cursor left>)

<Ctrl><Z> Places the cursor at the end of the line (also <Shift> <Cursor right>)

<Ctrl><K> Erases the text from the cursor to the end of the line

<Ctrl><U> Erases the text to the left of the cursor

<Ctrl><W> Moves the cursor to the next Tab position

<Ctrl><X> Erases the entire line

Recalling previously entered commands

This new feature also comes from the NEWCON: device. Every command entered is stored in a 2K buffer. Pressing the <Cursor up> key restores the last command. This function is very useful when a command is not executed due to a typographical error. With a keypress the command line re-appears; you can quickly correct it with the cursor keys.

If you're looking for a certain command that was entered a short time ago, the NEWCON: device can help. Enter the first letter of the command and press <Shift><Cursor up>. The command that starts with that letter reappears.

Pressing <Shift><Cursor down> moves you to the end of the buffer.

Control code handling

When a control code is entered in the Shell (for example: <Ctrl><L>), the code is displayed in inverse video. However, the control code operates normally.

Startup script file

Every time the NEWSHELL command is called, a script file with the name Shell-Startup automatically executes. This file is found in the S directory of the Workbench disk. Here the appearance of the Shell prompt can be stored. The following Shell functions are only useful when the Shell segment is integrated into the operating system before the Shell is called. The command needed here reads:

```
RESIDENT CLI 1:Shell-Seg System
```

These commands are usually found in the startup sequence of the Workbench 1.3 disk so that you don't have to enter them manually.

Resident commands

Programs can remain in the working memory of the Amiga for use by the Shell with the help of the RESIDENT command. These commands are immediately accessible to the user and do not have to be loaded from the disk drive. More information on this subject can be found under the handling of the RESIDENT command.

Shorter program names

You can give the AmigaDOS commands, found in the C directory of the Workbench disk, other names. Many programs use AmigaDOS commands and they should not be renamed.

The AmigaShell features a function that makes it possible to call a command under any name, or multiple names. This name is specified with the help of the ALIAS command. An example:

```
ALIAS EX EXECUTE
```

After entering this line the EXECUTE command can be called using the name ES. The ALIAS command assigns the first character string the same text as the rest of the line. In this case the rest of the line only consists of the word EXECUTE. The following use of the ALIAS command lets you call the startup sequence into ED by typing St-up:

```
ALIAS St-up ED S:Startup-sequence
```

If you don't want to enter the ALIAS command every time you open a Shell, you can place the ALIAS commands in the S directory in the script file Shell-Startup. As already said, this script file automatically executes with each NEWSHELL.

The ALIAS command without the additional statements lists the existing name assignments.

Workbench 2.0 implementation:

The command has been optimized for compactness and speed. It has also been made an internal command of AmigaDOS. The 2.0 NEWSHELL window also contains a close box, so it can be closed using the mouse. The 2.0 AmigaShell window also dynamically adjusts the contents of the window when it is resized, displaying as much information as possible in the window. The new AmigaDOS windows also have the new Workbench 1.2 zoom gadget and depth gadgets, in place of the front and back gadgets.

4.1.5 REMRAD

Workbench 1.3 implementation:

Syntax: REMRAD

Abbreviation for Remove Recoverable RAM Disk. This command erases the contents of the recoverable ramdrive.device called RAD:. The RAM disk then takes up a relatively small section of memory. When the computer reboots, this memory is returned to the system.

Workbench 2.0 implementation:

Syntax: REMRAD DEVICE,FORCE:

The command has been optimized for compactness and speed, but is larger than the 1.3 version since it contains more options.

DEVICE When you have installed multiple ramdrive.devices, the DEVICE argument allows you to specify which ramdrive.device to remove.

FORCE If the ramdrive.device is in use when you attempt to remove it, a drive in use message will be displayed. The FORCE argument can be used to remove a ramdrive.device currently in use.

4.1.6 RESIDENT

Syntax: RESIDENT NAME,FILE,REMOVE/S,ADD/S,REPLACE/S,PURE/S,SYSTEM/S

This command loads the user's favorite AmigaDOS commands into working memory. AmigaDOS commands or programs previously had to be loaded from disk before they could be used. Because of this you had to leave the Workbench disk in the drive even though the commands would often involve other disks (for example: the DIR command). RESIDENT makes it possible for the user to load his most frequently used commands into working memory. Then the command is in memory and immediately accessible.

Before the RESIDENT command existed, important commands could be copied into the RAM disk and DOS was informed by means of the PATH command to look in the RAM disk before it accessed the Workbench. This method functioned very well except for one large disadvantage: When a command in the RAM disk was called, it still had to be loaded just like from the disk drive. This is a very inefficient use of memory because the command is then present in two memory locations. Each new call of the program copied another command into RAM.

Commands loaded using RESIDENT are loaded into working memory once. When it is called a second time from a second AmigaDOS window, the program is executed from the location in RAM.

An AmigaDOS command must meet some requirements before RESIDENT can properly function:

- The command must be *re-executable*. This means that you must be able to use it from more than one AmigaDOS window. Example: The first AmigaDOS window lists the directory of drive **DF0:** while the DIR command is being used in the second AmigaDOS window for drive **DF1:**. Most AmigaDOS commands, with very few exceptions, are "re-executable" on the Amiga.

- The commands must be *re-entrant*. As described above, the program code of a RESIDENT command can only be found in one location when the command is executed in several places at the same time. The feature that makes a re-entrant command so good is the use of local variables that must be replaced with every call of the program.

To understand this problem we'll describe an example. Suppose you execute a drawing program that contains the color code for the current

text color in a memory location. This memory location gives the code for the color white directly after loading the program. Change this color to red and restart the program. The second program now has the color red as the default instead of white because they use the same memory location. This is a harmless example. The Amiga does not differentiate between the two and when the second is called, AmigaDOS 1.3 responded with a system crash (Guru meditation).

PURE A crash could have been prevented if the P (Pure) status flag had been set using the PURE argument. This flag, with the help of the PROTECT command, can be set or erased for each file. A Pure flag tells AmigaDOS that the program can be made resident using the RESIDENT command. LIST the AmigaDOS commands and you'll see that it makes sense to have the Pure flag set for many commands.

Now we come to the use of the RESIDENT command. When the command is entered without arguments, a list of the present RESIDENT commands appear. When the command is entered with the SYSTEM argument the resident system segments are also shown. For example:

```
>1 RESIDENT SYSTEM
Name                UseCount

Cd                         0
List                       0
Resident                   1
Execute                    0
CLI                   System
Filehandler           System
Restart               System
CLI                   System
```

UseCount supplies information about how active the respective command is at the time. This statement usually returns a 0. A 1 means that the command is being used at the time. System segments are listed as System.

NAME/FILE The NAME and FILE arguments specify the exact path of the command or segment that should become resident. The following example places the DIR command in the Shell:

```
RESIDENTR C:DIR
```

When you use the RESIDENT command in a file where the Pure flag is unset, the following error message appears:

```
Pure bit not set
Cannot load xxx
(xxx stands for the filename)
```

When Pure is unset, a file can still be loaded using RESIDENT by adding Pure. The message Pure bit not set is displayed in this case. The PURE argument should be used with caution because programs where the Pure flag is not set are not usually re-entrant.

REMOVE The REMOVE argument eliminates an entry from the list of resident files. The following example removes the EXECUTE command from the list:

```
RESIDENT EXECUTE REMOVE
```

The UseCount value of a system segment is set at -1. Because an entry can only be removed when UseCount is at zero, a segment cannot be erased using REMOVE.

ADD The ADD argument makes it possible to make more commands or segments resident.

REPLACE The REPLACE argument replaces any command (or segment) with a command (or segment) already in the list. Because an entry can only be removed when UseCount is at zero, an active command cannot be replaced. For example, if the EXECUTE command was resident, entering the following would replace it with the DATE command:

```
RESIDENT EXECUTE DATE REPLACE
```

Workbench 2.0 implementation:

Syntax: RESIDENT NAME,FILE,REMOVE/S,ADD/S,REPLACE/S,PURE=FORCE/S, SYSTEM/S

AmigaDOS Version 2.0 was rewritten for compactness and speed, and many of the commands have been made internal commands. Entering RESIDENT SYSTEM will display a list of resident commands, internal commands and system segments.

To keep the AmigaDOS system as flexible as possible the internal commands can be controlled with the REMOVE argument. You can remove internal commands using the REMOVE argument and ADD your own commands. The AmigaDOS internal commands can be activated again with the REPLACE argument.

The PURE argument can now also be called with FORCE.

The following is a list of the AmigaDOS 2.0 internal commands:

```
1. SYS:> RESIDENT SYSTEM
NAME                USE COUNT
Assign                 0
List                   0
Execute                0
Alias               INTERNAL
Ask                 INTERNAL
CD                  INTERNAL
Echo                INTERNAL
Else                INTERNAL
EndCLI              INTERNAL
EndIf               INTERNAL
EndShell            INTERNAL
EndSkip             INTERNAL
Failat              INTERNAL
Fault               INTERNAL
Get                 INTERNAL
Getenv              INTERNAL
If                  INTERNAL
Lab                 INTERNAL
NewCLI              INTERNAL
NewShell            INTERNAL
Path                INTERNAL
Prompt              INTERNAL
Quit                INTERNAL
Resident            INTERNAL
Run                 INTERNAL
Set                 INTERNAL
Setenv              INTERNAL
Skip                INTERNAL
Stack               INTERNAL
Unalias             INTERNAL
Unset               INTERNAL
Unsetenv            INTERNAL
Why                 INTERNAL
FileHandler         SYSTEM
Restart             SYSTEM
Shell               SYSTEM
CLI                 SYSTEM
1. SYS:>
```

4.1.7 SETPATCH

Workbench 1.3 implementation:

Syntax: SETPATCH R

This command is found in the startup sequence on Workbench diskettes. SETPATCH is used to patch or add updates to the operating system. New versions of SETPATCH will be made available as the AmigaDOS system is improved.

In Workbench 1.3 it modified the Kernal so that a Guru Meditation does not follow a Recoverable Alert. The R argument in 1.3 allows the recoverable ramdrive.device to work with the new 1 MEG chip ram machines, the 68010 and allow RUN to be used from the resident list.

Workbench 2.0 implementation:

Syntax: SETPATCH

The new ROMS have been corrected so the R option is no longer necessary. New versions of SETPATCH will be made available as the AmigaDOS system is improved.

4.1.8 SETENV/GETENV

Syntax:
```
SETENV NAME/A,String
GETENV NAME/A
```

These commands allow the use of environment variables. Environment variable are stored in the ENV:, which in 1.3 is the RAM: disk. To remove a definition use SETENV NAME, the variable will remain in ENV:, but it will be empty.

Workbench 2.0 implementation:

Syntax:
```
SETENV NAME/A,String
GETENV NAME/A
```

Entering SETENV without parameters will display the current environment variables. The SETENV and GETENV commands have been optimized for compactness and speed. The commands are now also

internal AmigaDOS commands. LOCAL and GLOBAL environmental variable may be implemented in the next release of AmigaDOS.

4.1.9 ICONX

Workbench 1.3 implementation:

This command allows you to call a script file from the Workbench by double clicking on it with the mouse. The following must be done beforehand:

- Create a Project icon for the script file with the help of the icon editor on the Extras disk. The `SHELL` icon is loaded into the editor from the Workbench, modified and then saved under the name of the script file.

- Open the disk drawer with the new icon, click on the icon and choose the `Information` item from the `Icon` menu (in 1.3 the `Info` menu item from the `Workbench` menu.) The `SYS:C:ICONX` command must be entered in the `Default Tool` gadget. Save the `Information` window. The script file can now be called by double clicking on the icon. Descriptions about the window size for the output of the script files can be made in the `Tool Types` string gadget in the `Information` window. For example:

```
TOOL TYPE WINDOW=CON:0/0/400/100/Script window
```

The window can stay open after processing the script file by entering the following:

```
TOOL TYPE DELAY=1000
```

Delay time must be given in 1/60 seconds.

You can use the extended selection capabilities of the Workbench to pass the Workbench file to the script file. The selected files appear to the script file as keywords, therefore the `.key` keyword must be on the first line of the script file.

Workbench 2.0 implementation:

Operation is identical to Workbench 1.3 but the command has been optimized for compactness and speed.

4.2 AmigaDOS 2.0 Commands

Some new, very useful commands have been added to AmigaDOS 2.0 internally. This section will explain these new commands.

4.2.1 MAKELINK

Syntax: `MAKELINK [FROM] <name> TO <name2> [HARD]`

Creates a file that points to another file. When the first file is specified, the linked file is called. The default is "soft-linked" files, which means the linked file can be on another diskette.

FROM <name>	The name of the original file.
TO <name2>	The name of the linked file.
HARD	Files will not be linked across volumes.

4.2.2 UNALIAS

Syntax: `UNALIAS NAME`

Removes an alias from the alias list.

NAME	The name of the alias to remove.

`V2.0` AmigaDOS 2.0 internal command.

4.2.3 UNSET/UNSETENV

Syntax: `UNSET NAME`
`UNSETENV NAME`

Unset an environmental variable.

NAME The name of the variable to unset.

V2.0 AmigaDOS 2.0 internal command.

5. AmigaDOS Tricks and Tips

5. AmigaDOS Tricks and Tips

The purpose of this chapter is to solve some of the problems that you may sooner or later encounter while you work with AmigaDOS. Some of these items have been known in the Amiga community for a long time; others are "hot off the press."

In addition to these tricks and tips, you'll also find plenty of additional information and advice about AmigaDOS.

5.1 Input and Output in AmigaDOS

You have probably wished that you could just press a key in AmigaDOS and have the output go the printer. What exactly starts the output? If you said, "Pressing the <Return> or <Enter> key", you'd be half right. There's more to it than pressing a key. You can also execute the printout by pressing the <Backspace> (←) key.

Running output in the background can almost act as a completely different task. It's possible to enter a new command while an old command is still working. Not all output can be carried on simultaneously, so the new command waits until the old command is finished executing. This has nothing to do with the multitasking capability of the Amiga, since DOS commands are always executed one after another. In spite of this, work can be done somewhat faster because DOS is active through all of this. An example:

```
COPY text TO duplicate
DELETE text
```

If for some reason the copy operation is uncompleted (e.g., the disk is full), the data file `text` is lost. Now there is neither a copy nor the original. A tip: The `DiskDoctor` can sometimes help in such a situation. It's assumed that no writing has taken place to the blocks of the data file. This should be checked out in any case.

5.2 Wildcards

MS-DOS users who upgraded to an Amiga miss the old * and ? wildcard characters. These characters made it possible to enter a shorter filename on the older generation computers (e.g., IBM, VIC20, C64). If the user wanted to erase the files named `test` (`test1`, `test2`, `test3`, etc.) from a disk, the `DEL` command used in conjunction with the name `test*` deleted all these files, and any other files beginning with the letters "test". If the user wanted to deal only with those data files whose names are five letters long, the question mark could be used (e.g., `DEL "test?"`). A file named `test1version` in this case would remain untouched. The two wildcards can be combined. For example:

```
"????version*"
```

In this case all the files with the word "version" in them starting at the fifth position and having any letters after that are addressed. The following data filenames would fulfill the requirements of this example:

```
TestVersion1
LastVersionOfToday
FourVersion
```

Those new to computers can get the idea of wildcards from these examples.

The number sign (#) can be used in place of the asterisk on the Amiga. The question mark looks for an exact character position in the filename. The # sign is as flexible as the asterisk. As you may know, a number sign at the beginning of a filename sometimes means nothing more than a word or number. Sometimes a numeric value is also expected behind it. For example, the `DIR` command can be enlarged by adding `Test#3a` so that it searches only for those filenames that start with Test and end in three A's. This function is not patterned after the asterisk. If, instead of the number, you entered a question mark, the characters following the question mark would be ignored. The number sign combined with the question mark becomes the equivalent of the asterisk wildcard. For example, the following input displays all files in the directory ending with `.info`:

```
DIR #?.info
```

A few examples follow. Their purpose is to show the use of the wildcard options.

Filenames:

Cat.1	1.Dog
Cat.2	2.Dog
Catnip	Doggy
Pussycat	Wiener_Dog

DIR ***versions:***

1. Cat.?
2. C#?
3. ??????????
4. #?
5. Dog?????
6. #?Dog
7. ??#2s#?

The following would be the results if each of the above `DIR` commands were entered:

1: Cat.1, Cat.2
2: Cat.1, Cat.2, Catnip
3: Wiener_Dog
4: all files
5: no files
6: 1.Dog, 2.Dog, Wiener_Dog
7: Pussycat

The last example clearly shows that many combinations are possible as long as they make sense.

5.3 Breaking in AmigaDOS

Keyboard breaks

The Amiga doesn't have a <Run/Stop> key like the C64 or a <Break> key like an IBM. But it's possible to stop the execution of an AmigaDOS command by pressing <Ctrl><C>. All AmigaDOS commands react by returning to the program from which they were called (in most cases, AmigaDOS itself). As a reminder of ending before the command was finished, the message `*** Break` appears. The three asterisks indicate an interruption of type C. From this type of interruption comes <Ctrl><D> through <Ctrl><F>. Each <Ctrl> break has its own advantages. <Ctrl><D> works only in conjunction with the SEARCH and EXECUTE commands.

The SEARCH command, which can search entire directories for a given character string, reacts to <Ctrl><C> like every other command: The command stops. If <Ctrl><D> is used, the file being searched at the time is dropped and the next file searched. The <Ctrl><D> command is a less suitable break command than <Ctrl><C>.

It is somewhat more difficult with the EXECUTE command. <Ctrl><D> stops execution of script files made up of AmigaDOS commands (see Chapter 6 for more details on script files).

Basically, <Ctrl><C> has a higher priority than <Ctrl><D> for the execution of a `AmigaDOS command` within a script file. The `AmigaDOS command` that is currently running ends, and control returns to the DOS level. When it leaves the command, it leaves an error code (error number >= 10) when it returns to EXECUTE, so that the script file is left under the output of the message (example: <Ctrl><C> for carrying out the SEARCH command). If the command returns to the DOS level without a message, the script file works as normal (Example: <Ctrl><C> for carrying out the `Dir` command).

<Ctrl><D> has a special function for the EXECUTE command. By entering this key combination, the active `AmigaDOS command` of the script file finishes its work and then execution of the text file is stopped. This can cause problems when the AmigaDOS command itself reacts to <Ctrl><D>:

The SEARCH command responds to <Ctrl><D>. If <Ctrl><D> is used while this command is working through a script file, the SEARCH command reacts and not EXECUTE. After the SEARCH operation ends, the script file isn't left unless there are no more commands.

Command breaks

If you start a command using RUN, an independent task starts. No more input can be sent to it from the original AmigaDOS window. This is treated as a "non-interactive process". Only the output is shown on the screen. A sample:

```
RUN DIR DF0:
```

After the input of the new number that is given for this process (for example [2]), the main directory of the disk is displayed in the AmigaDOS window. The command no longer reacts to <Ctrl><C>.

There is a way to exit a non-interactive process. The AmigaDOS BREAK command acts like the <Ctrl> key. The command waits for the number to be stopped and the associated letter of the branch interrupted (c-f). If all of the interrupt calls should be used, the ALL argument can be used to do this.

For interrupting the directory output of the above example the command must read:

```
BREAK 2 C
```

The BREAK command can stop operation of a command that was entered in a different AmigaDOS window than the one that is presently active. The number of the process that corresponds to AmigaDOS window must simply be entered.

5.4 The RAM Disk and AmigaDOS

This section is for the user who has only one disk drive. Difficulties frequently arise with this minimal configuration. These can be eased with the help of the RAM disk. The fortunate user can buy more floppy drives but the addition of a RAM disk can be much more rewarding.

Single drive copies

Copying a single data file or an entire drawer using two disk drives is no problem. The Copy command can read:

```
COPY DF0:Utilities/Notepad DF1:Helpprogram
```

How do you copy with only one drive? One option is to give the name of the disk instead of the drive number. For example:

```
COPY FROM BeckerText:Letters/Peter TO Text:Letters
```

DOS automatically alternates between these disks and ignores the drive number. This method has two disadvantages:

1. The name of the two disks must always be known
2. Many disk swaps must be made, even for a short file

The best method uses the RAM disk for storing the file before it goes to the destination disk. First the disk that contains the file to be copied is placed in the drive. The desired file is copied to the RAM disk using the COPY command. Then the destination disk is placed in the drive and the desired file is copied onto it from the RAM disk. The following process copies the entire C: directory to another disk:

```
1.                      (insert source disk)
2. COPY DF0:C RAM:      (copy the directory contents to RAM disk)
3.                      (insert destination disk)
4. MAKEDIR DF0:C        (create directory named "c")
5. COPY RAM: DF0:C      (copy directory contents to destination disk)
```

The operation can only function when the COPY and MAKEDIR commands have been made resident with the RESIDENT command or have been copied to the RAM disk and the system was informed using Assign or Path. The two commands must be available when the system disk is not in the drive.

After the copy operation, the data in the RAM disk should be erased so the memory can be returned to the system. The following command erases the entire C directory that was placed in the RAM disk.

```
DELETE RAM:C ALL
```

5.5 Printing from AmigaDOS

Those of you that do a lot of printing from the Workbench should get your money's worth from this section. This section deals with the basics of printing from AmigaDOS, the problems that can occur and some solutions.

5.5.1 File printout with COPY

COPY enables the duplication of an entire directory or a single data file. The argument template reads:

```
COPY FROM,TO/A,ALL/S,QUIET/S
```

A quick recap: The FROM and TO arguments specify the source and destination of the operation; the ALL argument copies all files from the FROM directory. QUIET performs a "quiet" execution of the command (it suppresses the output of the command).

COPY can access any connected device—it isn't limited to the floppy or hard disk. The possible data flow directions depend on each device. For example, the printer cannot be used as a data source; it can only be used as a destination device. A disk drive can be used for reading and writing.

Enter the ASSIGN command without arguments, and look under the Devices heading:

```
DF0 DF1 PRT PAR SER RAW CON RAM
```

We are only interested in the printer (PRT:) device for now.

The following command gives you a printout of a text file:

```
COPY name_of_file TO PRT:
```

Which output format should you use? That depends on the parameters set in the Preferences editors. Also, if a serial data transfer is desired, Preferences handles the printer as a serial device instead of a parallel device.

The printer device is easy to setup using `Preferences`. A program that sends data to the printer needs no more provisions than an interface type, printing width and paper length. The command characters for the printer are inserted so the data can be interpreted. For special uses this can be avoided by using the rough drivers `PAR:` and `SER:`. Data goes directly to the peripheral.

5.5.2 Redirecting output

Output can be sent to any device using DOS commands. The screen is usually the default device. There's no reason why you couldn't change that output device. The following example would send the directory of the current disk out the serial port (you could transmit your directory through a modem if you want):

```
DIR > SER:
```

All devices that can receive data can be replaced by the `PRT:` (printer) device. Simply enter the command, a space, a greater than sign, a space and the device name (`PRT:`). The following example prints the current main directory:

```
DIR > PRT:
```

The following example prints all the directories on the current disk:

```
DIR > PRT: opt a
```

The following example prints all the directories on the RAM disk:

```
DIR > PRT: ram: opt a
```

The following example prints the text `Hello`:

```
ECHO > PRT: "Hello"
```

The following example prints the startup sequence:

```
TYOE >PRT: DF0:S/Startup-sequence
```

5.5.3 Printer control characters

The printer drivers in `Preferences` have a few things missing. You can't easily use foreign character sets or double-strike mode. You can get around this in AmigaBASIC using the `CHR$` command (see your AmigaBASIC manual or Abacus' ***AmigaBASIC Inside and Out***). AmigaDOS has no equivalent of `CHR$`, so another way must be found.

The basic problem with control codes is that they can't be accessed directly from the keyboard. A few can be accessed by pressing <Ctrl> and another key. For example, control code 15 enables condensed mode on most Epson printers. The user can access this by pressing <Ctrl> <O>. Every control character has a corresponding letter. But be careful: The O is really an uppercase O and not a lowercase o (remember to press the <Shift> key). You cannot see control codes on the screen when they are entered, but that doesn't matter. The important thing is that the printer understands them. The user must direct the output to the printer.

The use of the `ECHO` command is applied here. It is used for special output of text. The following example shows how `ECHO` is used in conjunction with a control code (note the brackets):

```
ECHO > PRT: [Ctrl and O]
```

The brackets mean that you should enter the key combination, not the text. The space before <Ctrl><O> is required.

After the following command is input all text that is sent to the printer will be printed in condensed mode.

```
ECHO > PRT: [Ctrl and T]
```

It's difficult to interrupt an `AmigaDOS command` by pressing <Ctrl><C>. A trick here is helpful. Using BASIC, create a file on the disk that contains the command <Ctrl><C> in the form CHR$(3) (A=1, B=2, C=3). The following program shows you how:

```
OPEN "DF0:CTRL-C" FOR OUTPUT AS #1
PRINT#1, CHR$(3);
CLOSE#1
```

Run the BASIC program and return to AmigaDOS. Now <Ctrl><C> can be sent by using one of these two commands:

```
COPY FROM DF0:CTRL-C TO PRT:

TYPE >PRT: DF0:CTRL-C
```

Next is a multiple-number *escape sequence*. This covers all of the printer control codes that can be accessed through the <Esc> character. At least one <Ctrl> character follows each <Esc> command. The <Esc> key can be found in the upper left hand corner of the keyboard. A typical Epson command entered in BASIC serves as an example. The CHR$(9) enables the Amiga's Norwegian character set:

```
PRINT#1,CHR$(27);"R";CHR$(9);
```

The AmigaDOS equivalent looks like this:

```
ECHO >PRT: [ESC]R[Ctrl I]
```

A space must follow the colon, but there cannot be one before or after the R.

Of course some difficulty can arise when you work with other languages. C programmers use brackets [] and braces {}. They can be accessed by using the American character set if you enter a letter in the alphabet before "A": CHR$(0) is used. Again, a BASIC program that has the desired sequence in a file is of some help:

```
OPEN "DF0:USA-Set" FOR OUTPUT AS #1
PRINT#1,CHR$(27);"R";CHR$(0);
CLOSE #1
```

Run it and copy the file to your AmigaDOS work disk. The following line activates the new character set:

```
COPY USA-Set PRT:
```

5.6 Using the Console Device

The Console device permits a few added features in AmigaDOS. The following section should give a few experiments with this device.

Basically, all output created from AmigaDOS commands from the current window can be directed to any device. It is sometimes useful to direct output to a specified window. For example, if you want to see the main directory of a disk then return to AmigaDOS, you don't have to open a new window using the NEWSHELL command. The following command displays the main directory of the disk in drive zero in a specified window:

```
RUN DIR > CON:10/10/400/100/main DF0:
```

You can stop the display at any time by clicking on the window and pressing any key. The output continues if the <Return> key is pressed. The window automatically disappears after the command finishes executing.

Printer/ typewriter

Nothing is easier from AmigaDOS than turning a connected printer into a typewriter. The following command is all it takes:

```
COPY * TO PRT:
```

The asterisk, which represents the current AmigaDOS window, causes all input to be sent to the printer after the <Return> key is pressed. An advantage over a normal typewriter is that you can correct the line of text before pressing the <Return> key. The entire line can be erased by pressing <Ctrl><X>. Pressing <Ctrl><\> returns the user to AmigaDOS prompt.

Creating text files

There are times when you may need a text file in a hurry. To quickly create a small text file, the following will suffice:

```
COPY * to DF0:text
```

The difference between this command and the one in the previous section is that this command sends the characters to the disk drive instead of the printer. There they are placed in a data file under the name `text`. You can edit these files later using ED or `Edit`. This function can also be ended by entering <Ctrl><\>.

Appending text files

If you want to append one text file to another, use the following command sequence:

```
JOIN CON:10/10/400/100/Input DF0:textdatafile AS DF0:newdata
```

The JOIN command merges data files together. In this case, the first data file is in a CON window and the second file is a text file that already exists on the disk. The result is stored under the name Newdata on drive DF0:.

After entering the command, a window of the given dimensions appear. The entry that is going to start the text file can be entered here. Pressing <Ctrl><\> concludes the input and stores the resulting file on the disk under the given name.

Naturally the input can be added to the end of the text file. The first two parameters after the JOIN command must be exchanged:

```
JOIN DF0:textdatafile con:10/10/400/100/Input AS DF0:newdata
```

AmigaDOS alarm clock

You may find yourself losing track of time during these AmigaDOS sessions. There's the clock on the Workbench disk that can be programmed to sound an alarm. But it has a few disadvantages. First, it takes a long time to load. Second, it requires too much memory. Third, only absolute alarm times can be programmed. Finally, an alarm time must be preset.

The following lines create a low-budget alarm clock, don't forget the plus (+) sign in the first line:

```
RUN WAIT 10 MIN + (Return)
ECHO "Hey, the coffee's done !" (Return)
```

Unfortunately the output always appears in the window from which the process was started. To see the clock you should remember that this AmigaDOS window can never cover another window.

You can make the screen flash when your time is up. To do this, the command code must be enlarged by adding <Ctrl><g>. The revised code looks like this:

```
RUN WAIT 10 min + (Return)
ECHO [Ctrl g]"Hey, the coffee's done !" (Return)
```

The WAIT command makes the alarm go off at an actual clock time (see Chapter 3 for details of Wait). You may wish to create a text file with the alarm message and then TYPE the text file TO the SPEAK: device for an audible alarm!

Because AmigaDOS clock is a separate task, you can set more than one alarm clock at a time. If you open the maximum number of tasks (20), AmigaDOS slows down.

Keyboard/ ASCII conversion

Knowing the ASCII codes of all of the keyboard characters on the Amiga can be helpful when programming. Before you buy a book to look this up, enter the following line from AmigaDOS:

```
TYPE CON:300/10/150/50/Converter YO * OPT H
```

A small window appears in the upper right hand corner of the screen, into which you enter the characters for which you need to know the hex ASCII codes. The input must always be 16 characters. For example, if you want to find the hex ASCII code for the letters S and J, enter the two characters and press the <Return> key 14 times. The result appears in AmigaDOS window. The letters have the hex ASCII codes $73 and $6A, respectively. The rest of the places are occupied by the characters representing the <Return> key ($0A). The statements are always in hexadecimal format. This function can be stopped using <Ctrl><\>.

5.7 Using the Serial Device

Everyone knows about the multitasking capabilities of the Amiga. It is known as a true multi-user system. This enables multiple terminals to be connected to one central unit. Such a terminal can be any screen and keyboard that sends the entered characters to the central unit, and receives the information that the central unit gives. The task of the central unit is, in the case of the Amiga, to work on more than one program at a time.

The function of a terminal can theoretically be taken over by any small computer (e.g., Model 100, C64, Atari ST, etc.) interfaced to the central unit with the correct connection (null modem cable). Because the standard unexpanded Amiga can only address one serial connection, additional terminals can only be added by using extensive hardware and software.

We've tried this. Our configuration used a IBM 386 PC compatible as the terminal, and an Amiga 2000 with a hard drive and two floppy disk drives as the central unit. We used a null modem cable; you could make the connecting cable yourself. Each end had a DB-25 connector, joined to a 7-wire cable. The length of the cable is relatively unimportant because a serial transfer is not very susceptible to trouble. The connection looked like the following:

Pin with	Pin	Function:
2	3	TXD->RXD (sender - receiver)
3	2	RXD<-TXD (receiver - sender)
4	5	RTS->CTS (handshake one direction)
5	4	CTS<-RTS (handshake other direction)
6	20	DTR->DSR (function control)
20	6	DSR<-DTR (function control)
7	7	GND (ground)

Notice the completely symmetrical pinout. It doesn't make any difference which end is connected to which computer. When buying the DB-25 you should pay attention to the case: Some of the gray DB plugs fit poorly in the Amiga. The connectors that bolt into the Amiga connector are better yet (and naturally a little more expensive). These can be used to connect an Atari ST computer as a terminal as well. Many inexpensive display terminals are readily available. Any of these could be hooked up to your Amiga for all your AmigaDOS needs, freeing up your Amiga keyboard and Workbench for serious Amiga graphics work.

Our cable supported 3 wire handshake (xON, xOFF) and also the RTS/CTS handshake.

Attention: Make sure both computers are turned off before connecting the two. The 8520 chip and the RS-232 system will continue to work as long as you follow this rule.

For our first try in AmigaDOS we entered and saved the following parameters in the Amiga computer with the help of the `Preferences` programs. The communication program running on the PC was also set to these parameters:

Baud rate	9600
Buffer Size	512
Read Bits	8
Write Bits	8
Stop Bits	2
Parity	None
Handshaking	RTS/CTS

After making sure the AUX: device was mounted (ASSIGN DEVICES), we then entered the following command to activate the remote terminal:

```
NEWSHELL AUX:
```

We now had a true multi-user system for almost no cost.

Note: The following only works on Amiga's running Workbench/KickStart 1.3, Workbench 2.0 added other more powerful methods of programming debugging. We have one more use for the cable. Change the command used to load the Workbench to read:

```
loadwb -debug
```

After starting the Workbench everything looks normal. Press the right mouse button and move the pointer around on the menu bar. Suddenly an untitled menu appears, containing two items: `Debug` and `Flushlibs`.

It's now possible to display a debugging message on a second computer through the ROM-Wack (a kind of diagnosis program), in case a Task Held or Guru Meditation occurs.

If the computer hangs up and the right mouse button is pressed (press the right button to cancel/debug), the memory of the Amiga computer can be examined with a terminal program on another connected computer. The `Debug` item enables the ROM-Wack if no interruption takes place. The `Flushlibs` item normally doesn't do anything visible (we couldn't figure out the purpose of this item). Now for an example:

The Amiga is connected to another computer by an RS-232 null modem cable. The following lines create a Task Held condition in an Amiga running Workbench 1.3:

```
mount res0:
DIR > res0:
```

The `res0:` device is found in the data file `MountList` of the `devs` directory. If this device is integrated into the Amiga and output is sent to it, the `Task Held` condition usually appears.

If the right mouse key is pressed, a Guru Meditation appears. Pressing the right mouse button again causes the following output on the computer connected as a terminal:

```
rom-wack
PC: FE66E8 SR: 0015 USP: 022B32 SSP: 07FFF2 XCPT: 0003 TASK: 0221A0
DR: 0000FFFF 0000FFFF 0001C815 00000001 00000000 00022394 000221FC 00000019
AR: 000231B5 ABABABAB 00023090 00022394 00FF4834 00005D4A 000045B8
SF: 2271 ABAB ABDD 2269 0015 00FE 66E8 0000 0000 0000 0676 00FC 07DC 00FC 07DE
```

Assembler experts can analyze this statement and can send command via ROM-Wack to the interrupted computer. Pressing a question mark on the computer connected as a terminal shows a list of the commands that the ROM-Wack recognizes:

```
alter boot clear fill find go is limit list regs reset resume set show user
```

Here are their important commands:

`Alter`	changes the contents of the memory
`Boot`	the interrupted computer is re-booted
`Clear`	memory area is cleared
`Fill`	memory area is filled
`Find`	memory area searches for a certain hex value
`Go`	starts program
`Regs`	shows the contents of the register
`Reset`	(behaves like "boot")
`Resume`	If Debug was chosen from the Workbench, the Workbench is activated again

The address area is changed so that a new address can simply be entered (without each command).

We dealt with the `Task` structure that begins at $0221A0 in the example (see register notices under `Task`). The command simply reads:

```
"0221a0"
```

This gave the following message:

```
0221A0 0000 0810 0000 080C 0D0A 0002 2BB6 0002 ....^H^P....^H^L^M^J..^B +....^B
```

With the help of the Abacus ***Amiga System Programmer's Guide***, we pushed the address where the pointer for the name of the interrupt task was contained. It is the address "022BB6" from the above line. Here it reads:

```
022BB6 5245 5330 0000 0000 0000 0002 3338 0000 R E S O.............. 3  8....
```

It was really the task `Res0` that caused the Amiga to hang up. It is nice to know why a program causes a Guru Meditation. In a Task Held condition, the memory of the Amiga can be examined by a monitor program that was started from a second `CLI` process.

The interesting ROM-Wack function clearly shows that the RS-232 connection can function exactly as it is expected. Data transfer at 9600 baud functions flawlessly.

6.
Script Files

6. Script Files

All computers that have forms of DOS (Disk Operating Systems) have some form of script file processing capability. AmigaDOS is no exception. Script files are similar to batch files on MS-DOS computers.

This chapter shows you what can be done with this technique and how it is done on the Amiga. In addition, you'll find a number of practical uses for script files that you might not have thought of before.

6.1 Introduction to Script File Processing

The following sections will acquaint you with script file processing on the Amiga. We'll see at the end of this introduction if you understand more or less about this subject.

6.1.1 What are script files ?

Basically, AmigaDOS commands make up their own small programming language. For example, the `ECHO` command can be compared to the AmigaBASIC `PRINT` statement. The only problem is that AmigaDOS commands can only be entered in direct mode; AmigaDOS has no program mode like BASIC. The command executes after pressing the <Return> key. The most important feature of a programming language is missing: The commands cannot be stored.

Script files are text files that can be created in a word processing program or text editor. These files consist of a succession of AmigaDOS commands. It doesn't matter whether you create a script file using `ED`, BeckerText, the Notepad or whatever, just as long as the text is saved in ASCII format. The file can be saved under any name.

6.1.2 What script files look like

The simplest script files consist entirely of AmigaDOS commands, like the ones that you enter in the `Shell`. Each line can only contain one command. Comments are allowed; they must be separated from the command by a semicolon, and the length is limited to the length of the respective line.

A very simple script file can look like this:

```
COPY test#? testprogram ; copies all test versions into
DELETE test#?           ; a special drawer and erases
                        ; them from the main directory
```

Besides the "normal" AmigaDOS commands allowed in script files, a list of special commands exist, whose use in AmigaDOS windows wouldn't really make sense. They are the commands:

```
ECHO
FAILAT
QUIT
IF/ELSE/ENDIF
SKIP/LAB
ASK
WAIT
```

If the commands `IF`, `ELSE` and `ENDIF` did not exist, a script file would have to be executed from top to bottom, without any potential for branching. The additional commands make it possible to change the program flow. A detailed description of these commands can be found in Section 2.3.

6.1.3 Calling script files

The `EXECUTE` command runs the script files (see Section 2.3 for details of this command). In the simplest case, you would enter `EXECUTE` then the name of the script file you want to run. The following example runs the `Myscript` file on drive **DF0:**, if the file is available:

```
EXECUTE DF0:myscript
```

After `Myscript` executes AmigaDOS is ready to execute new commands.

It's very practical to use a background task for running script files. For this, the script file can be called using the `RUN` command:

```
RUN EEXECUTE DF0:Myscript
```

The operating system separates `EXECUTE` command sequences so that while the script file is running, further work can be done in the current AmigaDOS window. Eventually, output appears in the AmigaDOS window because a background task that is not interactive does not have its own output window.

6.1.4 A simple example

To close this short introduction, we want to go through a simple example step by step to show how a script file is created and configured.

In most cases, a script file is built using ED. This program uses the normal AmigaDOS commands and makes it easy to work on short texts. This is the perfect tool for script files. It is also possible to work with TextPro, BeckerText, Notepad or any other word processor that saves its text as ASCII data.

Assignment: Write a script file that lists many of the typical AmigaDOS script file commands on the screen. The file should contain the names of the commands.

Solution: You must open an AmigaDOS window. ED must be called using the following syntax:

```
ED Commands
```

After a while the empty input window of ED appears. Because no file with this name exists, the message `Creating new file` appears in the lower left corner of the screen. Now the required commands can be entered:

```
;Commands
ECHO "EXECUTE"
ECHO "ECHO"
ECHO "FAILAT"
ECHO "QUIT"
ECHO "IF/ELSE/ENDIF"
ECHO "SKIP/LAB"
ECHO "ASK"
ECHO "WAIT"
```

Press the <Esc> key then the <x> key to save the file under the given name in the actual directory and exit ED.

Now you can execute this script file using the EXECUTE command. Enter the following:

```
EXECUTE Commands
```

The result that appears in the CLI window looks like this:

```
EXECUTE
ECHO
FAILAT
```

```
QUIT
IF/ELSE/ENDIF
SKIP/LAB
ASK
WAIT
```

See Section 2.4.1 for detailed information about using ED.

6.2 Modifying the Startup-sequence

You now know what a script file is and how to use it. For many who heard about the concept for the first time in Section 6.1, you may not have known that your Amiga executes a script file every time you turn it on.

Before we explain this, we would like you to make a copy of your original Workbench disk. When the Workbench disk is mentioned in later sections, you should be using your backup copy, not the original. Store the original copy in a safe place.

Open an AmigaDOS window and place the Workbench disk in drive **DF0:**. Enter the `DIR DF0:S` command. A file named `Startup-sequence` is found in this directory. Before this file is displayed it is a good idea to make the AmigaDOS window as large as possible. Now enter:

```
TYPEDF0:S/Startup-sequence
```

After the drive runs for a short time, the contents of the file is displayed on the screen. We recommend that you have your mouse in hand right after you press the <Return> key. Press and hold the right mouse button to stop the scrolling of the screen; release the right mouse button to continue the scrolling. The Workbench Startup-sequence appears on the screen (an Amiga 500 Startup-sequence is given here; the Amiga 200 Startup-sequence is longer).

If you take a good look at this file, you may notice that only `AmigaDOS commands` are used. The commands of this file automatically EXECUTE after you start the Amiga.

This file tells the computer what conditions should exist when it starts (memory configuration, etc.). When you make changes to the Startup-sequence, remember that they should only be made to the copy of the Workbench disk. It's important that you have an unmodified Workbench in reserve, in case you make a typing error.

Your `Startup-sequence` may vary since the Amiga system is always expanding and improving.

```
c:SetPatch >NIL: r ;patch system functions
Addbuffers df0: 10
cd c:
echo "Amiga Workbench Disk. Release 1.3.2 version 34.32"
Sys:System/FastMemFirst ; C00000 memory to last in list
BindDrivers
SetClock load ;load system time from real time clock
```

```
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 Workbench
LoadWB -debug  ;wait for inhibit to end before continuing
endcli >NIL:
```

A line here or there could be removed. We'll discuss each line in succession, starting with the first. This line should not be deleted under any circumstances because this is where the system is "patched", or modified a bit. It should be mentioned that the SETPATCH command should only be entered once. This command should not be entered in the AmigaDOS Shell.

Now we come to the second line. The value of the ADDBUFFERS command could be changed from 10 to another value, but don't do it. It should also be retained.

The next line changes to the AmigaDOS command directory, CD C:. The ECHO command displays the startup message and lists the current release and version number. The next command, FASTMEMFIRST, organizes the memory area and should stay. The BindDrivers command should be eliminated if you don't have any memory or hardware expansion in your computer.

Whether or not you keep the SETCLOCK line depends on if you have a battery-powered realtime clock in your computer.

To obtain faster text output, the FF command must be used. This isn't absolutely necessary. This means that the command can be removed.

The following RESIDENT command absolutely must be there, otherwise the new Shell will not function. You can decide for yourself whether the EXECUTE command and other commands should be in the RESIDENT list. The new Shell needs the new Console handler NEWCON:. Next the NEWCON: is mounted.

The FAILAT command allows the following script file to be executed even if it encounters errors. The StartupII script file is then executed. Here is the StartupII script file:

```
resident c:Resident pure
resident c:List pure ;pre-load LIST and CD
resident c:CD pure
resident c:Mount pure ;the next 3 are loaded for speed
during startup
resident c:Assign pure
resident c:Makedir pure
;make IF, ENDIF, ELSE, SKIP, ENDSKIP, and ECHO resident
;if you use scripts much, and can afford the ram.
;also make Failat, WAIT, and ENDCLI resident if you use
;IconX a lot
makedir ram:t
assign T: ram:t ;set up T: directory for scripts
makedir ram:env ; set up ENV: directory
assign ENV: ram:env
makedir ram:clipboards ;set up CLIPS: assign
assign CLIPS: ram:clipboards
mount speak: ;just mounting doesn't take much ram at all
mount aux:
mount pipe:
resident Mount remove ;if you have enough ram, keep these
resident
resident Assign remove  ;by removing these lines
resident Makedir remove
;
break 1 C ;signal to other process its ok to finish
```

The first few commands make often used AmigaDOS commands resident for faster processing of the script file. Next important system directories are created and their paths assigned so that the Amiga can locate these directories. The devices are then mounted into the Amiga system. Finally commands are removed from the RESIDENT list.

The BREAK command signals the Startup-sequence that the StartupII sequence is finished. The Startup-sequence then continues from the WAIT command.

The next two lines adding the path and setting the correct key map should not be deleted. The following line loading the Workbench should be left alone. This line should always be kept. The last line closes the AmigaShell window. Whatever you want to add is up to you. Remember that every line you add increases the duration of the startup sequence.

You don't have to create a new sequence from scratch; just change the old sequence. The easiest way to change the old one is to use the ED editor from the Workbench disk.

Making changes:

Place the **copy** of the Workbench disk in drive **DF0:**. Enter the following command sequence from the AmigaShell:

```
ED DF0:S/Startup-sequence
```

The drive runs for a short time. The editor starts and the original Startup-sequence appears on the screen. The only thing that you must do now is position the cursor, with the help of the cursor keys, so that it stands in a line that isn't needed in your new startup sequence. It doesn't matter where the cursor is in the line. To erase this line, simply press <Ctrl><B>. All of the lines that need to be erased can be disposed of in the same manner. When all of the lines that don't belong in the new startup sequence are erased, the text must be saved. Press the <Esc> key, the <X> key and the <Return> key to save the edited text.

Now you have a faster Startup-sequence. Read the next section to see how you can make further modifications to make your own custom Startup-sequence.

6.2.1 A Custom Startup-sequence

In the previous section you discovered that it is possible to change the Startup-sequence. The Startup-sequence can be made more user-friendly. That means that the Startup-sequence has to be lengthened by a few lines. In the end you'll have to decide which is more important: speed or user-friendliness.

Because we want to modify the Startup-sequence again, we need a basic setup. We took the shortened version from Section 6.2 and edited it. The startup sequence of the Amiga 2000 can be edited in the same manner.

Before we change the sequence make sure you are familiar with the ASK command. In case you aren't, you should look back at Section 2.3.6.

Now back to the custom Startup-sequence. Have you ever been annoyed that you had to click on an icon after startup before you could have an AmigaDOS window? You can get around this problem by deleting the line:

```
EndCLI > NIL:
```

The problem can be handled more elegantly by adding the ASK command to the Startup-sequence. We would like to make a suggestion as to how this would work. Insert the following lines in your startup sequence before the last line.

```
ask "Would you like to open an AmigaDOS window ? (y/n) "
if warn
 ask "A large (y) or small (n) AmigaDOS window?"
  if warn
   newcli "con:0/0/550/200/Startup AmigaDOS "
  else
    newcli "con:0/0/160/30/Startup AmigaDOS "
```

```
  endif
endif
```

When you have finished adding this to your `Startup-sequence`, you can see the results by pressing the <C=> key, right <Amiga> key and the <Ctrl> key at the same time. Make sure to save the `Startup-sequence` before you reset your Amiga.

The `Startup-sequence` re-EXECUTEs. Before it is completely done, a question is asked:

```
Would you like to open an AmigaDOS window ? (Y/N)
```

You need to press the <Y> key or the <N> key, depending on whether you want a window or not. <N> closes the window and you're back to the Workbench. <Y> prompts another question: The program asks if you want a large or small window. Regardless of what you enter, an AmigaDOS window with the name `Startup AmigaDOS` opens. <Y> opens a large window. Pressing <N> opens a very small window.

We only wanted to show that something like that is possible. This principle of using `ASK` can be applied to other things as well. For example, you could ask if the Workbench should be loaded or not. When you are working with AmigaDOS, the icons aren't needed. This can be accomplished easily by using the `ASK` command:

```
ask "Should the Workbench be loaded? (y/n)"
if warn
  loadwb
endif
```

The `LOADWB` line in the `Startup-sequence` must be replaced by the above four lines.

The two examples above are only a small portion of what can be done to customize the `Startup-sequence`. You can arrange the sequence to fulfill your wishes and needs.

6.2.2 Shell-startup sequence

There are more startup files in the `S:` directory. One is called the `Shell-Startup` file. The `Shell-Startup` file is EXECUTEd every time a `Shell` window is opened. AmigaDOS 1.3 has both a `CLI-Startup` and a `Shell-Startup` file, Version 2.0 only has the Shell-startup. In the `CLI` this is only the command `Prompt "%N"`. In the `Shell` the following command is used:

```
alias xcopy copy [] clone
alias endshell endcli
c:Prompt "%N.%S> "
```

The Shell-Startup file may contain more ALIAS commands. What the ALIAS command does and how it can be used is discussed at the end of this chapter. There are also more examples for using the ALIAS command. These examples can be integrated into the Shell-Startup file. The Shell-Startup can be edited with an editor or a word processor. The CLI-Startup can be edited in the same manner, but the ALIAS command is not allowed in the AmigaDOS 1.3 CLI only with the Shell.

The S: directory is a good place for storing your custom script files.

6.3 Practical Script Files

Using script files can save you a lot of typing and time. We have put together some script files for you to examine. Even if you don't use these files you should look at them for examples of what is possible. You may want to create your own script files.

This book has an companion disk available for it. On this disk there is a directory named `Scripts`. All script files in this chapter are found in this directory. The names of each are found in a comment line at the beginning of each program.

6.3.1 A special printer script file

Have you ever printed a file out on your printer? You may have noticed that the text always has the same style. This section shows you how to change this situation. To do this, a little knowledge about printers is necessary.

Different control characters can be transmitted to the printer. These control characters control the appearance of the printed text. The Amiga uses printer escape sequences that send commands to the printer. We can send the printer escape codes with the help of the command:

```
COPY * TO PRT:
```

After entering this command and pressing the <Return> key, all AmigaDOS input goes directly to the printer. You only have to enter the desired command sequence on the keyboard. Since it's difficult to place these escape sequences, which are generated from the keyboard, inside a script file so that they can be sent to the printer, you must send the data in an indirect way.

We put these printer escape codes in different files for that reason. We would like to show an example of how this is done. In our example we'll deal with the command sequence for the NLQ (near letter quality) type style. Enter the following line in the „AmigaDOS window:

```
COPY * TO NLQ
```

The drive runs a short time after you press the <Return> key. It creates the file `NLQ`. All keyboard input that follows is sent directly to this file. That means that no input is shown on the screen. You will be typing "blind". When you enter the command sequence for the type style NLQ, it can't contain any errors or it will not work. Pay special attention when entering the following characters (Esc refers to the <Esc> key):

```
ESC[2"z
```

After you press the <Return> key the drive runs for a short time. The command sequence mentioned above goes to the file.

To return to normal mode, press <Ctrl ><\>.

The procedure is the same for accessing Bold, Italic and Reset printer type styles (just use the filenames Bold, Italics and Reset instead of NLQ). The following printer escape sequences are translated using the printer drivers included in the `Preferences` editors.

Printer Escape sequence	Meaning
<Esc>c	Initialize (reset) printer
<Esc>#1	Disable all other modes
<Esc>D	Line feed
<Esc>E	Line feed + carriage return
<Esc>M	One line up
<Esc>[0m	Normal characters
<Esc>[1m	Bold on
<Esc>[22m	Bold off
<Esc>[3m	Italics on
<Esc>[23m	Italics off
<Esc>[4m	Underlining on
<Esc>[24m	Underlining off
<Esc>[xm	Colors (x=30 - 39 [foreground] or 40 - 49 [background])
<Esc>[0w	Normal text size
<Esc>[2w	Elite on
<Esc>[1w	Elite off
<Esc>[4w	Condensed type on
<Esc>[3w	Condensed type off
<Esc>[6w	Enlarged type on
<Esc>[5w	Enlarged type off
<Esc>[2"z	NLQ on
<Esc>[1"z	NLQ off
<Esc>[4"z	Double strike on
<Esc>[3"z	Double strike off
<Esc>[6"z	Shadow type on
<Esc>[5"z	Shadow type off

Printer Escape sequence	Meaning
<Esc>[2v	Superscript on
<Esc>[1v	Superscript off
<Esc>[4v	Subscript on
<Esc>[3v	Subscript off
<Esc>[0v	Back to normal type
<Esc>[2p	Proportional type on
<Esc>[1p	Proportional type off
<Esc>[0p	Delete proportional spacing
<Esc>[xE	Proportional spacing = x
<Esc>[5F	Left justify
<Esc>[7F	Right justify
<Esc>[6F	Set block
<Esc>[0F	Set block off
<Esc>[3F	Justify letter width
<Esc>[1F	Center justify
<Esc>[0z	Line dimension 1/8 inch
<Esc>[1z	Line dimension 1/6 inch
<Esc>[xt	Page length set at x lines
<Esc>[xq	Perforation jumps to x lines
<Esc>[0q	Perforation jumping off
<Esc>(B	American character set
<Esc>(R	French character set
<Esc>(K	German character set
<Esc>(A	English character set
<Esc>(E	Danish character set (Nr.1)
<Esc>(H	Swedish character set
<Esc>(Y	Italian character set
<Esc>(Z	Spanish character set
<Esc>(J	Japanese character set
<Esc>(6	Norwegian character set
<Esc>(C	Danish character set (Nr.2)
<Esc>#9	Set left margin
<Esc>#0	Set right margin
<Esc>#8	Set header
<Esc>#2	Set footer
<Esc>#3	Delete margins
<Esc>[xyr	Header x lines from top; footer y lines from bottom
<Esc>[xys	Set left margin (x) and right margin (y)

Printer

Escape sequence	Meaning
<Esc>H	Set horizontal tab
<Esc>J	Set vertical tab
<Esc>[0g	Delete horizontal tab
<Esc>[3g	Delete all horizontal tabs
<Esc>[1g	Delete vertical tab
<Esc>[4g	Delete all vertical tabs
<Esc>#4	Delete all tabs
<Esc>#5	Set standard tabs

The type face on the screen may change when you access the printer type style. For example, when you EXECUTE the command sequence for italics, the output on the screen appears in italics. This can be solved by sending the command sequence for reset (<Esc><c>) as the last code you transmit. This resets all styles to normal.

The following script file uses the four script files `NLQ`, `Bold`, `Italics` and `Reset` contained on the companion diskette. The files contain the appropriate printer escape sequences. The optional disk for this book, named "AmigaDOS Opt Disk" contains these files in a directory named `Printer_routines`. Copy or create these files and place the four files in a directory with this name or alter the script file so it can find them.

This is the script file referred to by the title of this section. Enter this file using an editor and save it under the name `Printer`. You may wish to change the disk name and path name to match your system.

```
.key Filename
;Printer
if "<Filename>" eq ""
   ECHO "*nYou must enter a filename.*n"
   quit
endif
if "<Filename>" eq "??"
   ECHO "*n*nCall: EXECUTE Printer Filename*n"
   ECHO "Don't forget to enter the path.*n"
   quit
endif
if exists <Filename>
   ask "Print the file in NLQ? (y/n) "
   if warn
      copy AMIGADOS_OPTDISK:Printer_routines/Nlq to prt:
   else
      ECHO "Ok, draft mode, then."
   endif
   ask "Print the file in bold type? (y/n)"
   if warn
      copy AMIGADOS_OPTDISK:Printer_routines/Bold to prt:
   else
      ECHO "Ok, no bold text, then."
   endif
```

```
   ask "Print the file in italics? (y/n)"
   if warn
      copy AMIGADOS_OPTDISK:Printer_routines/Italic to
prt:
   else
      ECHO "Ok, no italics, then."
   endif
   copy <Filename> to prt:
   copy AMIGADOS_OPTDISK:Printer_routines/Reset to prt:
   ECHO "Ready."
else
   ECHO "Sorry...I can't find the file <Filename>."
endif
```

To print the `Startup-sequence` with this script file using the companion diskette, enter the following on one line:

```
EXECUTE AMIGADOS_OPTDISK:SCRIPT_FILES/PRINTER
SYS:S/STARTUP-SEQUENCE
```

You may have to alter the above command depending on where you stored the `Printer` script file. You can call it using `EXECUTE`, the script filename and a filename to print. The questions are asked one after another. Be sure that your printer is capable of each printer option before answering each question.

When the questions have been answered, the `Startup-sequence` file starts to print out in the selected type style. A different filename can be substituted instead of the `Startup-sequence`. The pathname must also be given if the file is stored in a subdirectory.

We used only three type styles. You can add more styles to the file if you desire. Before doing this, the respective escape sequence must be written to the file as described above.

6.3.2 Creating your own script files

Here are two more examples of creating your own script files. You could consider these new commands, even though they are accessed through the `EXECUTE` command.

BACKUP The first example copies any file on a diskette to a backup copy. The newly created file is different from the original in name only (copies have file extensions of `.bak`). Enter the following lines in an editor and save them under the name `Backup`.

```
.key Filename
;Backup
if "<Filename>" eq ""
   ECHO "*nYou must enter a filename.*n"
   quit
endif
if "<Filename>" eq "??"
   ECHO "*n*nCall: EXECUTE Backup Filename*n"
   ECHO "Don't forget to enter a path.*n"
   quit
endif
if exists <Filename>
   copy <Filename> to <Filename>.bak
else
   ECHO "*nSorry, I can't find the file <Filename>.*n"
endif
```

You can call it using `EXECUTE`, the script file and a filename. A complete command line looks like the following:

```
EXECUTE BACKUP Filename
```

This creates a file that has the label `Filename.bak`.

WINDOW

The second script file lets you open up to six AmigaDOS windows with one common command line. Enter the following lines in an editor and save the file under the name `Window`.

```
.key number
;Window
if "<number>" eq "??"
   ECHO "*n*nCall: EXECUTE Window number*n"
   ECHO "Number must be between 1 and 6 inclusive.*n"
   quit
endif
if "<number>" eq ""
   ECHO "*nYou must enter the number of the window.*n"
   quit
else
   skip <number>
   lab 6
    newshell "con:0/0/319/59/A_CLI"
   lab 5
    newshell "con:320/0/319/59/A_CLI"
   lab 4
    newshell "con:0/60/319/59/A_CLI"
   lab 3
    newshell "con:320/60/319/59/A_CLI"
   lab 2
    newshell "con:0/120/319/59/A_CLI"
   lab 1
    newshell "con:320/120/319/59/A_CLI"
endif
```

The command can be called by entering the following, with the variable n representing a number between 1 and 6 (be sure you are in the correct directory):

```
EXECUTE WINDOW n
```

The Amiga may respond with an error message. This can be caused by insufficient memory, or by the user entering a number outside the allowable numeric range.

Note: Before we end this section, we want to give you a piece of advice dealing with the t directory, the directory used for temporary storage. It stores different files here after an EXECUTE command has been run. These are used internally by the computer. When a script file is called and the message Disk is write protected appears, don't panic. The computer can only grab things from the T directory.

When working with a RAM disk, this message does not appear. The Startup-sequence usually creates a t directory in the RAM disk.

6.3.3 Starting script files with the mouse

Have you noticed that you must do a lot of preparation with a script file before you can start it? First you must double-click on the disk icon, then open an AmigaDOS window. A script file can also be started with a mouse click. AmigaDOS supplies a command that enables you to avoid some of this work. This command is called IconX and is found in the C: directory. Here's an example. Enter the following line in the Shell:

```
ECHO >ram:Batch "dir df0:*ncd ram:*ntype Batch"
```

With DIR RAM: you can check if the script file exists in the RAM disk. The file can be EXECUTEd by entering EXECUTE RAM:Batch in the AmigaDOS Shell. This displays the contents of drive **DF0:** on the screen followed by three command lines that display the contents of the script file. What must be done so that this file can be started with the mouse? First, we need an icon. This icon must be a project icon like the AmigaDOS Shell icon. We will use this icon in our example.

Exit to the Workbench. Open whatever drawers you need to get to the Shell icon. Click once on the icon. Press and hold the right mouse button. A menu bar appears in the first line of the screen. Workbench 2.0 users should select the Information item in the Icon menu, 1.3 users select the Info item from the Workbench menu. An Information window appears. In 2.0 the name of the program is listed and the type is listed (Project), in parentheses. In 1.3 the

word TYPE appears in the upper left hand corner. Right next to it is the word Project. When the program is of type Project or TOOL, this icon can be used for our purposes.

Open Shell again. Enter the following to copy the Shell icon information to the Batch script file in the RAM disk:

```
Copy SYS:Shell.info RAM:Batch.info
```

When you have entered the command and pressed the <Return> key, return to the Workbench. Double-click on the RAM disk icon. In the window that appears you should recognize an icon possessing the name of the script file. Click once on this icon. Then select the Information item from the Icon menu (Info from the Workbench menu in 1.3). This displays the Information window. Inside of this window is a string gadget containing the DEFAULT TOOL status. Click on this string gadget. Edit the text so that it reads SYS:C/IconX instead of SYS:System/CLI. This text gives the path where the IconX command is found. After you have entered the new text, click on the Save gadget. The window disappears from the screen. The Batch icon appears in the RAM disk window. Double-click on it. This opens an IconX window. The same output appears in this window as was displayed when the script file was started with EXECUTE. After all of the output is given in the window, the window remains open for a short time before it automatically closes. The time that the window remains open can be increased by adding the Wait 30 command to the script file. The window then remains open for 30 seconds longer.

That was an example of how a script file can be started from the Workbench. All script files can be started this way, with a few exceptions. IconX executable script files can only contain commands that can be entered directly in AmigaDOS. Commands like Skip, Lab, If, etc. are not allowed. You can access disk drives other than the RAM disk. You can edit the icon's appearance if you wish.

6.3.4 The Types script file

Hopefully you saved your work in the S: subdirectory on the Workbench if you've created script files while working through this book. That is where the Amiga looks for a script file if it is not found in the current directory. The following script file displays the contents of each file in a directory. You should be sure that the files in the given directory are script files or text files; programs will print garbage when they are displayed on the screen.

```
.key Directory
;Types
if "<Directory>" eq "??"
     ECHO "*n*nCall: EXECUTE Types Directory*n"
     ECHO "Don't forget to enter the path name.*n"
     quit
endif
if "<Directory>" eq ""
     ECHO "*n*nYou must enter a directory.*n"
     quit
endif
if exists "<Directory>"
     cd "<Directory>"
else
     ECHO "I can't find the directory <Directory>.*n"
     quit
endif
list >ram:Type.bat #? lformat="type %s"
EXECUTE ram:Type.bat
delete ram:Type.bat
cd sys:
```

You can start this file with `EXECUTE` or set the `S` bit of this file. The drive and directory must be given. The file is called `Types` on the companion disk available for this book, for the companion disk the call would be the following (your call may differ depending on where you saved the file):

```
EXECUTE AMIGADOS_OPTDISK:Script_files/Types sys:s
```

The output can be stopped by pressing the right mouse key. The output can also go to a printer. For this, the `lformat` option must be changed to `lformat="type >prt: %s"`.

6.3.5 Putting everything into the RAM disk

The program in this section should only be used when the user has a memory expansion. This program copies the entire Workbench disk onto the RAM disk and directs all access that would normally go to the Workbench to the RAM disk. You can use the `rad:` device, the recoverable RAM disk for this which will give you super fast resets. You must change the `HighCyl` parameter in the `MountList` to provide enough memory in the RAD disk. Using this file can take up so much memory that the other tasks may not have enough room. The first script file uses the normal RAM disk, the second uses the reset-resistant RAM disk.

```
;Ramcontrol script file one
ECHO "*nCopying directories..*n"
copy sys: ram: all
cd ram:
ECHO "*nTurning control over to the RAM disk.*n"
assign sys: ram:
assign c: sys:c
assign l: sys:l
assign fonts: sys:fonts
assign s: sys:s
assign devs: sys:devs
assign libs: sys:libs
assign t: sys:t
assign utilities: sys:utilities
assign prefs: sys:prefs
assign system: sys:system
assign empty: sys:empty
ECHO "*nDone.*n"


;Radcontrol script file two
;Remember to MOUNT RAD: before executing this file
if exists rad:Flag
   skip L1
else
   ECHO "*Copying directories..*n"
   copy sys: rad: all
endif
ECHO >rad:Flag "Flag set."
lab L1
cd rad:
ECHO "*Giving control to reset-resistant RAM disk.*n"
assign sys: rad:
assign c: sys:c
assign l: sys:l
assign fonts: sys:fonts
assign s: sys:s
assign devs: sys:devs
assign libs: sys:libs
assign t: sys:t
assign utilities: sys:utilities
assign prefs: sys:prefs
assign system: sys:system
assign empty: sys:empty
ECHO "*nDone.*n"
```

The second file is somewhat longer than the first. This is because the reset-resistant RAM disk is always present and may be remounted using `mount rad:` after a warm start. Therefore, the Workbench files don't need to be copied again. The additional lines take care of this.

6.4 Using ALIAS

The Shell has many advantages over the Amiga's original CLI. One advantage is that the command lines can be edited and that the entered command lines can be stored in sequence in a buffer.

The ALIAS command is very useful. This command makes it possible to use AmigaDOS commands in a different manner. You may ask why we discuss it in the scripts chapter. The ALIAS command can function as a script file, except it only takes up one text line. The line begins with the command word ALIAS, followed by a character string which is the label of the new command, followed by a list of commands or commands that EXECUTE a script file. That sounds difficult, but it really isn't. Before we give you any examples, a bit of advice. We created a script file called Alias.bat on the companion disk available for this book. This is found in the Script_files subdirectory. When you open a Shell window, you must enter the following to use the Alias.bat file:

```
EXECUTE AMIGADOS_OPTDISK:Script_files/Alias.bat
```

Back to our example. Enter the following line in the Shell:

```
Alias Ramdir Dir Ram:
```

Now you have access to the new command RAMDIR. Here's what you did: First the command word (Alias), then the label of the new command (Ramdir) and then the command's function (Dir Ram:). The example isn't very useful, but it serves its purpose. Which ALIAS commands are in use and how they are built can be determined by entering ALIAS without arguments and pressing the <Return> key.

When a Shell window is opened, the Shell-Startup script file is EXECUTEd. Placing your ALIAS command in the Shell-Startup file enables these commands whenever any Shell window is opened. Or you could write them in a script file that must be called so that you can use the commands.

The next few pages contain examples of what can be done with the ALIAS command.

1. If you would like to change the disk drive without much typing. Use the following lines:

```
alias 0 cd df0:
alias 1 cd df1:
alias 2 cd df2:
alias r cd ram:
  .
  .
  .
```

After entering these lines you can change the drive by entering `0`, `1`, `2` or `r`. You can do so with other drives as well (example: **DH0:**). You can also change the directory in a given drive. To change the `C` directory to drive **DF1:**, you must enter the command `1 c`.

2. Delete the contents of a `Shell` window.

```
Alias CLS ECHO "*ec"
```

Enter the line in the `Shell`. Then enter `CLS`. The contents of the AmigaDOS window are erased and the prompt appears in the top of the window. This is accomplished with the `ECHO` command and an escape sequence. The sequence is inside the quotation marks. The first two characters tell the computer that an escape sequence follows. The `c` erases the screen. You'll find more escape sequences in Appendix A. The prompt stands in the second line, not the first. To get it to appear in the first line after the screen is erased we need to use a little trick. First, we must change the prompt character a little with the following line:

```
Prompt "*e[11y*e[33m*e[1m*e[3m%s*e[0m*n*e[t"
```

After you enter the line and press the <Return> key you can see the result. The directory is displayed in italics and is in a different color. In addition, the input takes place in the next line. These appearances are a result of the different escape sequences in the `PROMPT` command.

Now enter the new `CLS` command.

```
Alias CLSC ECHO "*ec*e[2y*e[2t"
```

When you enter `CLSC` to clear the screen, the directory description appears in the top line of the window.

3. Here are some more uses for the escape sequences. The first example doesn't use the `ALIAS` command but is still rather interesting.

```
ECHO "*e[1m*e[3mThis is bold and italic.*e[0m"
ECHO "*e[32m*e[43mBlack text on an orange background.*e[0m"
ECHO "*e[7m*e[4mThis is inverted and underlined.*e[0m"
```

As you can see the changes affect more than the output. The ALIAS command can produce many other changes.

```
Alias Prom1 Prompt "*e[32m*e[43m%s> "
Alias Prom2 Prompt "*e[42m*e[31m%s> "
Alias Prom3 Prompt "*e[41m*e[33m%s> "
Alias Prom4 Prompt "%Enter Task*n%s*n- "
```

Enter ECHO "*ec" to return to the normal colors.

4. Print a file on the printer. At the same time the computer should still be available for use. The file should print in the background.

   ```
   Alias Print run copy to prt: [] clone
   ```

   The brackets act as placeholders for the parameters entered with PRINT. The new command must be given the name of the file it should print out. For example:

   ```
   Print sys:s/startup-sequence
   ```

5. Here is a command that makes a file "invisible" so that it isn't displayed when a DIR or LIST command is EXECUTEd. The HIDE command hides the specified file from these commands:

   ```
   Alias Hide protect [] +h
   ```

   The PROTECT command sets the h status bit of the file. The computer must have at least Kickstart 1.3 to use this command. This h bit is ignored when Kickstart 1.2 is used.

6. As in example 5, the status bit s can be set. The following line must be entered:

   ```
   Alias SBit protect [] +s
   ```

   When the s bit is set you can EXECUTE a script file without invoking the Execute command.

7. There is already an ALIAS command in your Shell-Startup file. It is the xCopy command. The same principle can be used for deleting:

   ```
   Alias xDelete delete [] all
   ```

   After entering this line a directory can be deleted. Enter xDelete and a directory in the Shell.

8. The AmigaDOS commands can be shortened with the ALIAS command:

```
Alias c copy
Alias p path
Alias d dir
Alias ex EXECUTE
Alias dl delete
Alias t type
Alias r rename
Alias e ECHO
   .
   .
   .
```

The individual commands can be used after entering the lines with the respective abbreviations.

7.
AmigaDOS and Multitasking

7. AmigaDOS and Multitasking

The blitter

What fascinated us most about the Amiga was the efficiency of the hardware and software. While other computer manufacturers delivered the *blitter* (the special chip for super fast memory operations) months after the announced date, the Amiga system was supplied with it from the start. While many other systems are limited to 64 colors, the Amiga can display 4096 colors at the same time.

Even more efficient than the Amiga hardware is the software supplied with the Amiga. Other computers use windows, but are severely limited in the number of open windows and the number of programs they can run at once. The Amiga operating system allows true *multitasking* (the topic of this chapter), with a few limitations. The software is not yet capable to take full advantage of the hardware.

Amiga software

AmigaDOS makes it possible to do multitasking on a home computer without restrictions. This section of the operating system is laid out so that the hardware can do many different functions and is easily expanded. The principle of device drivers play an important part in this flexibility. AmigaDOS supports devices that can be addressed with the same routines, printing to the screen or the printer can be handled by the same routine. The direction of input and output for different devices is an essential condition for multitasking.

AmigaDOS

AmigaDOS has one drawback. This super operating system that was delivered with the computer does not fully utilize all of its amazing possibilities. Multitasking on an Amiga 500 with 512K memory and one disk drive is like driving a Porsche in heavy traffic—exciting, but limited. Also, the 68000 processor is very good, but AmigaDOS is much more efficient with a 68020 processor and lightning fast with a 68030. We can only hope that these possibilities will soon become standard for all Amiga users.

7.1 What is Multitasking?

Some of our readers may quietly laugh at this question and say that is an old subject. Multitasking is when a computer does many things at once. The question is not that dumb, however. Many of you work with AmigaDOS and wait for the disk to finish formatting or for the C compiler to finish compiling before going on to other work. You aren't using the full capabilities of AmigaDOS.

Multitasking

Multitasking is something completely natural. Many people rarely use the computer to do more than one thing at a time. They wait for it to finish its work before going further. As an example, we will take a typical human task that points out a few problems of multitasking:

Let's make lunch; roast beef with mushrooms and onions, potatoes and asparagus. It should be on the table at 12 noon. You couldn't get it done if you did each task one after another. You could prepare the potatoes and asparagus at 8:00 a.m. then at 11:30 a.m. finish the gravy and set the table. But by then, the potatoes and asparagus would be cold. You have to think about which task takes the longest and then plan your time accordingly. Preheat your oven starting at 9:45 a.m. Put the roast beef in the oven at 10 o'clock because you know that it takes two hours to cook. After that, place the water for the potatoes on the stove and slice the potatoes while that heats up. While the potatoes are cooking in the water, you have 20 minutes to prepare the asparagus.

Let's get back to the subject of multitasking. This was a very simple example of everyday multitasking, and many other examples can be thought of—from lighting a cigarette while driving to reading the newspaper while watching TV. These examples have the same problems: While multitasking you can scald your finger in the water while the roast beef burns, or you can crash into another car after your cigarette falls in your lap. It is exactly the same for the computer. You can try to save a file on a disk that needs to be formatted and you could write text to a file that should first be printed. These are conflicts that should be avoided, and the following sections show you how to avoid them.

Computer multitasking is implemented so that many tasks appear to work at the same time. Or to put it another way, each task operates for a very short time so that no task has to wait for another task to finish. Fewer problems arise using this method, so it's better for users. Luckily, AmigaDOS was written so well that collisions and burnt fingers never occur while multitasking.

7.2 Multitasking with AmigaDOS and Workbench

Maybe you have been shown by enthusiastic friends and acquaintances how the Amiga can do many things at once. Especially interesting are the multiple programs that can be started one after another. The multitasking ability of the Amiga is limited by memory; the more memory you install, the more you can do. You can hardly show someone BeckerText, DataRetrieve and AmigaBASIC at the same time if you have an Amiga 500 with 512K. In spite of this there are many uses for multitasking, such as using AmigaDOS and Workbench simultaneously.

Before we show you the many possibilities for multitasking with the Workbench and AmigaDOS, we must mention that the Workbench does have "multitasking capabilities." To see this, format a disk with Format Disk (1.3 `Initialize`) and try to do something else while the disk is being formatted. It works.

It is important to have the correct commands `RESIDENT` if you are using an Amiga 500 with one disk drive. When `Intuition` loads a program from the disk and AmigaDOS looks for the desired command from the same disk this process takes a long time. If you have two disk drives, the system disk with AmigaDOS commands should be in one drive and the program should be started from the Workbench in the other drive.

You should make a copy of the original Workbench disk because we will change the `Startup-sequence` on it. The normal Workbench `Startup-sequence` ends AmigaDOS with:

```
ENDCLI >NIL:
```

The AmigaDOS window disappears. Place a copy of the Workbench disk in drive **DF0:** and load the `Startup-sequence` into `ED` with:

```
ED DF0:S/Startup-sequence
```

and erase the last line (`ENDCLI >NIL:`). Save the file by pressing <Esc><X> and continue with your work on the Amiga. How can you work with the Workbench and AmigaDOS at the same time?

An example: Suppose that you have some important data in the RAM disk and that you start a program from the Workbench. This program locks up during loading because it doesn't have 1 megabyte available. Now you can't do anything more with the Workbench. Only the wait pointer appears—you can't open AmigaDOS. You can get out of this by resetting the computer, but then you lose the data in the RAM disk. Had you placed an AmigaDOS window behind all of the other windows, you could simply click on it and copy the important data from the RAM disk with the command:

```
COPY RAM DF0: ALL
```

before resetting the computer. AmigaDOS 1.3 doesn't help against the large red `Guru Meditation` message. When the requester:

```
"Task held-finish all disk activities ......"
```

appears, most of the time you are without an AmigaDOS window. Remember a tiny AmigaDOS window in the background doesn't use that much memory and can save a lot of frustration. AmigaDOS 2.0 has all but eliminated the Guru message.

Another use for the combination of the Workbench and AmigaDOS is the copying process of more data. Say you are working with BeckerText and a friend comes by and asks you to copy your text onto his/her disk. You could make the `BeckerText` window very small and place all data files to his/her disk using the Workbench. It is simpler to click on the AmigaDOS window and use the `COPY` command to give your friend a copy of your files. And if his/her disk isn't formatted, you can save even more time and interruptions when working with programs started from the Workbench. For example, enter:

```
FORMAT DF0:
```

in AmigaDOS. When the cursor is in the next line you can enter the next command:

```
COPY DF1:text/#? DF0:
```

You don't have to wait until the disk is done formatting to continue your work. Your friends disk will be filled with your data while you show him/her the newest game.

Deleting files

This can apply to deleting files also. Using the COPY command you have placed a backup copy of your text in a special directory named Security. Now the space on your disk is getting tight and you don't need the backup copy anymore. You could use the Workbench to open the desired drawer, mark all the text and then erase it with Delete (1.3 Discard). While the files are being erased you cannot work with the Workbench. It's much easier and faster to click on the AmigaDOS window and enter:

```
DELETE DF0:Security/#?
```

and erase everything in the directory, and you can continue to work with the Workbench while this is happening.

The size and position of the AmigaDOS window might get in your way when you work with the Workbench and AmigaDOS. It must first be made smaller and pushed aside before you can really work with the Workbench. Unfortunately, the size can't be changed without using the mouse. There is an easier way to get the desired AmigaDOS window. Open a new AmigaDOS window with the desired dimensions and close the old AmigaDOS window. This section of an improved Startup-sequence can look like this:

```
.....
........
newshell "con:10/10/60/60/MyShell"
........
endshell >nil:
```

7.3 Multitasking with NEWSHELL

You don't have to use AmigaDOS for everything. Executing important tasks through AmigaDOS can save time and make you much more efficient. We'll take a simple example that has nothing to do with multitasking: You have written five new letters today and placed them in directory `DF0:Letters`. There are 40 letters there already and you want to print out the five new ones on the printer. Before printing them out, you want to take a look at them without loading the word processor. You know that this can be done simply with:

```
TYPE DF0:Letters/textname
```

and printing the text is done with:

```
COPY DF0:Letters/textname TO PRT:
```

The first problem now emerges: You don't remember the name of each letter. So you enter `CD DF0:Letters` and by using `DIR` look at the first text name. Aha! The name was `Peter1.10.87`. Now you can quickly look at the text with:

```
TYPE Peter1.10.87
```

and then print the file to the printer. The print process is done quickly, but where are the other filenames. `TYPE` removed the filenames from the screen; you must re-enter `DIR` to get the next name. Now you can call up the next filename using `DIR` and repeat the entire process.

There's an easier way. Open a new AmigaDOS window with `NEWSHELL`. Position the second AmigaDOS window so that it occupies the top half of the screen. Enter `DIR`, so that the text names are displayed in the window (the output on the screen can be stopped by pressing any key). You can now read the text names in the first AmigaDOS window, change to the second AmigaDOS window, look at the files using `TYPE` and print them. This method is also valuable for deleting or copying multiple files. Display the files in one AmigaDOS window and execute the command in another AmigaDOS window. When you don't need the other window anymore, you can get rid of it by entering `ENDSHELL`.

Let's suppose that the texts are rather long and take a minute to print. Then in our example you would have to wait a minute before you could continue work with your text files. In such a case you could open another AmigaDOS window. In one window you could read the text names, in the second window you could view the texts with `TYPE` and in the third window you could print them. One thing is missing: The

coffee break that you always took while the computer was busy working. This coffee break is no longer necessary with AmigaDOS, unless of course you would like one.

We hope that you have had a small idea of the many possibilities that exist with AmigaDOS and multitasking. This is only a small appetizer. We want to show you how to finish your work in the shortest time. You should always work with a second AmigaDOS window open. It's the same thing with the Workbench: If the main AmigaDOS window is being used or is hung up by a program, you can continue to work with the second AmigaDOS window. If you innocently enter `COPY text TO PRT:` when working with one AmigaDOS window you have to wait until the text file is printed or else open a second AmigaDOS window through the Workbench. It's much better to have a second AmigaDOS window open and simply be able to continue.

AmigaDOS can affect new AmigaDOS windows. Say you have chosen `DF0: Texts/Private` as the current directory, then opened another AmigaDOS window with `NEWSHELL`. New input is automatically sent to the directory of the previous AmigaDOS window. In our case, the current directory of the second AmigaDOS window is `DF0:Texts/Private`, you don't need to state the new directory.

It is just as easy to enter the dimensions and position of the new AmigaDOS window. A small AmigaDOS window can be created in the upper left hand corner with:

```
NEWSHELL CON:10/10/60/60/MySHELL
```

It is best to enter this long line only once using `ED` and save it on the disk under the name `NS` (for `NEWSHELL`). Then the new AmigaDOS window can be called by entering:

```
EXECUTE DF0:NS
```

Then if you have renamed the `EXECUTE` command to `E` and **DF0:** is the actual directory, the line that has to be entered for the second AmigaDOS window to appear is:

```
E NS
```

You could also save the `NC` script file in your `S:` directory and set the script file bit with `PROTECT NS +S` so you could simply enter `NS` when you wanted a new AmigaDOS window.

7.4 Multitasking using RUN

The possibility of using NEWSHELL to work on many different tasks at once is certainly a great help and time saver. We did run into a few problems after opening four windows:

- Although the Amiga can manage more screens at once, it's limited by the size of the monitor because all AmigaDOS commands work with windows on the Workbench screen. They quickly take over the entire screen. It isn't acceptable to constantly click windows into the foreground or background.
- Each screen requires a good portion of memory, especially on an Amiga 500 with 512K of memory, and as more screens are added, there is hardly any memory left to do useful work.

The RUN command was created to execute many tasks at once without needing a window for each task. With this command an AmigaDOS window is executed without its own window. Let's see how this works. Place the Workbench disk with AmigaDOS commands in **DF0**. Then enter the following two lines:

```
RUN COPY DF0:C/DISKDOCTOR RAM:
DIR DF0:
```

Immediately after you press the <Return> key a new line appears:

```
[CLI 2]
```

and then the `1SYS:>` (when more AmigaDOS processes have been started, this is a higher number). Then you entered the second line (hopefully quick enough that the drive was still working) and AmigaDOS displayed the contents of the directory. This example doesn't make too much sense, but it shows the basic use for the RUN command: all tasks that may possibly have a long duration and don't place output on the screen are allowed to be called with RUN.

While the first restriction is relatively evident—the process lasted only a second so you can hardly type the next line fast enough—the second restriction is not so obvious. Now a small example of two tasks that are running at the same time, one is started with RUN. Enter the following two lines one after another:

```
RUN DIR DF0:
TYPE DF0:S/Startup-sequence
```

At first glance everything appears to be running normally, but suddenly the directory of **DF0:** appears in the list of `Startup-sequence`

commands. That is because an AmigaDOS command started with RUN doesn't have a window to use for its output. It puts all of its output in AmigaDOS window.

In spite of this restriction, there are many uses for multitasking using RUN. These include all of AmigaDOS commands that don't generate any screen output and require relatively little time. So, for example, a text can be printed using:

```
RUN COPY Text PRT:
```

and then further work can be done. Amiga users that have a disk drive should be aware that while text is printing, the disk that the text is being read from cannot be removed from the drive. The text should be copied into the RAM disk if you want to work with another disk while it is printing.

The RUN command has a very important job when you want to start a program from AmigaDOS and would like to continue to work during the time required for the program to load. One option would be to start a new AmigaDOS window with NEWSHELL and call the program from the second AmigaDOS window with:

```
Program
```

Then you have a useless window that not only takes up space but also memory. It is easier to start the desired program from the first AmigaDOS window with:

```
RUN Program
```

A typical example uses the editor. For example, you have written a C program and have to enter corrections in the program. If you are using only one AmigaDOS window, you must leave the editor, call the compiler and then start the editor again. Starting the editor from a new AmigaDOS window requires more screen space and memory. Call the editor simply with:

```
RUN ED Program.c
```

After saving with <Esc>+<SA> you can call the editor again and after the first error message, you can edit the line in the editor once more. This saves a lot of time.

There is another restriction to RUN that can have unpleasant results. Remember, a program that is started with RUN is missing its own input and output windows. The output can go to already existing windows, but where can the input be entered? You've probably noticed that input is done in windows. Every program that needs input from the keyboard needs a window. Input for two independent programs through one window isn't possible. How would the Amiga know which program the input belongs to?

Maybe we should say "That AmigaDOS commands don't need any more input". But that is not completely true. The AmigaDOS commands allow <Ctrl><C> or another <Ctrl> function to be input. When a program or AmigaDOS command is started with RUN, <Ctrl><C> doesn't go to the command, but instead to AmigaDOS. To put it another way, you have 30K text you want to display it with:

```
RUN TYPE text
```

The output can be paused by pressing a key, but stopping the output by pressing <Ctrl><C> doesn't work. The inventors of AmigaDOS saw this problem and built in a solution. This solution is the BREAK command. Using BREAK, a <Ctrl><C> can be sent to commands that were started with RUN. Start the text output from the first AmigaDOS window with:

```
RUN TYPE text
```

you can enter:

```
BREAK 2
```

to stop the text output from the second AmigaDOS window, CLI 2. The complete command is discussed in Chapter 3. It's enough for us to show how to stop commands started with RUN.

7.5 Using AmigaDOS

You probably got an impression of how much time and work could be saved by using the full range of AmigaDOS options in the AmigaDOS Tricks and Tips chapter. It is important that you know the different options and their effects. You begin to learn these by working with the Workbench. By doing this you can comfortably resolve many tasks, but you must use AmigaDOS to avoid long pauses when copying and formatting disks. In addition, AmigaDOS, used in combination with the Workbench, can make many things possible that aren't possible from the Workbench. Not only can you see all the data files with AmigaDOS, but you can also look in drawers without icons. That is important when you have a disk that doesn't show anything on the Workbench.

As you work more with AmigaDOS, you should plan your use of AmigaDOS processes. You shouldn't use the RUN command when using commands that produce output to a window. It is extremely important to have a second AmigaDOS window in the background at all times for security.

An effective way to use multitasking with AmigaDOS is to correctly plan the devices. That can cause drastic results if you only own one disk drive and have only 512K. Then you must live with a few limitations. Here you should decide which AmigaDOS commands are most frequently used and make these RESIDENT.

It isn't possible to print text from two different disks. You must plan ahead and have the text saved on one disk or use the RAM disk.

Users of the 1 megabyte Amiga computers have it easier. They can put important programs in RAM and a generous amount of commands as RESIDENT, and they can also copy important files into RAM. Adding an additional disk drive to an Amiga is very advantageous. Then the Workbench disk can remain in **DF0:** and the data can be in **DF1:**. That way a large amount of memory can be saved for programs.

It is not possible to print two data files on the printer at the same time and make it readable. While this may be possible on the screen (start one TYPE with RUN and a second TYPE from the original AmigaDOS window) AmigaDOS prevents the output of more than one task or process through a port (printer, RS-232).

Most good word processors can print text to a file. When a text file, which contains all of the printer command codes, is printed, it can be sent to the printer from AmigaDOS. Meanwhile, the word processor prints the next text to a another file.

Long directories with many files take a long time to be displayed using DIR. The output can be directed to a file and this file displayed on the screen using TYPE. This is much quicker than waiting for the result. Simply enter:

```
RUN DIR >RAM:Contents DF0:c
```

and later, using:

```
TYPE RAM:Contents
```

you can display the contents of the large C: directory very quickly. RUN is very important if we don't want to wait for the DIR command.

A very efficient use of multitasking can be achieved by skilled use of script files. Workbench 1.3 has a very long Startup-sequence that sets up the Amiga. The Startup-sequence we use lasts two minutes. Why should we wait this long? Simply insert this line at the beginning of your Startup-sequence:

```
NEWSHELL
```

and you will have an AmigaDOS window in which to work. You can begin work before the complete Startup-sequence is finished.

7.6 CHANGETASKPRI

In the last section we learned about the different ways to complete tasks at the same time. A lot of waiting time can be spared that way. Still, a few problems arise that prevent the full multitasking capacity from being used. Take an example:

The ED editor is not very fast. When you want to do work with many tasks, you are better off using ED. Everything is done in brief time intervals, one task after another. For important tasks, this can be very disturbing. The Amiga operating system can order each task according to a priority. Tasks with high priority are handled first, tasks with lower priority are handled last.

Suppose you entered text with ED and want to print it and continue to edit it at the same time. In this case it is certainly not that important if the text is printed after one or two minutes. It's more important that you don't have to wait to continue editing until the editor finally comes up with the next line. To make sure it's done in the right order, different priorities are assigned. The CHANGETASKPRI command changes the priority of AmigaDOS processes. This command sets the priority for AmigaDOS tasks at a number between -128 and +127. You can see this with:

```
STATUS FULL
```

and the actual information about AmigaDOS processes appears. The display for an AmigaDOS window looks like the following:

```
task 1: stk 4000, gv 150, pri 0 loaded as command: status
```

The `pri 0` is important to us. That is the priority of our process and also the standard value at which all AmigaDOS windows are automatically started. Now we'll change this value to a 3:

```
CHANGETASKPRI 3
```

When we display the information about the process again, we get the following notice:

```
task 1: stk 4000, gv 150, pri 3 loaded as command: status
```

We see that something has changed, the `pri 3`. Our AmigaDOS window has a priority of 3. Now you may think that you will wait forever to access AmigaDOS. This isn't the case. This is because the Amiga has a multitasking operating system and does not wait for one task to finish before moving on to the next. Now a task with a lower

priority comes along and also a task with a higher priority. We can investigate this quickly with an example. Create a second AmigaDOS window with NEWSHELL and arrange the windows so that the first window occupies the bottom half of the screen and the second window takes the top half. Now we want to observe the difference in processing when they have different priorities.

Enter CHANGETASKPRI -127 in the bottom window and CHANGETASKPRI 127 in the top window. Now it comes down to exact stopping and starting of the execution. Write the following command line in both windows (don't press the <Return> key):

```
LIST DF0:c
```

Now you must enter both commands as quickly as possible. Click the mouse in the lower window, put the mouse in the top window, press the <Return> key, click above and press the <Return> key again.

You can follow the different speeds that the contents of DF0:C are displayed. Notice that the speed difference is not much. That is not possible because of the way the operating system is programmed. As soon as a task requires an action from the operating system and the task waits for that action, it is placed in a waiting list and doesn't require anymore computing time. Then the tasks with a much lower priority have the chance to get in line. This is the case, for example, when a task reads a track on a disk. It stays in the waiting list until the track is completely read.

Despite this apparently small difference, for which we have the programmers of the Amiga operating system to thank, the difference can be put to good use. Enter CHANGETASKPRI 0 in AmigaDOS window 1 and create a second process with:

```
RUN ED RAM:difference
```

Press the <Return> key 20 times in the empty window of the editor so that the cursor is inside the ED window. Then move the AmigaDOS window so that you can input next to the editor. Display the directory of DF0:C in this window with:

```
DIR DF0:C
```

and try to produce the cursor inside the ED window simultaneously. You'll notice that this is difficult and it always stops. Now we want the work in the editor to be more important so we must change the priorities accordingly. Leave the editor using <Esc><X> and change the priority of AmigaDOS to 99 (CHANGETASKPRI 99). Restart the editor (RUN ED RAM:Difference) and set its priority to -99. Look at the distribution of the priorities with STATUS FULL. It will look something like this in AmigaDOS 1.2:

```
task 1: stk 1600, gv 150, pri 157 loaded as command: status
task 2: stk 3200, gv 150, pri 99 loaded as command: ed
```

There is a bug in the AmigaDOS 1.2 `Status` command. This was corrected in Workbench 1.3. The `pri 157` should really be -99. The AmigaDOS 1.2 status command can only show numbers from 0-255 and the priorities have numbers from -128 to +127. So a -99 is shown as +157. Now we want to go through the same test with the changed values. Start the output of the long directory in AmigaDOS and try to produce the cursor in the editor.

You'll soon determine that the work with ED is no longer hindered. The output of the filenames is stopped completely when a key is pressed in the ED window. ED doesn't wait for pauses in the directory, but `CLI 1` waits for pauses in ED.

Now let's put this information together:

- To work with many tasks without problems, a correct distribution of priorities is necessary. When the less important tasks require less time, the more important tasks are not hindered.

- The priority can be changed with CHANGETASKPRI. Values between -128 to +127 can be used. Although we used values from +99 to -99 in our example, normally values should be set from +5 to -5. We'll show you exactly why in the next section. In our example the large numbers were not a problem.

- The use of CHANGETASKPRI should be done correctly. Only the priority of AmigaDOS processes, that were called from the command, can be changed. This AmigaDOS process gives its priority to all "daughter processes" (like standard input and output and current directories). The priorities of processes started with RUN can also be set to the desired value.

- The priorities can be examined using STATUS FULL. Negative values are displayed incorrectly in Workbench 1.2. The correct value can be obtained by subtracting 256 from the number displayed on the screen. If 255 is displayed, the difference is -1, the correct value. This was corrected in Workbench 1.3.

- Only AmigaDOS processes can be influenced with CHANGETASKPRI. It's not possible to change the priority of a program started from the Workbench.

7.7 Multitasking dangers

We mentioned at the beginning of this chapter that multitasking not only has important advantages but also needs to be used correctly. In this section we want to point out a few limitations and dangers. We want to show, with an example, where problems are possible. Enter:

```
CD DF0:
RUN DIR
RUN LIST
```

A surprising result occurs, especially when you have an Amiga 500, if you enter these last two lines quickly and press the <Return> key. It behaves like the Amiga is tearing the disk in half, and each command can only use one half of the disk. The problem lies in the fact that each process can read a little portion of the disk. Because the regions that have been read are far from each other, the read head of the disk drive must travel large distances. In the most extreme case it must go from track 79 for one process and then go back to track 0 for the other process. This not only wears out the machinery and your ears, but it also wastes time. Many times the processing of two commands can take much longer than it would take to execute them one after another.

Our example is not useful if the output occurs in one window. It should only make the basic phenomena clearer. This problem constantly comes up when a process is reading information from a disk and must load the next AmigaDOS command from the same disk. That happens in the following example:

```
RUN COPY DF0:text TO PRT:
DIR DF0:
```

While `COPY` is reading from the disk to print the `Text`, AmigaDOS must read the next command (`DIR`) from the disk. In this case the movement of the drive head does not last very long. The use of a second drive or the RESIDENT command are helpful. An external drive or RAM disk will do the trick. It is important that different processes do not access the same drive. In our case we need AmigaDOS commands in **DF0:** and the text in **DF1:**.

Another problem occurs when multiple processes must access one file at the same time. Reading a file at the same time is not a problem. This can be seen by opening a second AmigaDOS window and displaying the data file in both windows. AmigaDOS refuses to let another process access a file if a program or process has opened that file for writing. That is important because otherwise invalid data could be read. We'll examine this in the following example:

```
COPY DF0:s/Startup-sequence RAM:Datafile
CD RAM:
COPY Datafile Datafile1
```

Now try to create a new file, `Datafile1`, and at the same time read `Datafile1`. Simply enter:

```
RUN TYPE > Datafile1 Datafile
TYPE Datafile1
```

After a short time the message `Can't open Datafile1` is displayed. When `WHY` is used to ask the reason for the error, AmigaDOS replies:

```
Last command failed because object in use
```

While the first `TYPE` command writes in `Datafile1`, the second `TYPE` command cannot read from it. An error message also appears if a process reads a file and then another process tries to open that file for writing. This causes the error message, `CLI error: Unable to open redirection file`, when the output is directed to `Datafile1` while another AmigaDOS window reads from it:

```
TYPE > Datafile1 Datafile
```

This problem doesn't occur very frequently, but it has a special meaning for those that write their own programs and commands. For example, you write a BASIC program that opens an already existing file for reading, and then interrupt the program without closing this file. Then no other process can access this file. Luckily, AmigaBASIC closes all open data files when you are done working with it. A self-written C program should not end under any circumstances without closing all open data files.

Now we come to the last and most important point about working with multiple processes. When you change priorities, if possible, you should not choose a value less than -5 or greater than +4. A value greater than +5 has a higher priority than the `trackdisk.device` which is used for controlling disk access. Programs that are not written well, in regard to priorities, can interrupt the entire system. A program in a multitasking system waiting list cannot be reached through loops or commands. There are operating system routines for this that make such waiting lists possible. Other processes occur through these routines.

8.
Creating AmigaDOS Commands

8. Creating AmigaDOS Commands

The C: directory

The Amiga operating system has so many possibilities that it's impossible to mention all of them. New libraries can be created and by adding additional devices, new hardware can be added. Instead of being integrated into the operating system, AmigaDOS commands are small programs on the Workbench disk. So you can easily add new commands. As an AmigaDOS expert, you have to look in the `C:` directory on the Workbench disk for the AmigaDOS commands. AmigaDOS searches for commands in the current directory and, as a last resort, it looks in the `C:` directory. When the computer is turned on, this directory is assigned to drawer `C:` of the Workbench disk. Each AmigaDOS command can be found this way.

The number of commands is not limited or set. You can copy your favorite AmigaDOS command under as many different names as you want until the disk full requester appears. A somewhat limited use for this capability is to store frequently used commands under a different, shorter name. Example: `X` for `EXECUTE`, `FC` for `FILECOPY`, etc. It's also possible to put user-defined commands into the `C:` drawer. Since a command is basically nothing more than a short program, the clock can be copied here. When the output from the `LIST` command is viewed for the `C:` directory, the clock program stands out because of its high memory requirements. The most frequently used AmigaDOS commands have one thing in common: They are relatively short and can be loaded into memory rather quickly. Such compact code, like that of a true AmigaDOS command, can usually only be created by using an assembler. The language, in which the commands were written in AmigaDOS 1.3, is BCPL, which most people don't have for their Amiga. The C language is an alternative. AmigaDOS 2.0 commands were completely rewritten in C, which greatly reduced their size yet increased their execution speed. When creating new commands you should stick to a particular programming style. You should at least pay attention to the following items:

1. Most AmigaDOS commands are "non-interactive". This means that they don't require information from the user once they are started. The command must be called with all of the correct parameters in a list after the command.

2. The commands normally output their information in the same AmigaDOS window from which they were called. A command should not open its own window for output.

3. True AmigaDOS commands contain an argument template that can be activated by entering the command, a space and a question mark. Should the user enter a false parameter, it responds with a proper argument template. Example:

   a) Argument template

   ```
   Input:      DATE ?
   Output:     TIME,DATE,TO=VER/K:
   ```

   b) Bad arguments

   ```
   Input:  DATE birthday
   Output:
           *** Bad args
           use DD-MMM-YY or <dayname> or yesterday
           etc. to set date
           HH:MM:SS or HH:MM to set time
   ```

8.1 AmigaDOS Commands in C

If the programming is done correctly, a command written in C and an AmigaDOS command cannot be differentiated. It's also possible to pass the parameters directly in the C programming language. The greater-than character allows redirection of the output.

The input line should be read over to make sure that the user entered the correct parameters. When a user enters just any parameters, this shows his/her unfamiliarity with the purpose of the command. In this case the `Bad args` message should appear.

The input line evaluation is programmed in C as follows:

The first `main` function receives the input data in the form of two parameters: The first parameter, which is called `ArgC` (from Argument Counter), is of type `Int` and contains the number of assigned arguments. The name of the program becomes one of these arguments. The second parameter usually has the name `ArgV` (for Argument Vector). It must be declared as a field of type `Char`. The elements of this field contain pointers for the character strings of the input line, which is ended with a null.

This task is not completed from AmigaDOS Shell as you might think. The first line of the main function might look like this:

```
main(argc, argv)
int argc;
char *argv[];
{
...
```

Because `ArgC` and `ArgV` were assigned outside the `main` function, you must declare them as parameters of the desired type before the function bracket.

A brief example should familiarize yourself with this programming technique. We compiled all C programs with an Aztec C® compiler. Using other compilers should not present a problem, if you pay attention to the instruction in the compiler manual.

```
/* Program: Evaluation */

main(argc, argv)
int argc;
char *argv[];
   {
   int i;
```

```
    printf (" Quantity: %d \n",i, argc);
    for ( i = 0; i < argc; i++)
        {
        printf (" Nr.: %d, Argument: %s \n",i , argv[i]);
        }
    }
```

This program is called from AmigaDOS using:

```
Evaluation one and another parameter
```

The following output is received:

```
Quantity: 5

        Nr.: 0 , Argument: Evaluation
        Nr.: 1 , Argument: one
        Nr.: 2 , Argument: and
        Nr.: 3 , Argument: another
        Nr.: 4 , Argument: Parameter
```

A space in the input is interpreted as a separator. How can you assign parameters to a C program from AmigaDOS if the text contains spaces? The simplest method is used very often: The text must be in quotation marks to obtain the desired result. Call our example program with the following:

```
EVALUATION "one and another parameter"
```

Which has the result:

```
Quantity: 2

Nr.: 0 , Argument: EVALUATION
Nr.: 1 , Argument: one and another parameter
```

Until now that's all that the parameter assignment had to do with `CLI` commands. Now you could ask the question, are the > and < characters interpreted as completely normal characters in our program, or couldn't the input or output be directed to any device? A test:

Enter the command:

```
EVALUATION >DF0:Datafile parameter
```

the output is written (directed) to the desired file (`Datafile`). The file contains:

```
Quantity: 2

Nr.: 0 , Argument: EVALUATION
Nr.: 1 , Argument: parameter
```

Trying to redirect the output using the < character fails. The computer doesn't pay attention to which input device is active, but reads the line from the keyboard. In spite of this it's still possible to receive data from other devices. The following example program shows how to do this:

```
/* Redirectinput.c test program */
#include <stdio.h>
FILE * Input(); /* Declaration of an external function */
main(argc, argv)
int argc;
char *argv[];
   {
   int i;
   char Reader;
   FILE *Infile;    /* Pointer for Structure-Type FILE */
   char buffer[100];
   long Length;
   printf (" Number of Parameters: %d \n", argc);
   for ( i = 0; i < argc; i++)
       printf (" Argument: %s \n", argv[i]);
   Infile = Input();
   Reader = Read (Infile, &buffer[0], 30L);
   buffer [Length] = 0;
   printf (" Read: %s\n", &buffer[0]);

   }
```

A small input data file can be created using:

```
echo >datafile "one two three"
```

and the above program named `redirectinput` is started by using:

```
Redirectinput: <datafile hello there
```

will output:

```
Number of Parameters: 3

Argument: redirectinput
Argument: hello
Argument: there
Read: one two three
```

The function from `ArgC` to `ArgV` remains unchanged. The <datafile is completely ignored and that's why `ArgC` confirms the presence of only three arguments.

After the output of the normal parameters, the `Input()` function, which is found in the DOS library, prepares the standard input devices of the called program (also those of AmigaDOS) for our program. The tasks of the variables are:

Infile: Pointer for standard input
Buffer: Contains the characters that are read
Length: Actual number of characters read

Using the < command in AmigaDOS command line switches the standard input device to the given device.

8.2 REPLACE

You probably would like a command that could replace characters or strings in a file. Every good word processor has such a function. There is AmigaDOS `Search` command that searches for characters strings in files, but you can't replace anything with this command. We have created a new AmigaDOS command to do this called `REPLACE`.

A big advantage of this command is that the result is placed in a new file. Should an error occur, the original file is unchanged. This command also allows any characters to be searched for and replaced by any characters. For example, you could replace a carriage return with a space or every space with 100 periods. To allow any characters to be entered, they must be entered in ASCII form. A space is a 32 and a carriage return is a 13. Because it is unusual to enter characters as numbers, you can also enter the character strings. `REPLACE` examines the first character of the input. It takes all of the following characters as ASCII values if the first character is a numeral. Otherwise it takes them as character strings.

Now we come to the call of the new command. It reads:

```
Filenameold Filenamenew String
```

`Filenameold` is the name of the file that should be read. The complete path can be entered.

`Filenamenew` is the name of the new file that should be written.

Note: `REPLACE` does not check if the file exists and also overwrites existing files like all other AmigaDOS commands.

`String` is the string that is searched for and should be replaced. Search and replace strings are separated by a colon and cannot contain spaces. An example of a call could be:

```
Replace Oldfile Newfile Meier:Mayer
```

In the file `Newfile` all `Meiers` would be replaced with `Mayers`. In this case character strings would be entered. `REPLACE` recognizes that neither `Meier` nor `Mayer` starts with a numeral (0-9). Now lets take another example that involves ASCII values. The call:

```
REPLACE Oldfile Newfile 13:32
```

replaces every CR (13) with a space (32). Combining both input possibilities is allowed. The following:

```
REPLACE Oldfile Newfile 32:.....
```

replaces every space (32) with 5 periods. To replace a string of ASCII values, separate the individual numbers with commas. So

```
REPLACE Oldfile Newfile 32,32,32:32
```

replaces three spaces (32,32,32) with one (32).

The length of the search strings is limited to 127 characters, which is enough for normal use. You can create a test file by doing the following:

First build a file with the same characters using:

```
ECHO >RAM:TEST "aaaaaaaaaaaaaaaaaaa"
```

Now replace each a with many new a's using:

```
Replace RAM:TEST RAM:TEST1 a:aaaaaaaaaaaaaaaaaaaaaaaaaaa
```

When you want to remove a character string out of the file, you must not enter anything after the colon. For example:

```
Replace Oldfile Newfile 32:
```

removes all spaces from the file.

The first character differentiates between numeral and character input. So `Replace` would not accept the following input:

```
Replace Oldfile Newfile 3Days:4Hours
```

In this case we should explain the exact operation of the program because this has nothing to do with AmigaDOS. It is structured and documented for all programmers. Our program uses a circular buffer for characters it has read, but not yet analyzed. They are stored in this circular buffer until it can be determined whether the string has been found. Such a string can be utilized in C using the `Modulo` command. We use a more complicated yet faster method. The sub-function for reading new characters checks to see if the end of the buffer is reached and places the valid input at the beginning of the buffer.

```
/* Replace.c This program replaces strings in files  */
/* For 100 K on the  RAMDisk it took  27 Seconds     */
/* ------------------------------------------------- */
/* *************** Copyright Manfred Tornsdorf ***** */
/* ------------------------------------------------- */
#include <stdio.h>

#define FALSE 0             /* Constants, make reading  */
#define TRUE 1              /* the program easier */
#define byte unsigned char
```

```
#define Maxbuffer 512      /* Size of the Read buffer */

 void Shift();
 char SearchString[127];        /* Search string */
 char ReplaceString[127];       /* replacement string */
 char Bufferin[Maxbuffer];       /* read buffer */
 int Pwrite;          /* Place in buffer for writing */
 int Pread;          /* Place in buffer for reading */

 unsigned long Num;  /* Num for characters */
 unsigned int Fromlen;
 unsigned int Tolen;
 unsigned int NumRepl; /* Num for : how often replaced */
 FILE *Rin, *Rout;



/* -------------------- Get Parameters --------------- */
void GetPara(Para, First, Next)
char *Para;
char *First;
char *Next;
{
 int Number;
 int Length;       /* Number for length, 0 not counted */

 *First = 0;
 *Next = 0;
 Length = 0;       /* Length set to 0 */

 Number = atoi(Para);    /* check String or ASCII value */
 if (Number != 0)          /* ASCII-value for String1 */
 {
   while ((*Para != 0)&&(*Para != ':'))
     /* still more  String1 */
   {
      Number = atoi(Para);
      if (Number > 255)          /* number not allowed */
      {
         printf("Number is not ASCII-value\n");
         exit();
      }
      *(First++) = (char) Number;      /* take char */
      Length++;
      while ((*Para != 0)&&(*Para != ',')&&(*Para != ':'))
         Para++;          /* separate search */
      if (*Para == ',') Para++; /* skip , */
   }  /* End While */
   *First = 0;          /* end String1 with 0  */
   Fromlen = Length;       /* save length of String1 */
   }    /* End if( ASCII) */

 else              /* String1 is a String */
 {
    for(Length = 0; (*Para!=0)&&(*Para!=':'); Para++)
    {
```

```
            *(First++) = *Para;
            Length++;
         }
         *First = 0;          /* end search string with  0 */
         Fromlen = Length;       /* save length of string */
      }    /* End else */

   if (Fromlen == 0)
      {
         printf("Search string must be given!\n");
         exit();
      }


   if (*Para != ':')
      {
         printf("Colon missing!\n");
         exit();
      }

   Para++;          /* Set Para after colon */
   Length = 0;          /* Length set to 0 */
   Number = atoi(Para);  /* Check if String or ASCII-value */
   if (Number != 0)          /* ASCII-value for String2 */
   {
      while (*Para != 0)     /* Parameters for String2 */
      {
         Number = atoi(Para);
         if (Number > 255)          /* number not allowed */
         {
            printf("Number is not ASCII-value\n");
            exit();
         }
         *(Next++) = (char) Number;   /* take characters */
         Length++;          /* increase length */
         while ((*Para != 0)&&(*Para != ',')&&(*Para != ':'))
            Para++;          /* separate search */
         if (*Para == ',') Para++; /* skip */
      }  /* End While */
   *Next = 0;          /* end String2 with 0  */
   Tolen = Length;       /* save length of String2 */
   }   /* End if(ASCII)
   else              /* String2 is a String */
   {
      for(Length = 0; (*Para!=0); Para++)
      {
         *(Next++) = *Para;
         Length++;
      }
      *Next = 0;          /* end replace string with 0 */
      Tolen = Length;       /* save length of string */
   }    /* End else */

}

int Reader(Amount)
```

```
int Amount;
{
 register char Charsin;
 register int numread;

 numread = 0;

 for(numread = 0; numread < Amount; numread++)
 {
   Charsin = fgetc(Rin);
   if (Charsin == EOF)
     return(numread);       /* no more chars in file */
   Num++;          /* increase num chars */
   Bufferin[Pread++] = Charsin;
   if (Pread >= Maxbuffer)  /* Buffer at end, overflow */
      Shift();       /* shift to beginning */
 }
 return(numread);
}

void Shift()  /* contents in buffer shifted */
 {
  register int howmany;
  register int i;

  howmany = Pread - Pwrite;
  for (i = 0; i< howmany; i++)
     Bufferin[i] = Bufferin[Pwrite+i];
  Pwrite = 0;              /*  set new Indices */
  Pread = i;
 }

int Checkbuffer() /* compare search string with buffer */
{
 register char *string;
 register int i;

 i = Pwrite;
 for (string = &SearchString[0]; *string != 0; string++)
 {
    if (*string != Bufferin[i])
       return(FALSE);
    i++;
 }
 return(TRUE);
}

void Writecharacter()  /* Write Char in buffer to file */
{
 register char Charsin;

 Charsin = Bufferin[Pwrite];
 fputc(Charsin, Rout);
 Pwrite++;       /* reset write pointer */
}
```

```
void Replace()     /* write replace string, off buffer */
{
 int i;
 char *string;
 register char Charsin;

 i = 1;
 for (string = &ReplaceString[0]; *string != 0; string++)
 {
    Charsin = *string;
    fputc(Charsin, Rout);
    i++;
 }
 Pwrite = Pread; /* write=read pointer -> clear buffer */
}

void Writerest() /* no more char in file, write buffer */
{
 register char Charsin;
 int i;

i = 0;
 for(i = Pwrite; i < Pread; i++)
 {
    Charsin = Bufferin[i];
    putc(Charsin, Rout);
 }
}



main (Amount, Argument)
int Amount;
char *Argument[];
{
 int Numtoread;
 int Totalread;
 int Error;
 if (Amount != 4)
    {
        printf("This program replaces char strings \n");
        printf("Copyright Manfred Tornsdorf\n");
    printf("Call: Filenameold Filenamenew String\n");
 /* next line should be entered on one line in Amiga*/
        printf("either :String =
Number,Number,Number...:Number,Number,Number...\n");
        printf("or     :String = string:string\n\n");
 /* next line should be entered on one line in Amiga */
        printf("Example :Replace t.txt tt.txt
Mytext:32,33,32\n");
        printf("gives   :<Mytext> -> < ! >\n");
        printf("No replace string, so it is deleted\n");
    exit();
        }

 GetPara(Argument[3], &SearchString, &ReplaceString );
```

```
/* build search and replace string  */
 Fromlen = strlen(SearchString);
 Tolen = strlen(ReplaceString);

 Rin = fopen(Argument[1], "r");
 Rout = fopen(Argument[2], "w");
 if (Rin == 0)
 {
 puts ("Input file not found!");
 exit ();
 }
 if (Rout == 0)
 {
 puts ("Cannot open the output file!");
 exit ();
 }
 Num = 0;
 Pwrite = 0;        /* Write pointer to start of buffer */
 Pread = 0;         /* read pointer to start of buffer */
 Numtoread = Fromlen;   /* Search string chars to read */
 NumRepl = 0;       /* Default: not found */

 Totalread = 1;       /* Default, the TC to count */
 while(Totalread != 0)
 {
    Totalread = Reader(Numtoread);
              /* read amount for file */
    Error = Checkbuffer();
    if(Error == FALSE)     /* not found */
    {
       Writecharacter();     /* 1. write chars */
       Numtoread = 1;              /* read one char */
    }
    else              /* found */
    {
      Replace();    /* write replace string, off buffer */
      NumRepl++;        /* increase number replaced */
      Numtoread = Fromlen;
    }
 }   /* End while(), File empty */
 Writerest();         /* Write last char from buffer */

 printf("\nString replaced %d times!\n", NumRepl);
 fclose(Rin);
 fclose(Rout);
 exit();

}
```

8.3 Public Domain and shareware AmigaDOS Commands

If the programming is done correctly, a command written in C and an AmigaDOS command cannot be differentiated. The advantage in this is that a good programmer can add his own commands, and the Amiga community has many excellent programmers. Many of these programmers have written replacement commands for many AmigaDOS commands. Like a new DIR command that displays multiple columns, a command that displays the calendar on the screen and more.

Many of these programmers have placed their creations in the public domain so other Amiga users can enhance their AmigaDOS system. The Amiga is truly and ever expanding and improving system. Public domain program authors choose not to seek formal rights for their programming work and distribute their work free of charge.

Check with your dealer, local user group or public domain software vendor for more information on Amiga Public Domain software. An excellent source is the public domain library of the Champaign-Urbana Commodore Users Group, Inc., P.O. Box 716, Champaign, IL 61824-0716. Their CUCUG CUSTOM WORKBENCH disk is one of the best collections of AmigaDOS utilities. The CUCUG Custom Workbench is an enhanced AmigaDOS environment that makes you more productive when working with your Amiga. This disk contains the best "tricks of the trade" and some excellent PD and shareware WB enhancements. Contact CUCUG directly for more information on ordering this excellent disk.

The following public domain programs are among the favorites of the Abacus editing staff. These public domain programs can be found on many of the telecommunication networks and from many Amiga user groups.

Command	Description
CAL	Displays calendar of specified month and year.
C-SHOW	IFF files viewer for script files.
DD	Multiple column directory display.
IFFICON	Converts IFFs to icons and icons to IFF format.

Command	Description
INFORM	Expanded INFO command.
LS	Excellent LIST replacement.
PRFONT	Prints out and displays all the fonts in your FONTS: directory.
PRINT	Excellent file printer.
SUB	Subliminal messages on your Amiga.
WHENCE	Shows where a program is located.
ZIPPY	Fast little directory program.

Shareware programs

This type of program is slightly different than the public domain program. These programs are sold directly by the authors, on a "try it, then buy it" basis. You can legally obtain a copy from a friend or make copies for others. This is how the programs are distributed. However, if you end up using the program regularly you should register it with the program developer by paying the registration fee. Registered users may receive documentation, updates and technical support from the author. Many of these programs can be found on the companion diskette available for this book in the Shareware-PD directory. See the README file in the Shareware-PD directory for a complete list.

Command	Description
dcCLOCK	Excellent clock and calendar program.
FILETYPE	Show file type.
FILEDFIND	Finds file on disks.
LISTCOM	Lists files by comments.
PRINT	Prints files easily.
SOUND	Plays and AMiga IFF sound file.
UDATE	Check file updates.
XICON	Adds icon execution to script files.

9.
ARexx

9. ARexx

This chapter contains information about ARexx for the Amiga. The three sections of this chapter explain ARexx, all ARexx commands and present some very usable ARexx application programs.

If you think about it, it's almost silly. Your Amiga word processor is up and running and, thanks to the magic of Amiga multitasking, right there on the same screen is a database program displaying an address. You'd like to insert that address into the header of the letter you're typing. So why should you have to read the address and type it in by hand? Why can't the word processor just get the address directly from the database program and insert it into your document automatically? Isn't your Amiga even aware of its own processes? Can't it find data that's displayed on its own screen? Truth is, though the Amiga can run a multitude of programs simultaneously, until recently the only "official" means built into the Amiga's operating system for one program to communicate with another program has been the usually misunderstood and vastly underused Clipboard Device.

What's required is *inter-process communication*, a standard that defines how one program (or process) can talk to another. On the Amiga, this capability is provided by a language called ARexx. Though it has been available as a third-party product since 1987, ARexx can now be considered an official standard because it comes installed on version 2.0 of the Amiga's Workbench operating system.

Developed by William S. Hawes, ARexx is an Amiga implementation of the REXX language previously available on IBM systems. ARexx is a full-fledged programming language, similar in many ways to BASIC. But more than that, ARexx also supplies the Amiga with a new system-level communications protocol that works similar to a telephone exchange, detecting and directing calls and passing information and commands from one program to another. ARexx also serves as a standardized macro language for all Amiga programs that are equipped to take advantage of it. So ARexx actually serves three very useful functions.

9.1 RUNNING AREXX

Just as you must open a CLI window or load the Amiga's Workbench operating system before you can use them, ARexx must first be loaded into the Amiga environment before it can be used. This is a simple process accomplished by adding the command "rexxmast" to your startup-sequence, typing it from the CLI, or clicking on its icon. Provided the proper commands and libraries have been copied to your system's LIBS: and C: directories (there is an "install" icon that will do this for you for v1.3, and it comes pre-installed on v2.0 system disks), the ARexx system is quickly and transparently installed in the background, taking up about 40K of RAM. Once invoked, it simply waits in the background for one program or process to try to communicate with another, or to be invoked directly to run a stand-alone ARexx program.

ARexx works through invisible communications "ports" that are opened by ARexx-compatible programs. ARexx cannot simply open its lines of communication with all Amiga programs automatically; the programmer must add some special code which interfaces with the ARexx system. It's similar to having to installing a telephone before you can call someone. The extra code involved is simple (most of the work is done by ARexx itself), and adds only about one to two kilobytes of code to an average program. Once a programmer has added an ARexx interface to an application, that program has the capability to communicate with any other ARexx-capable program running on the Amiga, or with ARexx itself. There are already hundreds of Amiga programs that support ARexx, and now that Commodore is officially supporting ARexx, most new Amiga programs are sure to include ARexx interfaces.

The ARexx system comes with several helpful support programs. HI sends a halt request to all running ARexx applications. This is useful if you've accidently written a program that gets stuck in an infinite loop. RX runs stand-alone ARexx programs. RXSET adds or changes entries in ARexx's 'Clip List', which is used to hold and pass values. RXC closes down ARexx completely after the last ARexx program is finished running. WaitForPort is used to check and see if a particular program's ARexx communications port is available. TCC, TCO, TE, and TS control ARexx's powerful TRACE debugging tools.

9.2 AREXX PROGRAMS

ARexx is an interpreted programming language, providing the same type of immediate feedback and instant gratification as BASIC. ARexx can even be accessed from the CLI in an "immediate mode" similar to the Commodore 64's built-in BASIC interpreter. For example, you can type:

```
rx "do 5;say 'Hi';end"
```

from the CLI, and ARexx will print the word "Hi" five times. The initial "rx" is always required; it invokes the ARexx interpreter. The rest of the line is an actual ARexx program consisting of ARexx commands surrounded by quotes and separated by semicolons. This particular example is the ARexx equivalent of the BASIC program:

```
FOR x=1 TO 5:PRINT "Hi":NEXT
```

It sets up a loop which prints the word "Hi" in the CLI window five times, then quits. More complex programs can be created by using a text editor and saving them as text files, just as you would create an AmigaDOS script file. Entering:

```
rx dh0:Sample
```

from the CLI will run an ARexx program named "Sample" (or "Sample.rexx" if "Sample" is not found) on drive dh0:. The pathname is not required if your program is located in the current directory, or in a directory that has been assigned the virtual device name REXX:.

Since ARexx is an interpreted language, it is not as fast as a compiled language like C or Modula-2. Most people will not want to write large application programs in ARexx. However, the current version of ARexx is as much as three times faster than the original release, and should be more than adequate for simple programming tasks, such as text formatting and file maintenance. ARexx is, of course, capable of more complex tasks; there is even a public domain "Star Trek" game written completely in ARexx.

9.3 PROGRAM MACROS

Although ARexx can function quite well as a stand-alone programming language, it really shines when you invoke its ability to communicate with an application program.

ARexx can act as a versatile macro language for any program that is ARexx capable. ARexx macro programs can be specifically written to perform complex tasks such as automatically inserting formatting codes into a word processor document, or sending control scripts to a telecommunications program for automated on-line sessions. ARexx can pass commands to running programs, making menu selections, typing in filenames, and making choices based on changing conditions. ARexx macros can be quite complex, automating hundreds of operations if necessary.

An ARexx macro is usually a combination of actual ARexx commands and specialized instructions for the program involved. For example, an ARexx macro for a graphics file conversion program like ASDG's *Art Department Professional* might use ARexx commands to determine whether or not a file exists on disk, then use the program's specialized instructions to instruct the program to convert that file from a HAM to a 32-color image, saving the result to disk. Then ARexx commands might take over and rename or move the file to a different drawer, while deleting the original file.

ARexx macros can be very simple (e.g., inserting a single control character at the cursor position in a word processing document) or quite complex (e.g., perhaps opening an application program, making multiple changes to a disk full of files, printing them out, then closing the program).

9.4 MULTITASKING

By far the most powerful feature of ARexx is its ability to function as a communications link between two or more running applications. For example, you might want your word processor to output information to an ARexx program where it is sorted and formatted, then inserted automatically into a database manager. Or you might ask your telecommunications program to download stock quote information and transfer it directly to a spreadsheet. You can even link multiple programs, passing information back and forth among a database, a word processor, a spreadsheet, and a telecommunications program simultaneously. With the proper ARexx programs as "glue" between your applications, there is really no limit to what you can accomplish.

Best of all, ARexx can even make decisions based on parameters, such as data values, time and date, disk space available, or system configuration, so that it can pass commands and process data selectively. It can even run programs or execute necessary AmigaDOS commands under program control. It is possible to automate an entire computing session with the right ARexx program, launching programs, handling errors, and making decisions based on pertinent data in an almost human manner. Your imagination is really the only limit. All that is required is this: each program with which you want to communicate must open its own ARexx communications port.

9.5 HOW AREXX WORKS

9.5.1 DATA

ARexx is similar to other computer programming languages, such as BASIC, C, and Modula-2. But there are also some differences. One major difference is that ARexx variables are *typeless*. This means that a variable may contain a text string or a numeric value without first being declared as one or the other. You can then use the contents of the variable in any way that is legal. For example, an ARexx program might make this declaration:

```
number = '1234567'
```

Because of the quotes surrounding the assignment string, most languages would consider the variable `number` to contain a string of characters, as does ARexx. But since the string is also a valid numeric value, ARexx will also consider `number` a valid operator for numeric calculations. If you type in this line:

```
say number + 2
```

ARexx will respond with the correct answer, '1234569'. But you can also perform string functions. For example:

```
say left(number,4)
```

will result in the response, '1234'. It is very unusual for a programming language to treat its variables in such a nonchalant manner, but this capability gives ARexx a great deal of power in the manipulation of numbers and strings.

Numeric operators in ARexx are always expressed as decimals. Binary numbers can be expressed as a string of ones and zeroes enclosed in quotes and followed by a 'b', as follows: '10101100'b. Hexadecimal numbers can be expressed as a string of digits 0-9 and letters A-F, enclosed in quotes and followed by an 'x', as follows: 'CD10'x. Hexadecimal and binary strings can be converted to decimal numbers using the ARexx conversion function C2D(). `X=C2D('1011011'b` will return a value of decimal 91 for X. `X=C2D('CD10'x)` returns a value of decimal 52496 for X.

9.5.2 SYMBOLS

ARexx works with four different types of data, which it calls *symbols*. In other languages, they would be called constants and variables.

The first is a *fixed* symbol, which is comprised of the digits 0-9, a decimal point, an exponential symbol 'E', and/or a leading '+' or '-' sign. These all make up constant values, such as 9, 3.60, .123, -74, or 1.43E+5.

The second is a *simple* symbol, which would be called a simple variable in most languages. Simple symbol names cannot begin with a number and cannot include a period. `Hello`, `number`, and `x` are all examples of simple symbols. However, if a simple symbol has not been assigned a value, it is assigned a string value equivalent to its name that appears in uppercase characters. (If the variable `hello` has not been assigned a value, `Say hello` responds with 'HELLO'.)

```
say hello       /* ==>  'HELLO'            */
hello = 123     /* declare variable value */
say 'hello'     /* ==>  'hello'            */
say hello       /* ==>  '123'              */
```

This example also illustrates how ARexx defines a literal string: any collection of characters (including the keyboard's international character set) that is enclosed in single ('Hi') or double ("Hi") quotes is interpreted as a literal string.

The third type of symbol is a *stem*, which would be called an array or a subscripted variable in most other languages. A stem symbol has a name like `number.`, with a single period at the end of its name. The name before the period is the stem name of what will be extended to become the fourth type of ARexx symbol, the *compound* symbol. One or more extension names are added after the period, which act as indexes to the stem. Here's an example:

```
number. = 10        */ initialize all values at '10' */
do i = 1 to 5       */ do loop five times            */
  number.i = i      */ assign value equal to index   */
  end               */ end loop                      */
```

This program assigns the value '1' to the variable 'number.1' the first time through the loop. On subsequent passes, it assigns '2' to 'number.2', '3' to 'number.3', and so on up to five. If we then ask ARexx to `say number.4`, it will respond with '4', the value stored in the symbol 'number.4'. However, the first program line initialized all incidences of the stem 'number.' with the value '10'. So if you ask ARexx to `say number.7`, it will answer back '10'.

A compound symbol can also include multiple extensions (e.g., 'number.ind.mark'). Both 'ind' and 'mark' can represent simple symbols which index into the compound data structure. This example also illustrates the fact that extensions don't have to be numeric.

9.5.3 OPERATORS

ARexx uses four different types of data operators.

Arithmetic operators include the familiar 'four-banger' calculator functions of addition '+', subtraction '-', multiplication '*', and division '/'. As in most languages, multiplication and division are performed first in a calculation, then addition and subtraction. Parentheses '()' can be used to force a particular calculation order. For example, `say 3*2+10/5` will return the result '8' because 3*2 and 10/5 will be calculated first and then added together. However, `say (3*(2+10))/5` will return '7.2'. Exponentiation is performed with a double asterisk '**'; 3**2 is interpreted as 'three squared'. Exponents must be integers, even though they may be negative. Exponentiation takes precedence over all other operators. The final two arithmetic operators are closely related. Integer division '%' returns the integer portion of a division operation; `say 22%5` returns '4' because 5 goes into 22 only four times, with a remainder of 2. Remainder, or 'modulo', division '//' performs the same operation, but returns the remainder portion of the answer; `say 22//5` returns '2'.

Concatenation operators join two or more strings. The only explicit concatenation operator in ARexx is the double vertical bar '||'. `say 'hello'||'world'` returns the result 'helloworld', with no space between the strings. Simply typing the command `say 'hello' 'world'` with one or more spaces between the strings results in the output 'hello world', with a single space separating the strings. Typing `say 'hello''world'` without an intervening space causes ARexx to interpret the doubled quote as meaning "I want to print a quote mark here", so it responds with 'hello'world'. However, if a variable name is first defined and included, one or more strings can be 'jammed together', as in this example: `x = 'world';say 'hello'x`. This command returns the response 'helloworld', just as though the explicit concatenation operator had been used.

Comparison operators compare two numbers or strings. *Exact* comparisons look for complete equivalence. The two exact comparisons are '==', exactly equal, and '~==', exactly unequal. *String* comparisons ignore leading blanks, and pad the shorter string with trailing blanks if necessary. *Numeric* comparisons convert operators to numeric form, using ARexx's current NUMERIC DIGITS setting. The numeric and string comparisons are equal '=', not equal '~=', greater than '>', greater than or equal to '>=' or '~<', less than '<', and less than or equal to '<='

or '~>'. Note that the tilde '~' is used to negate a comparison. For example, '<=' is 'less than or equal to', while '~>' is technically 'not greater than', which is logically equivalent.

Logical, or 'boolean', operators always return a value of TRUE '1' or FALSE '0'. The four logical operators are NOT '~', AND '&', OR '|', and XOR (exclusive OR) '^' or '&&'. You should remember that the caret or up-arrow sign '^' is used by ARexx not for exponentiation, but as the XOR logical operator.

9.5.4 PROGRAMS

A short ARexx program can be entered from the CLI as a single line, as we illustrated previously:

```
rx "do 5;say 'Hi';end"
```

This simple program actually consists of three ARexx commands separated by semicolons. Semicolons can always be used to allow multiple commands on a single line. Longer ARexx programs must be entered using a text editor or word processor. ARexx programs entered in this way must begin with a comment line. The comment line is ARexx's signal that what follows is, in fact, an ARexx program, and not just a text file. A comment is any line preceded by the characters '/*' and followed by '*/'. A comment may take up more than one line, or may follow a command statement on its line. The following are examples of legitimate ARexx comments:

```
/* This is the opening comment line which is REQUIRED*/
/*   by ARexx This comment spans several lines, but
     doesn't require new opening and closing slash,
     asterisk marks. ARexx considers everything
     following the opening comment marks a comment
     until it encounters the closing mark:  */
say 'Hi There'  /* This comment follows an executable
                    command */
```

Beyond the opening comment line, it is good programming practice to use comments liberally throughout your programs. They improve readability, and will help you debug or modify your programs later, if needed.

9.5.5 COMMANDS & FUNCTIONS

Since the main purpose of ARexx is to serve as a utility language, it is especially adept at handling text and numbers. ARexx includes specific text-handling functions to perform such esoteric tasks as centering a line, stripping out substrings, deleting words, and even reversing a string of characters. ARexx number-handling functions have the ability to convert among decimal, hexadecimal, and binary, find the maximum or minimum in a list, and even set the number of digits of precision in ARexx math operations. (This function is called "FUZZ".)

Of course, ARexx is also equipped with a full complement of program control functions. Our little one-line example demonstrated an elementary DO loop; SELECT-WHEN-OTHERWISE and IF-THEN-ELSE conditionals are also supported, in addition to others.

A list of the ARexx commands can be found at the end of this chapter. You should refer to this list as you encounter various commands in the examples.

ARexx programs can even issue AmigaDOS commands or launch other programs, so they are really limited only by the imagination of the programmer. Advanced programmers can make use of a full range of highly sophisticated ARexx commands that control interrupts, alter the system configuration, and even access machine addresses. They can also call support libraries that allow opening screens and windows and creating gadgets.

9.5.6 PURE POWER

It should be obvious by now that ARexx enhances the Amiga's already impressive multitasking capabilities. It allows programs to actually control each other, passing data and commands back and forth with a minimum of human input. ARexx is even perfectly capable of running an Amiga entirely by itself if a properly constructed control program is provided.

Since ARexx is now an official standard, Amiga developers are more likely to take advantage of the new capabilities ARexx provides. If nothing else, it makes a standard macro language available to every ARexx-capable program. And as programmers experiment, they are bound to come up with new and innovative ways for programs to share important data easily, and even to control each other. The coming years should see the development of many multitasking innovations brought about by the existence of the ARexx standard.

ARexx v1.15 is shipped on the v2.0 Workbench release from Commodore. It is also available for $49.95 from: William S. Hawes, PO Box 308, Maynard MA 01754, 508-568-8695, BIX: whawes, CIS: 72230,267, PLINK: whawes

9.6 AREXX COMMANDS & FUNCTIONS

9.6.1 FLOW & CONTROL

DO

Defines a group of instructions to be performed as a block. DO incorporates many control structures. Any combination of controls may be used, with the exception that WHILE and UNTIL are mutually exclusive.

Example:

```
DO                          /*  Execute block of instructions once. */
  instructions...           /* Often used with IF...THEN...ELSE.    */
  END                       /* All DO loops end with END.           */

DO i                        /* Does loop 'i' number of times.       */
  instructions...
  END

DO i = lo TO hi BY index    /* Iterative loop. Advances 'i' from    */
  instructions...           /* 'lo' to 'hi' values by increment     */
  END                       /* 'index'. BY is optional, default = 1 */
                            /* and index may be negative or decimal.*/

DO i = lo TO hi FOR limit   /* Same as above, but will not exceed   */
  instructions...           /* value of 'limit'.                    */
  IF j=k THEN ITERATE       /* Skips rest of loop, increments i.    */
  IF k>0 THEN LEAVE         /* Exit iterative loop if condition=TRUE*/
  instructions...
  END

DO WHILE i < limit          /* Performs loop until value of 'i'     */
  instructions...           /* equals or exceeds value of 'limit'.  */
  END                       /* Evaluated at start of loop, so loop  */
                            /* may not be executed at all.          */

DO UNTIL i > limit          /* Like WHILE, but condition is checked */
  instructions...           /* at the END of a loop, so the loop is */
  END                       /* always executed at least once.       */

DO FOREVER                  /* Execute forever.                     */
  if i>6 then BREAK         /* Exit loop if condition is met.       */
  END
```

EXIT

Halts execution of an ARexx program, optionally returning a value.

Example:

```
IF x>6 THEN EXIT            /* Halts execution.                    */

IF x>6 THEN EXIT 12         /* Halts execution & returns value '12' */
```

IF-THEN-ELSE

Conditionally controls program statement execution. An IF-THEN statement may consist of a single program line, or multiple lines through inclusion of a DO loop.

Example:

```
IF x>5 THEN SAY x           /* If condition is true, print x.      */

IF x>5 THEN                 /* Multi-line loop is executed only if  */
  DO                        /* condition is met.                    */
    SAY x
    x = x+1
    END
  ELSE SAY 'Done'           /* ELSE is performed only if condition  */
                            /* is NOT met. May be multi-line with DO*/

IF x>6 THEN                 /* In the case of double IF statements  */
  IF y<4 THEN               /* it may be necessary to bind an ELSE  */
    SAY x                   /* to an inner loop using NOP, or 'No   */
    ELSE NOP                /* Operation'. This allows the real ELSE*/
  ELSE SAY y                /* to work as it's supposed to.         */
```

SELECT

Allows multiple-choice type program execution.

Example:

```
SELECT                      /* Begin block.                         */
  WHEN i=1 THEN SAY i       /* If condition is met, say value of i. */
  WHEN j<3 THEN             /* If condition is met, then DO block   */
    DO                      /* of instructions.                     */
      j=j+6
      SAY j
    END
  OTHERWISE SAY 'Sorry!'    /* If no conditions are met, give a     */
  END                       /* response. End with an END statement. */
```

SIGNAL

Essentially a special-purpose GOTO statement, SIGNAL passes control to label points in a program depending on various conditions. Valid options for SIGNAL are BREAK_C, BREAK_D, BREAK_E, BREAK_F, ERROR, FAILURE, HALT, IOERR, NOVALUE, and

SYNTAX. If a SIGNAL condition is met, a special variable called SIGL is set to the line number which was executing when the condition was met.

Example:

```
SIGNAL ON BREAK_C           /* If a 'CTRL-C' is detected from the   */
                            /* keyboard, control passes to the point*/
                            /* in the program labeled 'BREAK_C:'    */

SIGNAL OFF BREAK_C          /* Disable check for 'CTRL-C'.          */

SIGNAL 'start'        /* The program jumps to the label START:      */
```

9.6.2 STRINGS

ABBREV()

Returns a boolean value indicating whether one string is an abbreviation of another, with an optional specified minimum length.

Example:

```
say ABBREV('abcde','abc')   ==> 1 /* True  */

say ABBREV('abcde','abc',4) ==> 0 /* False, because 'abc'<4 chars   */
```

CENTER()

Centers a string in a string of the specified length, trimming it to size or padding it with blanks or a specified pad character if needed. May also be expressed as the British CENTRE().

Example:

```
x=CENTER('abc',6)   ==> ' abc  '  /* Extra spaces go to right.  */

x=CENTER('abc',6,'+')  ==> '+abc++'

x=CENTRE('abcdef',3)   ==> 'bcd'
```

COMPRESS()

Removes leading, trailing, and imbedded spaces from a string. Can also remove specified characters from a string.

Example:

```
x=COMPRESS('  a  bc  ')  ==> 'abc'      /* Removes blanks.       */

x=COMPRESS('=+/a/bc=','/=')  ==> '+abc' /* Removes '=' and '/'. */
```

COMPARE()

Compares two strings and returns the position of the first character which doesn't match, or 0 if they're identical. Pads the shorter string with spaces, or a pad character if specified.

Example:

```
x=COMPARE('abcde','abcce')  ==> 4   /* 4th char doesn't match   */

z=COMPARE('abc+-','abc','+') ==> 5  /* Short string is padded   */
                                    /* with '+', so 5th char is */
                                    /* first 'no match'.        */
```

COPIES()

Returns a string composed of the specified number of copies of the supplied string.

Example:

```
x=COPIES('abc',4) ==> 'abcabcabcabc'
```

DELSTR()

Deletes a substring from the specified position to the end of the string, or for a specified number of characters.

Example:

```
x=DELSTR('abcdefg',4)  ==> 'abc'     /* Begins at 4 */

x=DELSTR('abcdefg',4,2)  ==> 'abcfg' /* Begins at 4 for 2 chars */
```

DELWORD()

Similar to DELSTR(), but deletes a specified number of words beginning at the indicated word number. If no length is specified, it deletes the rest of the string.

Example:

```
z=DELWORD('I am a nun',3) ==> 'I am'  /* Start at word 3 */

z=DELWORD('I am a nun',3,1)  ==> "I am nun' /* 1 word, start at 3 */
```

FIND()

Finds a phrase of words in a larger string, and returns the word number of the matched position. Returns 0 if not found.

Example:

```
z=FIND('He is the champ','is the')  ==> 2
```

INDEX()

Searches for the first occurrence of the specified pattern in a string, optionally starting at a specified position. Returns the position first matched, or 0 if not found.

Example:

```
z=INDEX('abcdabcdef','cd')  ==> 3   /* First position found.    */

z=INDEX('abcdabcdef','cd',5) ==> 7  /* Started at position 5.   */
```

INSERT()

Inserts the first string into the second string. Begins at position 0, or after a specified starting position. Either string may be padded to a specified length if required, and the first string may be truncated if necessary.

Example:

```
z=INSERT('ab','abcdef',4) ==> 'abcdabef'    /* Insert at position 4 */

z=INSERT('abcd','abcdef',4,3)  ==> 'abcdabcef' /* Insert after 4,    */
                                   /* truncate inserted string to   */
                                   /* 3 characters.                 */

z=INSERT('ab','abcde',6,4,'+') ==> 'abcde+ab++'  /* Pad string to 6 */
                                  /* chars using '+', then insert   */
                                  /* first string, padding it to 4  */
```

LASTPOS()

Search backwards for the first occurrence of the first string in the second, beginning at the optional start position. Similar to INDEX().

Example:

```
z=LASTPOS('cd','abcdabcdef')  ==> 7   /* Last position found.    */

z=LASTPOS('cd','abcdabcdef',5) ==> 2  /* Started at position 5.   */
```

LEFT()

Returns the specified number of leftmost characters of a string. Pads with an optional pad character if necessary.

Example:

```
z=LEFT('abcdef',4) ==> 'abcd'
```

```
z=LEFT('abc',6,'+') ==> 'abc+++'
```

LENGTH()

Returns the length of the string.

Example:

```
z=LENGTH('123456')  ==> 6
```

OVERLAY()

Overlay the first string on the second, beginning at an optional starting position (default is the first character) for an optional length (default is the length of the first string). If necessary, pad with blanks or an optional pad character.

Example:

```
z=OVERLAY('abc','123def')  ==> 'abcdef'

z=OVERLAY('abcdef','123456',4,7,'+') ==> '123abcdef+'  */ Begin at */
                                    /* position 4, for 7 chars, pad '+' */
```

POS()

Searches for the first occurrence of the specified pattern in a string, optionally starting at a specified position. Returns the position first matched, or 0 if not found. Identical to INDEX() except for the order of the arguments, so be careful.

Example:

```
z=POS('cd','abcdabcdef')  ==> 3   /* First position found.    */

z=POS('cd','abcdabcdef',5) ==> 7  /* Started at position 5.   */
```

REVERSE()

Reverses the sequence of characters in a string.

Example:

```
SAY REVERSE('abcdefg')  ==> 'gfedcba'
```

RIGHT()

Returns the rightmost specified number of characters from a string. If the requested length is longer than the length of the string, RIGHT() will pad it on the left with blanks or a specified pad character.

Example:

```
z=RIGHT('abcdef',3)  ==> 'def'

z=RIGHT('abc',7,'+')  ==> '++++abc'
```

SPACE()

Reformats a string so there are the specified number of spaces (or optionally specified pad characters) between each pair of words.

Example:

```
z=SPACE('Hi there buddy',3) ==> 'Hi   there   buddy'

z=SPACE('Hi there buddy',0) ==> 'Hitherebuddy'

z=SPACE('Hi there buddy',2,'+') ==> 'Hi++there++buddy'
```

STRIP()

Removes extraneous leading or trailing spaces, or both (default), using the options 'L','T', or 'B'.

Example:

```
z=STRIP('   abc   ','T')  ==> '   abc'
```

SUBSTR()

Returns a substring of the supplied string, starting at the specified position, for the rest of the string or for an optional specified length, padding with blanks or an optional pad character.

Example:

```
z=SUBSTR('123456',3,3)  ==> '345'

z=SUBSTR('123456',5,4,'+')  ==> '56++'
```

SUBWORD()

Returns a substring of the supplied string, beginning at the specified word and continuing on to the end of the string or for an optional specified length in words. Leading and trailing blanks will be stripped.

Example:

```
z=SUBWORD('Hi there good ole buddy',3,2) ==> 'good ole'
                        /* Begin at word 3 for a length of 2 words */
```

TRANSLATE()

Translates selected characters in a string to certain other selected characters, according to an output and an input table. If no output and input tables are given, TRANSLATE() simply makes the string all uppercase.

Example:

```
z=TRANSLATE('abcdef','123','abc') ==> '123def'  /* 'a' is replaced  */
                                  /* by '1', 'b' by '2', and 'c' by '3' */
```

TRIM()

Removes trailing blanks from the supplied string.

Example:

```
z=TRIM('abcdef   ')  ==> 'abcdef'
```

UPPER

Translates a string to uppercase. There is also an UPPER() function.

Example:

```
SAY UPPER "Hi There"  ==> "HI THERE"

z=UPPER("Hi There")
```

VERIFY()

Compares a string against a list of characters, returning the index of the first character in the string argument which is *not* in the list, or 0 if all characters are in the list. If the optional keyword MATCH is included, VERIFY() returns the index of the first character in the string which *is* in the list.

Example:

```
z=VERIFY('hello','leh') ==> 5  /* 'o', in position 5, is the first */
                               /* unmatched character.             */

z=VERIFY('hello','leh','M') ==> 1  /* 'h', in position 1, is the  */
                                   /* first matched character.    */
```

WORD()

Returns the specified number word in the string, or a null string if there aren't that many words in the string.

Example:

```
z=WORD('Hi there old man',3) ==> 'old'
```

WORDINDEX( )

Returns the starting position of the specified number word in the string, or 0 if there are fewer words than the number specified.

Example:

```
z=WORDINDEX('Hi there old man',3) ==> 10 /* Word 3 starts at pos. 10
*/
```

WORDLENGTH( )

Returns the length of the specified number word in the string.

Example:

```
z=WORDLENGTH('Hi there old man',3) ==> 3 /* Word 3 is 3 chars. long
*/
```

WORDS( )

Returns the number of words in the string.

Example:

```
z=WORDS('Hi there old man') ==> 4
```

XRANGE( )

Creates a string composed of all the characters numerically between the specified start and end values. If no start and end are specified, XRANGE() creates a string of all values from decimal 0 to 255 (hex '00'x to 'FF'x).

Example:

```
z=XRANGE('a','f')  ==> 'abcdef'  /* Try SAY XRANGE(' ','z')     */
```

9.6.3 NUMBERS

ABS()

Returns the absolute value of a numeric argument.

Example:

```
x = ABS(y)              /* If y=-5.6, then x=5.6 */
```

MAX()

Returns the maximum of a list of arguments. See also: MIN().

Example:

```
z=6;x=MAX(1,5.4,z,3+2)  ==> 6
```

MIN()

Returns the minimum of a list of arguments. See also: MAX()

Example:

```
z=6;x=MIN(1,5.4,z,3+2)  ==> 1
```

RANDOM()

Returns a pseudo random positive integer between a minimum and a maximum value, up to 999. May be supplied with a seed if desired.

Example:

```
z=RANDOM(1,99,TIME('s'))    /* Seed from 'seconds' timer.      */
```

RANDU()

Like RANDOM(), but always generates a number between 0 and 1. May be given an optional seed value.

Example:

```
z=RANDU(TIME('s'))          /* Seed from 'seconds' timer.      */
```

SIGN()

Returns 1 if the argument is >=0, -1 if the argument is <0.

Example:

```
SAY SIGN(-5) ==> -1
```

TRUNC ( )

Returns the integer part of the argument followed by the specified number of decimal places (default 0), padded with zeroes if necessary.

Example:

```
z=TRUNC(123.456,1) ==> 123.4
```

9.6.4 INTER-PROCESS COMMUNICATION

ADDRESS

ARexx keeps track of two host addresses to which it can send commands, the 'current' and 'previous' address. ADDRESS is used to swap the two, or to specify a different address permanently or temporarily. (In ARexx host names, case *is* significant, so 'EDIT' and 'edit', for example, identify two *different* hosts.)

Example:

```
ADDRESS                     /* Swaps 'current' and 'previous'.      */

ADDRESS 'host' 'com'        /* Issues the command 'com' to the      */
                            /* ARexx host named 'host' without      */
                            /* changing 'current' or 'previous'.    */

ADDRESS 'host'              /* Makes 'host' the 'current' host      */
                            /* and swaps 'current' to 'previous';   */
                            /* the former 'previous' is lost.       */

ADDRESS VALUE a b c...      /* Same as above, but evaluates the     */
                            /* host name from the terms.            */

ADDRESS COMMAND 'dir'       /* A special address; passes a command  */
                            /* to AmigaDOS.                         */
```

ADDRESS ( )

Returns the current host address string. Tells you where commands will be sent.

Example:

```
x=ADDRESS()                 /* If the current host is 'TxEd', then  */
                            /* x='TxEd'                             */
```

PUSH

Prepares a stream of data for the standard input stream, which can then be read by any program which looks for keyboard input. Stacked commands are read out in a "last-in, first-out" format.

Example:

```
PUSH 'dir'                  /* Push 'dir','list', and 'cd' to SIDIN */
PUSH 'list'                 /* stream. Commands will be read out in */
PUSH 'cd'                   /* reverse order: 'cd','list', 'dir'    */
```

QUEUE

Same as PUSH, but commands are stacked in "first-in, first-out" order.

SHELL

Synonymous to ADDRESS.

9.6.5 FILES

CLOSE( )

Closes the file specified by the given logical name. All files are closed automatically when an ARexx program exits.

Example:

```
CLOSE('output')
```

EOF( )

Checks the specified logical file and returns a boolean TRUE '1' if the end of file has been reached. Otherwise returns FALSE '0'.

Example:

```
DO UNTIL EOF('infile')      /* Perform loop until end of file.  */
   s = READLN('infile')
   END
```

EXISTS( )

Returns a boolean value (1 or 0) indicating whether a given AmigaDOS file exists.

Example:

```
SAY EXISTS('df0:ReadMe.txt')  ==> 1 or 0
```

OPEN()

Opens an AmigaDOS file for READing, WRITEing, or APPENDing, giving it an ARexx logical filename to which it will be referred by all other ARexx file operations. READ, WRITE, and APPEND may be abbreviated by their first letters. Also may be used to open a custom console window. Returns a boolean value indicating whether the file opened successfully. Though open files are closed automatically when an ARexx program ends, you can close them with the CLOSE() function.

Example:

```
z=OPEN('infile','df0:ReadMe.txt',R)  /* Open 'infile' for READ  */

OPEN('Window','CON:160/50/320/100/Window/cds') /* Open custom CLI */
```

READCH()

Reads the requested number of characters from the named logical file.

Example:

```
z=READCH('infile',32)
```

READLN()

Similar to READCH(), but reads a complete line from the logical file up to the next newline character.

Example:

```
z=READLN('infile')
```

SEEK()

Moves to a new position in the named logical file. The position can be relative to the file's BEGIN, CURRENT, or END. The returned value is the actual position relative to the start of the file.

Example:

```
z=SEEK('infile',43)      /* Move ahead 43 characters from CURRENT. */

z=SEEK('infile',35,'B')  /* Move to 35th character from BEGIN.     */

SAY SEEK('infile',0,'E') /* Returns length of file to END.         */
```

WRITECH()

Writes the string to the specified logical file. Returns the actual number of characters written. Does *not* write a newline character.

Example:

```
z=WRITECH('outfile',"Hi there") ==> 8  /* Wrote 8 characters.       */
```

WRITELN()

Same as WRITECH(), but adds a newline character to the end of the line.

Example:

```
z=WRITELN('outfile',"Hi there") ==> 8  /* Wrote 8 characters.       */
```

9.6.6 CONSOLE I/O

ARG

Shorthand for PARSE UPPER ARG. Gets input from the console.

Example:

```
ARG a b c .                 /* If program was run as            */
                            /*   rx program 12 15 howdy         */
                            /* then a=12, b=15, c='HOWDY'       */
```

ECHO

Synonymous to SAY.

PARSE

Extracts substrings from a string (or the console input) and assigns them to variables. PARSE has many more options than it is possible to illustrate here. (For example, the options NUMERIC, SOURCE, and VERSION can be used to check various bits of system information). Check your Amiga or ARexx manual for complete information.

Example:

```
PARSE UPPER ARG x y z       /* Synonymous to ARG. Optional      */
                            /* keyword UPPER converts to uppercase*/

PARSE UPPER PULL x y z      /* Synonymous to PULL.              */
```

```
PARSE VAR variable x y z .  /* If variable = '12 45 hello dear',*/
                            /* x=12, y=45, z='hello'. The final */
                            /* '.' is a 'placeholder' used to   */
                            /* force z to be equal to only      */
                            /* 'hello' and not the entire       */
                            /* remainder of the variable. A     */
                            /* placeholder may be used anywhere */
                            /* to 'eat up' an unwanted word.    */

PARSE VAR variable x ',' y  /* If variable = '12,45' then       */
                            /* x=12, y=45. The delimiter may be */
                            /* any valid character.             */

PARSE VALUE 'Hi'||x WITH z  /* If x='de', then z='Hide'.        */

PARSE VAR r 1 x 4 y +3 z    /* If r='1234567890', then x='123', */
                            /* y='456', and z='7890'. That is,  */
                            /* x equals the characters from     */
                            /* position 1 up to but not incl.   */
                            /* position 4, y equals the chars   */
                            /* from position 4 up to but not    */
                            /* including position 4 plus 3, and */
                            /* z equals the rest of the string  */
                            /* beginning at position 4 plus 3.  */
```

PULL

Synonymous to PARSE UPPER PULL. Pulls input from the console, first printing a prompt if a prompt string has been set with OPTIONS.

Example:

```
OPTIONS PROMPT 'Input a,b:' /* Set an input prompt string.      */
PULL a b .                  /* Pull values from console for a,b */
```

SAY

Sends output to the console window. Synonymous to ECHO.

Example:

```
SAY 'Howdy' 3+7     ==>  Howdy 10
```

9.6.7 FUNCTIONS & PROCEDURES

ARG

Shorthand for PARSE UPPER ARG. Gets arguments for a function.

Example:

```
sqr:
```

```
ARG a,b,c                    /* If function was called as          */
                             /*   x = sqr(1,2,'howdy')             */
                             /*  then a=1, b=2, c='HOWDY'          */
```

ARG( )

In a function, tells you how many arguments were supplied to the current function, what they were, or whether or not they exist, depending on options. ARG() may also be used to report the number of arguments passed on the command line when a program is run, but it will always report '1' argument with a value equal to a string containing all the arguments supplied.

Example:

```
say ARG()                    /* If the function was called as    */
                             /*  func: (12,23,34)                */
                             /* ARG() returns '3', the number of */
                             /* arguments in the function call.  */

say ARG(3)                   /* Following above example, returns */
                             /* '34', the third argument.        */

say ARG(2) EXISTS            /* Following above example, returns */
                             /* '1' or TRUE, because there IS a  */
                             /* second argument. Also may use    */
                             /* OMITTED, and abbr. to 1st letter */
```

CALL

Invokes an internal or external function, and passes arguments to it, if any. Also allows you to call a function without returning a value.

Example:

```
CALL print x y z             /* Executes a function called 'print'   */
                             /* and passes it the values in x,y,z    */

CALL OPEN 'infile','df0:ReadMe.txt','R' */ Does not return a value  */
```

PROCEDURE

This command is used within an internal function to protect the symbols used by making them local.

Example:

```
a=10;b=5                     /* Define symbols in main program.      */
fact: PROCEDURE EXPOSE b     /* Create a function called fact:.      */
  say a              ==> A   /* The symbol 'a' is undefined, because */
  say b              ==> 5   /* 'PROCEDURE' has made all symbols     */
                             /* local except 'b', which the EXPOSE   */
                             /* option has made global.              */
```

RETURN

This command ends a function call and returns the value indicated. Functions that have been called with the CALL command do not return a value.

Example:

```
fact:                       /* Define a function called fact:.      */
  ARG i                     /* Get value from calling operation.    */
  IF i <=1                  /* Test value.                          */
    THEN RETURN 1           /* Return a value of 1 if true.         */
    ELSE RETURN i*fact(i-1) /* Calculate and return value if false. */
```

9.6.8 SYSTEM

ADDLIB()

Adds a function library or function host to the Library List. ADDLIB() is most often used to invoke ARexx's rexxsupport.library. After an ADDLIB() command, the functions contained in the library may be used by ARexx in the same manner as the ARexx resident functions.

Example:

```
q=ADDLIB('rexxsupport.library',0,-30,0)
                            /* The numbers are accurate for         */
                            /* 'rexxsupport.library, but may be     */
                            /* different for other libraries.       */
```

DATE()

Returns the date in the specified format. Valid options are BASEDATE (days since 1/1/10, CENTURY (days since 1/1/1900), DAYS (days since 1/1 current year), EUROPEAN (DD/MM/YY), INTERNAL (days since 1/1/78 ("Amiga Time")), JULIAN (YYDDD), MONTH (current month), NORMAL (DD MMM YYYY), ORDERED (YY/MM/DD), SORTED (YYYYMMDD), USA (MM/DD/YY), and WEEKDAY (current day of the week). These options can be shortened to just the first character. The DATE() function also accepts optional second and third arguments to supply the date either in the form of internal system days or in the 'sorted' form YYYYMMDD. The second argument is an integer specifying either system days (the default) or a sorted date. The third argument specifies the form of the date and can be either 'I' or 'S'.

Example:

```
SAY DATE() ==> 20 Jul 1991
SAY DATE('M') ==> April
```

```
SAY DATE('s',date('i')+21) ==> 19910609
SAY DATE('w',19910609,'S') ==> Sunday
```

DIGITS()

Returns the current numeric digits setting.

Example:

```
NUMERIC DIGITS 6; SAY DIGITS() ==> 6
```

ERRORTEXT()

Returns the error message associated with the error number.

Example:

```
SAY ERRORTEXT(15)  ==> 'Function not found'
```

FORM()

Returns the current numeric form setting.

Example:

```
NUMERIC FORM SCIENTIFIC;SAY FORM() ==> SCIENTIFIC
```

FUZZ()

Returns the current numeric fuzz setting.

Example:

```
NUMERIC FUZZ 3;SAY FUZZ() ==> 3
```

GETCLIP()

Returns the value associated with the supplied name from the ARexx Clip List, which is used as a common repository by all running ARexx applications. The name used is case-sensitive. See SETCLIP().

Example:

```
q=SETCLIP(Hello,123) ==> 1  /* Store a value in the Clip List.  */
z=GETCLIP(Hello) ==> '123'  /* Retrieve the value.              */
```

HASH()

Returns the hash attribute of a string as a decimal number, and updates the internal hash value of the string. Useful for checking data integrity.

Example:

```
SAY HASH('ijk')   ==>  '62'
```

INTERPRET

Interprets a string or symbol as an ARexx command block. This command allows you to input or build command strings to be executed.

Example:

```
greetings = 'say "Howdy"'
INTERPRET greetings  ==> 'Howdy'
```

NUMERIC

This command allows you to set certain attributes that affect how ARexx does math and comparisons.

Example:

```
NUMERIC DIGITS 5            /* Sets number of digits of precision   */

NUMERIC FUZZ 3              /* Specifies number of digits to be     */
                            /* ignored when making comparisons.     */

NUMERIC FORM SCIENTIFIC     /* Sets output to scientific notation.  */

NUMERIC FORM ENGINEERING    /* Sets output to engineering notation. */
```

OPTIONS

A command to set various system attributes. Keyword NO will reset the requested attribute to its default setting.

Example:

```
OPTIONS FAILAT 10           /* Sets minimum error signal at 10      */

OPTIONS PROMPT 'Input:'     /* Sets input prompt for PULL as        */
                            /* 'Input:'                             */

OPTIONS RESULTS NO          /* Tells ARexx not to request results   */
                            /* when issuing commands to ext. host   */
```

PRAGMA( )

Can be used to get or change the current directory, and modify or poll certain other system parameters.

Example:

```
CALL PRAGMA 'directory','df0:Text'  /* Sets new current directory   */

z=PRAGMA('D')   /* Gets current directory   */
```

REMLIB()

Removes the named library from the Library List. Returns a boolean value indicating whether the operation was successful. See ADDLIB().

Example:

```
SAY REMLIB('rexxsupport.library')
```

SETCLIP()

Adds a name-value pair to the ARexx Clip List. Returns a boolean value. See GETCLIP().

Example:

```
q=SETCLIP(Hello,123) ==> 1  /* Store a value in the Clip List.  */
z=GETCLIP(Hello) ==> '123'  /* Retrieve the value.              */
```

SHOW()

Returns the names in the resource list specified, or searches for a named entry, in which case it returns a boolean value. Lists supported are CLIP, FILES, INTERNAL, LIBRARIES, and PORTS. The name entries are case-sensitive.

Example:

```
SAY SHOW('libraries')   /* Lists all available libraries       */

SAY SHOW('C',Hello)   /* Checks for clip named 'Hello'         */
```

SOURCELINE()

Returns the text for the specified line of the ARexx program. If the line argument is omitted, returns the total number of lines in the file. Often used to embed "help" information in a program.

Example:

```
/* A sample program */
SAY SOURCELINE()  ==> 3
SAY SOURCELINE(1) ==>  /* A sample program */
```

SYMBOL()

Tests whether the supplied name is a valid ARexx symbol. Returns the string BAD if it's not, LIT if it is valid but unused, and VAR if it is valid and assigned.

Example:

```
zed='Hi'; SAY SYMBOL('zed')  ==> VAR */ zed is valid and assigned.  */
```

TIME()

Resets or reads the system time clock. Valid options may be abbreviated to their first letter. Options are ELAPSED program time in seconds, HOURS since midnight, MINUTES since midnight, SECONDS since midnight, or RESET the elapsed time clock. If called without an option, TIME() returns the system time in the form HH:MM:SS.

Example:

```
SAY TIME()       ==> HH:MM:SS

z=TIME('R')      /* Resets timer   */
```

TRACE

ARexx supports a very powerful TRACE debugging option. Its many features are beyond the scope of this chapter. Refer to the Amiga or ARexx manual for complete information on using TRACE. Valid TRACE options are ALL, BACKGROUND, COMMANDS, ERRORS, INTERMEDIATES, RESULTS, LABELS, RESULTS, and SCAN. A preceding "?" toggles interactive mode, and a "!" toggles command inhibition. There is also a TRACE() function, which can be used to set or read the current tracing option.

9.6.9 DATA

B2C

Converts a string of binary digits to its character equivalent.

Example:

```
SAY B2C('01100001')   ==>
```

C2B()

Translates a string into a string that is its binary equivalent.

Example:

```
x=C2B('abc') ==> '011000010110001001100011'
```

C2D()

Translates a string into a number expressed in ASCII digits 0-9.

Example:

```
x=C2D('CDEF'x)  ==> 52719
```

C2X()

Translates a string into a string that is its hexadecimal equivalent.

Example:

```
x=C2X('abc')   ==> '616263'
```

D2C()

Translates a decimal number into its packed binary representation. Opposite of C2D().

Example:

```
x=D2C(31)   ==> '1F'x
```

D2X()

Translates a decimal number into its hexadecimal representation.

Example:

```
D2X(32)  ==> '20'x
```

DATATYPE()

Tells you which data type a string conforms to, either NUM if a valid number, or CHAR. If a data type is specified, returns a boolean value (1 or 0) indicating whether the string is of that data type. Valid data types are ALPHANUMERIC, BINARY, LOWERCASE, MIXED (upper and lowercase), NUMERIC, SYMBOL (valid ARexx symbol name), UPPERCASE, WHOLE, and X (hex digits string). Each may be shortened to its first letter.

Example:

```
say DATATYPE('abc1')  ==> 'CHAR'

x=DATATYPE('AbCDE','UPPERCASE')  ==> 0 (FALSE)
```

DROP

Drops the value of a symbol, resetting it to its uninitialized state, which is the same as its name. DROPping the value of a stem resets all compound variables created from the stem.

Example:

```
a =1;b=2;c=3
SAY  a b c      ==> 1 2 3
DROP a b c
SAY  a b c      ==> A B C
```

VALUE()

Returns the contents of the supplied symbol.

Example:

```
v=12;SAY VALUE(v)  ==> 12
```

X2C()

Converts a string of hex digits into its packed character representation.

Example:

```
z=X2C('10FD')  ==> '10FD'x
```

X2D()

Converts a hexadecimal number to decimal.

Example:

```
z=X2D('1F') ==> 31
```

9.6.10 BITS

BITAND()

Logically ANDs the two argument strings together.

Example:

```
BITAND('0313'x,'FFF0'x)   ==> '0310'x
```

BITCHG()

Toggles the state of the specified bit in a string. Bits are numbered from right to left, with the rightmost bit referred to as bit '0'.

Example:

```
BITCHG('0313'x,4)  ==> '0303'x
```

BITCLR()

Similar to BITCHG(), but clears the specified bit to 0.

Example:

```
BITCLR('0313'x,4) ==> '0303'x
```

BITCOMP()

Compares two strings bit by bit, and returns the position of the first bit in which they differ. Returns -1 if the strings are identical. Bits are numbered from the rightmost bit to the left, starting at 0.

Example:

```
BITCOMP('1010101'b,'1011101'b) ==> 3
```

BITOR()

Logically ORs the two argument strings together.

Example:

```
BITOR('0313'x,'FFF0'x)   ==> 'FFF3'x
```

BITSET()

Similar to BITCHG(), but sets the specified bit to 1.

Example:

```
BITSET('0313'x,5) ==> '0333'x
```

BITTST()

Returns the boolean state of the specified bit in the argument string.

Example:

```
BITTST('0313'x,4) ==> 1
```

BITXOR( )

Logically XORs the two argument strings together.

Example:

```
BITAND('0313'x,'FFF0'x)    ==> 'FCE3'x
```

9.6.11 MEMORY

EXPORT( )

Copies data from a string into a previously allocated memory buffer area, which must be specified as a 4-byte address. A maximum length may be specified; the default is the entire length of the string, and if the length specified is greater than the length of the string supplied, it will be padded with blanks or a specified pad character. This function returns the number of characters actually copied to memory. It is imperative that the actual address used be obtained with the ARexx GETSPACE() function; otherwise memory may be ruined. See also: FREESPACE().

Example:

```
count=EXPORT('0004 0000'x,'Valuable data')  /* The value of count   */
                                            /* is 13.               */

count=EXPORT('0004 0000'x,'Data',8,'+') /* The value of count is 8, */
                                        /* the maximum specified.   */
                                        /* The data actually in the */
                                        /* buffer is: 'Data++++'.   */
```

FREESPACE( )

Returns a block of memory of the indicated size back to the available ARexx pool. This command is used to free memory used by the EXPORT() command, but no longer needed. This function returns a boolean value indicating whether or not the process was successful. See also: GETSPACE().

Example:

```
SAY FREESPACE('0004 0000'x,32)  ==> 1
```

GETSPACE( )

Allocates a block of memory of the specified length from ARexx's internal pool. Returns the four-byte address of the block returned. GETSPACE() is the only safe way to allocate memory for the EXPORT() command. See also: FREESPACE().

Example:

```
say C2X(GETSPACE(32))    ==> '00242D18'
```

IMPORT()

Pulls data from memory and inserts it in a string. If a length is not specified, IMPORT() quits when a null byte is found. See also: EXPORT().

Example:

```
z=IMPORT('0004 0000'x,32)
```

STORAGE()

Returns amount of free system memory, or can be used with a syntax identical to EXPORT() to store strings to memory. Use with caution.

Example:

```
z=STORAGE()          /* z = amount of free system RAM                */

z=STORAGE('0004 0000'x,'Data',8,'+')    /* The value of z is 8,      */
                                        /* the maximum specified.    */
                                        /* The data actually in the  */
                                        /* buffer is: 'Data++++'.    */
```

9.7 Example ARexx Programs

The following ARexx programs are included on the companion diskette included with this book. The programs show you a small fraction of the capabilities of the ARexx language.

Test

This program determines whether a file exists in a directory on diskette.

```
/*  Program name: Test                                                  */
/*                                                                      */
/*  This ARexx program determines if a file exists in the directory     */
/*  "dh0:text".                                                         */
/*                                                                      */
/*  To run this program, type:                                          */
/*  rx test filename                                                    */
/*                                                                      */

PARSE ARG x                          /* Get filename from input stream. */
x = 'dh0:text/' || x                 /* Add pathname to filename.       */
IF EXISTS(x) THEN SAY x 'exists'     /* Tell the user the file exists.  */
  ELSE SAY 'No such file as' x       /* Tell the user there is no file. */
```

Substitute

This program substitutes the number 1 to 9 for the letters 'a' to 'i' in a string, printing the reverse of the resulting string at each iteration of the loop.

```
/*  Program name: Substitute                                            */
/*                                                                      */
/*  This example frivolously substitutes the numbers 1 to 9             */
/*  for the letters 'a' to 'i' in a string, printing the reverse        */
/*  of the resulting string at each iteration of the loop.              */
/*                                                                      */
/*  To run this program, type:                                          */
/*  rx substitute                                                       */
/*                                                                      */

a = XRANGE('a','i')                  /* Make a string of nine letters.  */
DO z = 1 TO 9                        /* Set up a loop.                  */
  a = OVERLAY(z,a,z)                 /* Substitute number for letter.   */
  ECHO a 'backwards is:' REVERSE(a)  /* Print the string and reverse.   */
  END                                /* End the loop.                   */
```

Printfile

This program inputs lines from a file and outputs them to the CLI preceded by a line number and caret, such as '66>'. It's very useful for listing ARexx programs to find line numbers.

```
/*  Program name: Printfile                                          */
/*                                                                   */
/*  This program inputs lines from a file and outputs them to the CLI  */
/*  preceded by a line number and caret, like this: '66>'            */
/*  It's very handy for listing ARexx programs to find line numbers.   */
/*                                                                   */
/*  To run this program, type:                                       */
/*  rx printfile dxn:pathname/filename                               */
/*                                                                   */

ARG filename                          /* Get input filename from CLI. */
IF ~EXISTS(filename) THEN DO          /* Check to see if file exists. */
  SAY filename 'does not exist'       /* Inform user if no file exists*/
  EXIT                                /*   and end program.           */
  END                                 /* End DO loop.                 */
CALL OPEN("readme",filename,"R")      /* Open file to read on disk.   */
n = 0                                 /* Zero line counter.           */
DO UNTIL EOF("readme")                /* Do until end of file.        */
  n = n+1                             /* Increment line counter.      */
  line = READLN("readme")             /* Get a line from disk.        */
  SAY n '>' line                      /* Output line to CLI window.   */
  END                                 /* End do loop.                 */
CALL CLOSE("readme")                  /* Optional 'close' statement.  */
```

RollDice

This program simulates the roll of two dice and displays the results in the CLI window.

```
/* Program name: RollDice                                            */
/*                                                                   */
/* This program simulates the roll of two dice, and displays the     */
/* results in the CLI window.                                        */
/*                                                                   */
/* To run this program, type:                                        */
/* rx rolldice                                                       */
/*                                                                   */

d.1.1 = '   '                         /* Define dice shapes           */
d.2.1 = ' * '                         /*   using 'stem' variables.    */
d.3.1 = '   '                         /*                              */
d.1.2 = '  *'                         /* There are actually only      */
d.2.2 = '   '                         /*   five patterns here:        */
d.3.2 = '*  '                         /* '   ''* *''  *'' * ''*  '    */
```

```
d.2.3 = ' * '                          /* Can you come up with a       */
d.3.3 = '*  '                          /*   more economical way of     */
d.1.4 = '* *'                          /*   defining the dice shapes?  */
d.2.4 = '   '
d.3.4 = '* *'
d.1.5 = '* *'
d.2.5 = ' * '
d.3.5 = '* *'
d.1.6 = '* *'
d.2.6 = '* *'
d.3.6 = '* *'

die1 = roll(6)                         /* Call roll function.          */
die2 = roll(6)                         /* Call roll function again.    */
total = die1 + die2                    /* Calculate total.             */
SAY 'Total rolled is:' total           /* Print total to CLI.          */
EXIT                                   /* All done.                    */

roll:                                  /* Define the 'roll' function.  */
  ARG n                                /* Get high value for die.      */
  die = RANDOM(1,n,TIME('S'))          /* Generate number from 1 to n. */
  SAY '    ---'
  SAY '   |' || d.1.die || '|'         /* Print die face.              */
  SAY '   |' || d.2.die || '|'
  SAY '   |' || d.3.die || '|'
  SAY '    ---'
  SAY ''
  RETURN die                           /* Return the value rolled.     */
```

LoadLibrary

This example shows a proper way to open the Support Library included with ARexx. The rexxsupport.library should be in your LIBS: directory before running this program. If it isn't, the program will report that the library is not available.

```
/* Program name: LoadLibrary                                          */
/*                                                                    */
/* This example shows a proper way to open the Support Library        */
/* included with ARexx. The rexxsupport.library should be in your LIBS: */
/* directory before running this program. If it isn't, the program    */
/* will report that the library is not available.                     */
/*                                                                    */
/* To run this program, type:                                         */
/* rx LoadLibrary                                                     */
/*                                                                    */

IF ~SHOW('L',"rexxsupport.library") THEN DO     /* Is the lib mounted?  */
  IF ADDLIB('rexxsupport.library',0,-30,0) THEN /* Try to add it if not,*/
    SAY 'added rexxsupport.library'             /* and say if successful*/
  ELSE DO                                       /* Otherwise,           */
    SAY 'Support Library not available.'        /* say it's not there   */
    EXIT                                        /* and quit the program.*/
    END                                         /* End 'ELSE DO' loop.  */
  END                                           /* End 'IF DO' loop.    */

SAY 'Pausing for five seconds...'   /* Put up a warning...            */
```

```
CALL DELAY 250                          /* DELAY() is a function contained  */
                                        /*    in the rexxsupport.library.   */
CALL REMLIB('rexxsupport.library')      /* Remove the library, free memory. */
```

FindWord

This program takes a word as an argument and sees how many words can be made from recombinations of its letters. It tests the words formed against a master list contained in a file in the current directory called "Words.dat". This file is not contained on the companion diskette, you must create the Words.dat file. You can create such a list using a word processor or use a pre-existing word list from a spelling checker. (Words should be separated by a carriage return.) This program outputs to the CLI and to a file in the current directory named the same as your input word. This program does not test to make sure files can be opened. It will simply quit unceremoniously with an error message if an error occurs.

```
/* Program name: FindWords                                                  */
/*                                                                          */
/* This program takes a word as an argument and sees how many words         */
/* can be made from recombinations of its letters. It tests the             */
/* words formed against a master list contained in a file in the            */
/* current directory called "Words.dat". You can create such a list         */
/* using a word processor or use a pre-existing word list from a            */
/* spelling checker. (Words should be separated by a carriage return.)      */
/* This program outputs to the CLI and to a file in the current             */
/* directory named the same as your input word.                             */
/* This program does not test to make sure files can be opened. It will     */
/* simply quit unceremoniously with an error message if an error occurs.*/
/*                                                                          */
/* To run this program, type:                                               */
/* rx FindWords inputword                                                   */
/*                                                                          */

ARG instring                            /* Get input word from CLI.         */
q = OPEN("wordlist","Words.dat","R")    /* Open the word list on disk.      */
q = OPEN("outlist",instring,"W")        /* Open output file.                */

DO UNTIL EOF("wordlist")                /* Test until end of file.          */
  teststring = instring                 /* Copy the input word.             */
  testword = READLN("wordlist")         /* Read a word from the file.       */
  testlength = LENGTH(testword)         /* Find the word's length.          */
  DO i = 1 TO testlength                /* Test each character.             */
    letter = SUBSTR(testword,i,1)       /* Get a letter from the word.      */
    p = POS(letter, teststring)         /* Is it in the test word?          */
    IF p = 0 THEN BREAK                 /* If not, leave the loop.          */
    teststring = OVERLAY("-",teststring,p) /* If so, mark the place.        */
    IF i = testlength THEN DO           /* If all letters match, then       */
      ECHO testword                     /*   print to CLI                   */
      CALL WRITELN("outlist",testword)  /*     and to file.                 */
      END                               /* End inner DO loop.               */
  END                                   /* End middle DO loop.              */
END                                     /* End outer DO loop.               */
```

RandNum

Creates a given number of unique random numbers between 1 and a limit, and outputs those numbers to a custom CLI window.

```
/* Program name: RandNums                                              */
/*                                                                     */
/* Creates a given number of unique random numbers between 1 and a     */
/* limit, and outputs those numbers to a custom CLI window.            */
/*                                                                     */
/* To run this program, type:                                          */
/* rx RandNums                                                         */
/*                                                                     */

OPTIONS PROMPT 'Input Maximum number? 10-999: '  /* Prompt for input.  */
PULL high                        /* Get the highest number to generate. */
OPTIONS PROMPT 'Input How Many numbers? 5-498: ' /* Prompt for input.  */
PULL howmany                     /* Get the number of numbers to make.  */

IF high <10 THEN high = 10       /* Make the high number at least 10,   */
IF high >999 THEN high = 999     /*   but no more than 999.             */
IF (howmany<5) | (howmany>TRUNC(high/2)) THEN howmany=TRUNC(high/2)
                                 /* Set a reasonable value for howmany. */
SAY 'Generating' howmany 'unique random numbers from 1 to' high

OPEN('Random','CON:160/50/345/100/Random')  /* Open custom CLI.        */

q. = 0                           /* Set all the test values to zero.    */
CALL TIME 'R'                    /* Reset system timer.                 */
DO howmany                       /* Do until howmany numbers are made.  */
  DO UNTIL q.z ~= z              /* Test for uniqueness.                */
    z=RANDOM(1,high,TIME('s'))   /* Generate a random number in range.  */
    END                          /* End inner DO loop.                  */
  q.z = z                        /* Set the value to say it's used.     */
  CALL WRITECH('Random',RIGHT(z,3,'0') '') /* Output to the window.     */
END                              /* End the outer DO loop.              */
t = TIME('E')                    /* Get elapsed time.                   */
SAY 'Elapsed Time:' t 'Seconds'  /* Report elapsed time.                */
CALL CLOSE('Random')             /* Close the output window.            */
```

Guessname

This program generates a number between 1 and a high limit chosen by the user. It then prompts for guesses until you guess the number or give up.

```
/* Program name: GuessNum                                              */
/* This program generates a number between 1 and a high limit chosen   */
/* by the user. It then prompts for guesses until you guess the number */
/* or give up.                                                         */
/*                                                                     */
/* To run this program, type:                                          */
/* rx GuessNum                                                         */
/*                                                                     */
DO i = 2 to 5                                 /* DO loop for instruct. */
  SAY SOURCELINE(i)                           /* SAY lines 2-5 above.  */
  END                                         /* End DO loop.          */

OPTIONS PROMPT 'High Number (10-999)? :'      /* Set prompt for input. */
PULL highnum                                  /* Take input.           */
IF (highnum>999) | (highnum<10) THEN DO       /* Is highnum in range?  */
  SAY "You didn't follow directions!"         /* If not, tell the user,*/
  SAY "I'll pick a High Number for you!"      /*   and do it for him.  */
  highnum = RANDOM(10,999,TIME('S'))          /* Pick a valid highnum. */
  END                                         /* End DO loop.          */
SAY 'Generating a random number between 1 and' highnum  /* Feedback.   */
randnum = RANDOM(1,highnum,TIME('S'))         /* Generate random number.*/
count = 0                                     /* Zero counter.         */

OPTIONS PROMPT 'Guess a number (0 to quit) :' /* Prompt for guess.     */
DO forever                                    /* Infinite loop.        */
  DO until (guess<=highnum) & (guess>-1)      /* Is guess in range?    */
    PULL guess                                /* Get guess from user.  */
    END                                       /* End input loop.       */
  count = count + 1                           /* Increment counter.    */
  SELECT                                      /* Make some comparisons.*/
    WHEN guess = 0 THEN DO                    /* Did user quit?        */
      SAY 'Giving up after only' count 'guesses?' /* Chastise user.    */
      SAY 'The number was:' randnum           /* Give the answer.      */
      EXIT                                    /* Quit the program.     */
      END                                     /* End inner DO loop.    */
    WHEN guess = randnum THEN DO              /* If user got it right, */
      SAY 'You guessed the number in' count 'guesses!' /* tell him so, */
      EXIT                                    /* and quit the program. */
      END                                     /* End inner DO loop.    */
    WHEN guess > randnum THEN SAY 'Lower'     /* If high, say 'Lower'. */
    OTHERWISE SAY 'Higher'                    /* Otherwise, say 'Higher'*/
    END                                       /* End SELECT structure. */
END                                           /* End 'infinite' DO loop.*/
```

10.
Quick Reference

10. Quick Reference

This chapter contains all the information you have seen so far about AmigaDOS commands, as well as commands for controlling ED and Edit. The three sections of this chapter show these commands in abbreviated form to help you find them easily.

Sections 10.1 and 10.2 present an overview of the key combinations and their effects in the ED and Edit editors. These are listed in table format.

Section 10.3 lists the AmigaDOS Shell commands in a similar format. Hopefully this format will help you find commands much faster.

10.1 The ED Program

10.1.1 ED 1.14

The ED editor uses two types of commands. One type executes immediately after the corresponding key combination is pressed; the second type requires entry of the first command in command mode.

First we will present the direct commands that always consist of two key combinations. These key combinations always begin with the <Ctrl> (control) key. You press another key to implement the command.

The command mode commands will be presented next. You enter command mode by pressing the <Esc> (Escape) key. You can tell if you are in the command mode by a small star (asterisk) in the lower left corner of the editor screen. You only need to enter the command and press the <Return> key to start the command.

There are commands in both sections that have the same effect, so you must decide which type of command works better for you.

Direct commands (without <Ctrl>)

Tab	Moves the cursor to the next tab mark.
Del	Erases the character under the cursor.
Backspace	Erases the character to the left of the cursor.
Return	Text between the cursor position and the end of the line are moved to the next line.

Direct commands (with <Ctrl>)

<Ctrl><I>	Moves the cursor to the next tab mark.
<Ctrl><U>	Moves the cursor up 12 lines.
<Ctrl><D>	Moves the cursor down 12 lines.
<Ctrl><R>	Moves the cursor to the end of the word to its left.
<Ctrl><T>	Moves the cursor to the start of the next line.
<Ctrl><J>	Moves the cursor to the start or end of the line.
<Ctrl><E>	Moves the cursor to the start or end of the window.
<Ctrl><H>	Erases the character to the left of the cursor.
<Ctrl><O>	Erases a word or all spaces up until the next word.
<Ctrl><Y>	Erases everything from the cursor position to end of line.
<Ctrl><B>	Erases the entire line.
<Ctrl><M>	Text between the cursor and end of line is moved to next line.

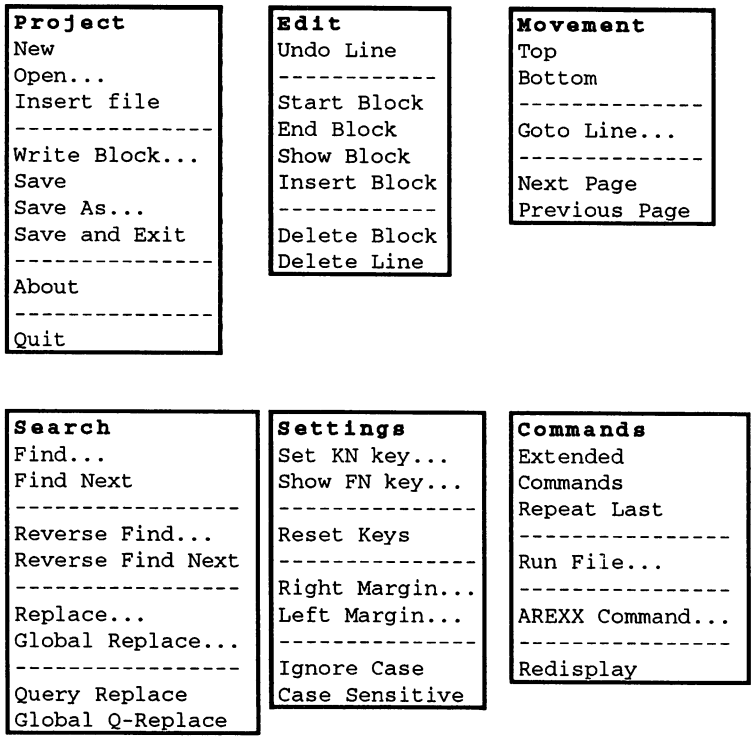
<Ctrl><A>	Inserts a line.
<Ctrl><G>	The last command mode command is repeated.
<Ctrl><[>	Enter command mode same as pressing <Esc>.
<Ctrl><F>	Toggles between upper/lowercase.

Command mode commands (with <Esc>)

M n	Moves the cursor to the nth line.
CL	Moves the cursor one position to the left.
CR	Moves the cursor one position to the right.
CS	Moves the cursor to the start of the line.
CE	Moves the cursor to the end of the line.
P	Moves the cursor to the start of the previous line.
N	Moves the cursor to the start of the next line.
T	Moves the cursor to the start of the text.
B	Moves the cursor to the end of the text.
BS	Marks the cursor position as the start of a block.
BE	Marks the cursor position as the end of a block.
SB	Shows the marked block in a window.
WB "data"	Saves marked block to "data".
IB	Inserts marked block at the cursor position.
DB	Deletes marked block.
DC	Deletes character at cursor.
D	Deletes entire line.
S	Moves text between cursor position and end of line to next line.
J	Combines current line with next line.
I "Text"	Inserts "Text" before current line.
A "Text"	Inserts"Text" after current line.
IF Data	Inserts Data at cursor position.
E "Text1"Text2"	Exchanges "Text1" for "Text2".
EQ "Text1"Text2"	Exchanges "Text1" for "Text2" after prompt.
F "Text"	Begins search for "Text" at cursor position.
BF "Text"	Searches for "Text" up to the cursor position.
LC	Enables case sensitivity during a text search .
UC	Disables case sensitivity during a text search.
X	Exits ED and saves text.
Q	Exits ED without saving text.
SA	Saves text.
RP	Repeats command until an error occurs.
SH	Displays current editor settings.
U	Changes to the current line are canceled.
SL n	Sets left margin to n.
SR n	Sets right margin to n.
EX	Ignores right margin on current cursor line.

10.1.2 ED 2.00

ED has been updated by John Toebes III, and the improved version is excellent. The new 2.0 operating system allows programmers to add customizable menus to their programs. The customizable menus for the latest version of ED are contained in the `ED-Startup` script file in the `S:` directory. They allow you to use the mouse when using ED to access often used commands. The normal ED commands are still available. The new version includes menus and the new standard Amiga file selector. All commands may still be accessed using the keyboard commands. The following are the new standard ED 2.0 menus (no `ED-Startup` file):



10.2 The Edit Program

Besides the ED editor, described in the previous section, there is another editor on the Workbench disk. It's the `Edit` editor. The following list of `Edit` commands is intended as a reference only, not as detailed instructions. More detailed information about the operation of the editor can be found in Section 2.4.2.

Partial arguments are passed together with the command words. The slash / serves as a separator between strings. Arguments that have alternate input possibilities are placed in parentheses (). So that the command text doesn't become too long we use the following abbreviations:

```
a,b   = line number (or . or *)
bg    = command group
m,n   = numbers
q     = qualifier
sk    = search criteria
sw    = change value (+ or -)
z1,z2 = string
```

and now to the commands:

<	Moves the character pointer one character to the left.
>	Moves the character pointer one character to the right.
#	Deletes the character at the character pointer position.
$	Changes the character at the character pointer position.
%	Changes the character at the character pointer position to uppercase.
PA(q)/zl/	Moves the character pointer to the position after the specified string zl.
PB(q)/zl/	Moves the character pointer to the position before the specified string zl.
PR	Moves the character pointer to the start of the line.
M n	Moves the character pointer to line n.
M +	Moves the character pointer to the last line of source data.
M -	Moves the character pointer to the first available line in the buffer.
N	Moves the character pointer to the next line.
P	Moves the character pointer to the previous line.
Rewind	"Rewinds" the source file.
F((q)/sk/)	Searches line that contains the string sk (forward search).
BF((q)/sk/)	Searches line that contains the string sk (reverse search).

DF((q)/sk/)	Searches line containing the string sk (forward search), deletes all skipped lines.
?	Verifies the current line.
!	Verifies the current line with all non-printable characters.
T	Displays the source file up to the end of the file.
T n	Displays the next n lines of the source file.
TL n	Displays the next n lines of the source file with line numbers.
TN	Displays as many lines as will fit in the text buffer.
TP	Displays all lines of the text buffer and sets the pointer to the beginning of the text buffer.
V sw	Enables (+) or disables (-) verification.
A (q)/z1/z2/	Inserts string z2 after string z1.
AP(q)/z1/z2/	Inserts string z2 after string z1, then sets the character pointer behind z1.
B(q)/z1/z2/	Inserts string z2 before string z1.
BP(q)z1/z2/	Inserts string z2 before string z1, then sets the character pointer behind z1.
CL(/z1/)	Combines the current line, zl and the next line.
D	Deletes the current line.
DFA(q)/z1/	Deletes the current line after string z1.
DFB(q)/zl/	Deletes the current line from the beginning of string z1.
DTA(q)/z1/	Deletes the current line from the beginning of the line to the end of string z1.
DTB(q)/z1/	Deletes the current line up to, but not including, string z1.
E(q)/z1/z2/	Replaces string z1 with string z2.
EP	Replaces string z1 with string z2, then positions the character pointer following z2.
I(a)	Inserts text in front of the current line or before a line from the keyboard until z is entered.
I z1	Inserts entire contents of the file z1 before the current line.
R(a(b))	Deletes lines a through b and allows text entry from the keyboard.
(a(b))z1	Deletes lines a through b and inserts the text contained in file z1.
SA(q)/z1/	Separates the current line after string z1 when it occurs.
SB(q)/z1/	Separates the current line before string z1 when it occurs.
GA(q)/z1/z2/	Inserts string z2 after string z1 in the current line.
GB(q)/z1/z2/	Inserts string z2 before string z1 in the current line.
GE(q)/z1/z2/	Replaces string z1 with string z2 in all current lines.
CG(n)	Disables global operation n or all global operations.
DG(n)	Temporarily disables global operation n or all global operations.
EG(n)	Enables global operation n or all global operations.
SHG	Displays information about all global operations that have been active until now.
From	Makes original From file the current source file.

From .z1	Makes file z1 the current source file.
To	Makes original To file the current destination file.
To .z1	Makes file z1 the current destination file.
CF .z1.	Closes file z1.
=n	Assigns the current line number n.
C .z1.	Reads additional `Edit` commands from file z1.
H n	Sets the next breakpoint to line n. (n=* erases all breakpoints).
Q	Exits command mode; if you exit the highest command level, the rest of the source file is transferred.
SHD	Displays saved command information.
Stop	Exits `Edit`.
TR sw	Enables (+) or disables (-) sensitivity to leading spaces.
W	Carries the rest of the source file to the destination file.
Z z1	Changes the end character for the insert command z1.

10.3 The AmigaDOS Commands

This section briefly covers the AmigaDOS commands. First the correct 1.3 syntax of the command appears, then a short description of the command, followed by a description of the arguments. If the command supports additional arguments in Version 2.0 they are described. The AmigaDOS commands added to Version 1.3 are marked with the identifier (**AmigaDOS 1.3**). Version 2.0 improvements are also marked. All AmigaDOS 2.0 commands have been rewritten in C, which has greatly reduced their size and enhanced their execution speed. Many of the commands were made internal AmigaDOS commands in Version 2.0.

The following qualifiers are used in the command descriptions:

/A (Argument) This qualifier always requires a certain argument. If you omit the argument, the command cannot execute.

/K (Key) The qualifier's name must appear as input and a keyword must appear as well. The parameters allowed and the functions executed depend on the respective `Shell` command (see Chapter 2 for more details).

/S (Switch) This qualifier needs no arguments. It acts as a switch (toggle) for a command. Switches in commands do just what a wall switch does—they turn a command on or off, or switch the command to another mode.

The following are possible qualifiers that can appear in an argument template only in AmigaDOS 2.0:

/N (Numeric) This qualifier indicates that a numeric argument is expected (DOS 2.0 only).

/M (Multiple) Multiple arguments can be included. In 1.3 commas were used to signify multiple arguments. Multiple arguments must be separated by spaces. This has been updated in DOS 2.0, also the number of arguments is unlimited in DOS 2.0 (DOS 2.0 only).

/F (Final) The argument is the final argument. This allows using strings without enclosing them in quotation marks (DOS 2.0 only).

, (comma) The command takes no arguments (DOS 2.0 only).

ADDBUFFERS DRIVE/A, BUFFERS/S

Reserves a buffer on a drive with a certain amount of memory.

DRIVE	The drive assigned the buffer.
BUFFERS	The size of the buffer to be allocated.

ALIAS NAME STRING/F (Shell only)

This command can only be used in conjunction with the `Shell`. The command assigns a string to a word (See Chapter 6).

NAME	The new command word.
STRING	Contains the command that is called with `NAME`.

V1.3 Command available in AmigaDOS 1.3 Shell.

V2.0 Command made an AmigaDOS internal command and correct argument template added.

Ask PROMPT/A

Asks a question answered with only (Y)es or (N)o: `y` returns an error code of 5 and `n` returns no error code.

PROMPT	Contains text displayed on the screen, usually in the form of a question.

V2.0 Command made an AmigaDOS internal command.

ASSIGN NAME,DIR,LIST/S,EXISTS/S,REMOVE/S

Assigns a logical device to a directory.

NAME	The logical device.
DIR	The directory assigned the logical device.
LIST	Lists the assignments of the logical devices.
EXISTS	Searches for `NAME` in the `ASSIGN` list. The error code 5 is returned if `NAME` is not present.
REMOVE	Removes `Name` from the `ASSIGN` list, used for development only.

V2.0 **TARGET/M,DISMOUNT/S,DEFER/S,PATH/S,ADD/S, VOL/S,DIRS/S,DEVICES/S**

TARGET	The TARGET/M argument allows you to make multiple assignments to a single device.
DISMOUNT	The DISMOUNT/S argument allows devices and directories to be removed from the assignment list.
DEFER	The DEFER/S argument creates a late-binding assignment. This assignment only takes effect when the assigned object is accessed.
PATH	The PATH/S argument creates a non-binding assignment. It does not take effect until it is referenced and only remains in effect while it is needed.
ADD	Adds assignment.
VOL	The VOL/S argument will only display information on the current volume assignments.
DIRS	The DIRS/S argument will only display information on the current directory assignments.
DEVICES	The DEVICES/S argument will only display information on the current device assignments.

AVAIL CHIP,FAST,TOTAL (AmigaDOS 1.3)

Displays an overview of the present available memory configuration.

CHIP	Optional, displays total chip memory.
FAST	Optional, displays total fast memory.
TOTAL	Optional, displays total available memory.

V2.0 **FLUSH/S**

FLUSH	Flushes memory areas.

BINDDRIVERS

Binds additional device drivers to the system.

BREAK PROCESS/A,ALL/S,C/S,D/S,E/S,F/S:

Stops a task in process.

PROCESS	Process to be broken off.
All	Sets the break level at C, D, E and F.
C,D,E,F	Sets break level.

V2.0 **PROCESS**

PROCESS/A/N	Specified as numeric.

CD DIR:

Changes the directory or displays the current directory.

DIR: The drive or the directory which should be accessed.

V2.0 Command made an AmigaDOS internal command.

CHANGETASKPRI PRI/A,PROCESS/K

Changes the priority of a process started from the CLI.

PRI Priority, shown by Status command. Contains the new priority (-128 to 127).
PROCESS The new priority is assigned to PROCESS number. See the Status command.

V2.0 PRI=PRIORITY/A/N,PROCESS/K/N

PRIORITY Specified as numeric and same as PRI.
PROCESS Specified as numeric.

COPY FROM,TO/A,ALL/S,QUIET/S, BUF=BUFFER/K, CLONE/S,DATES/S,NOPRO/S,COM/S:

Creates a copy of files or a directory.

FROM The source file.
TO The target file.
ALL Copies the entire directory.
QUIET Displays no output to the screen.

BUF-BUFFER
Uses BUF 512K buffers for copying.
CLONE Date, Status bits and comments are also copied.
DATES Date is also copied.
NOPRO The Status bits are reset when copied.
COM The comments are also copied.

V2.0 COPY FROM/A/M,TO/A,ALL/S,QUIET/S, BUF=BUFFER/K/N, CLONE/S,DATES/S, NOPRO/S,COM/S,NOREQ/S:

FROM Multiple files may be copied.
BUF Specified as numeric.
NOREQ No requesters will be displayed if an error is encountered.

DATE `DATE,TIME,TO=VER/K`

Input or output of date and/or time.

DATE	The date to be input.
TIME	The time to be input.
To=VER	The name of the file into which the date or the time is written.

V2.0 `DAY`

DAY	Advances date to next day input. Version 2.0 also allows numeric input into the month field.

DELETE `,,,,,,,,,,,All/S,Q=QUIET/S:`

Erases files and/or directories.

,,,,,,,,,,	Ten files or directory names to be deleted.
ALL	The entire directory is deleted.
Q=QUIET	There is no message output to the screen.

V2.0 `FILE/M/A,ALL/S,QUIET/S,FORCE/S:`

FILE	Multiple files or directory names to be deleted.
FORCE	Forces deletion, even if file is in use.

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

Displays the directory of a disk.

DIR	Name of the disk drive or the directory (pathname).
OPT	Allows input of abbreviations, A=ALL, D=DIRS, F=FILES, and I=INTER.
ALL	Shows all files in the directory including its subdirectories and their contents.
DIRS	Displays only directories.
FILES	Displays only files.
INTER	The contents are interactively output. After each file or directory the following inputs can be made:

?	Displays the possible commands.
B	Back up the directory (directory only).
E	Enter the displayed directory (directory only).
T	Type the file (files only).
Del	The file is deleted.
Q	Quit the `Dir` command.

Note: When using these arguments (`ALL`, `DIRS`, `FILES`, `INTER`) do not include the OPT argument.

DISKCHANGE DEVICE/A

Tells AmigaDOS that a disk has been changed.

DEVICE Which drive has experienced a disk change.

DISKCOPY [FROM] <disk> TO <disk> [NOVERIFY] [MULTI] [NAME <name>]

Creates a copy of a disk.

FROM <disk>
The source drive.
TO <disk> The destination drive.
NOVERIFY No verification performed during the copy.
MULTI Multiple copies on a single master may be made.
NAME Name Names the copy Name.

DISKDOCTOR DRIVE/A

Repairs errors on a disk. Damaged files may or may not be removed.

DRIVE The drive the program will attempt to recover.

ECHO ,NOLINE/S,FIRST/K,LEN/K:

Sends a text to the current output path, usually the screen.

, Text that is output to the current output path.
NoLines After the output of the given strings, the output doesn't jump to a new line.
First n The starting position of the text to be output.
Len n The length of the text to be output.

V2.0 Command made an AmigaDOS internal command and FIRST and LEN were specified as numeric.

ED/EDIT

Used to edit text files. See Section 2.4 for details and Sections 10.1 and 10.2 for the ED and EDIT quick reference sections.

ELSE

Allows alternative conditions in script files (see IF).

V2.0 Command made an AmigaDOS internal command.

ENDCLI/ENDSHELL

Exits `CLI` or `Shell` window.

V2.0 Command made an AmigaDOS internal command.

ENDIF

Ends an `IF/ENDIF` construct in a script file (see `IF`).

V2.0 Command made an AmigaDOS internal command.

ENDSKIP

Script file resumes execution at line following this command during a `Skip`.

V2.0 Command made an AmigaDOS internal command.

EVAL VALUE1/A,OP,VALUE2,TO,LFORMAT/K:

Evaluates simple expressions.

Value1	Decimal, hex or octal value
OP	math operator: +, -, *, /, mod, &, |, ~,<<, >>,xor,eqv
Value2	Decimal, hex or octal value
TO	Optional
LFORMAT	Specifies output format:
	%Xn hex (n is number of digits)
	%On octal (n is number of digits)
	%N decimal
	%C character

EXECUTE NAME TEXT

Executes a script file.

NAME	The name of the script file to execute.
TEXT	The arguments passed to the file.

FAILAT RCKLIM/N

Sets the return error code limit or returns the current return error code limit.

RCLIM	Contains the size of the new Return error Code LIMit.

V2.0 Command made an AmigaDOS internal command.

FAULT /N,/N,/N,/N,/N,/N,/N,/N,/N,/N: (AmigaDOS 2.0)

Prints information about a specific error.

N The valid error number.

V2.0 Command made an AmigaDOS internal command.

FF -0, -N (AmigaDOS 1.3)

This command accelerates the text output on the screen. FF was written by C. Heath, used by permission of Microsmiths, Inc®.

-0 FastFont text output is turned on.
-N FastFont text output is turned off (**Note:** you should enter -N, not a number for N).

V2.0 Implemented internally in AmigaDOS 2.0.

FILENOTE FILE/A COMMENT/A

Inserts a comment into a file.

FILE Which file will receive the comment.
COMMENT The comment of the file.

V2.0 ALL/S,QUIET/S:.

ALL All files will receive the comment.
QUIET No text is displayed during command operation.

Format DRIVE <disk> NAME <Name> [FFS][NOICONS] [QUICK]

Formats a disk and gives it a name.

DRIVE Required to specify drive.
<disk> Location of the drive containing the disk to format.
NAME Required to specify Name.
<name> The formatted disk receives the name "Name."
FFS The FastFileSystem is used to format.
NOICONS Optional (the disk will not have an icon if this option is used).
QUICK Only formats root and boot blocks.

GET/GETENV NAME (AmigaDOS 1.3)

This command reads the contents of an environment variable.

NAME The label of the variable whose contents should be read.

V2.0 Command made an AmigaDOS internal command.

ICONX (AmigaDOS 1.3)

Assigns icon and data to a script file. This lets you access the script file from the Workbench using the mouse (see Chapter 6).

IF NOT/S,WARN/S,ERROR/S,FAIL/S,EQ/K,GT/K,GE/K, VAL/S,EXISTS/K:

This allows choices to be made in script files, based upon conditions.

NOT	Logical reversal of a condition.
WARN	Condition is fulfilled when error code is larger than or equal to 5.
ERROR	Condition is fulfilled when error code is larger than or equal to 10.
FAIL	Condition is fulfilled when error code is larger than or equal to 20.
Text1 EQ Text2	Condition fulfilled when `Text1` equals `Text2`.
GT/GT Val	Greater than and greater than or equal to. Val used for numeric calculations.
Exists Name	Condition fulfilled when file `Name` is accessible.

V2.0 Command made an AmigaDOS internal command.

INFO DEVICE

Displays information on the screen about connected disk drives.

Device	Specifies a device.

INSTALL DRIVE/A,NOBOOT/S,CHECK/S

Converts a blank formatted disk into a boot disk.

DRIVE	The drive which contains the disk to be installed.
NOBOOT	Makes the disk a non-bootable DOS disk.
CHECK	Checks to see if the disk is bootable and if the standard Amiga boot code is present.

V2.0 FFS/S

FFS	Use the FastFileSystem.

JOIN ,,,,,,,,,,AS=TO/K

Joins two or more files together.

,,,,,,,,,	First of the two files to be joined together.
,,,,,,,,,	Second of the two files to be joined together.
AS	The file to which the joined files are written.

V2.0 FILES/M

FILES Multiple files may be specified.

LAB Text

Defines a string as the branch label for a script file.

Text The string to be defined as a label.

V2.0 Command made an AmigaDOS internal command.

LIST DIR,P=PAT/K,KEYS/S,DATES/S,NODATES/S,
TO/K,SUB/K,SINCE/K,UPTO/K,QUICK/S,
BLOCK/S,NOHEAD/S,FILES/S,DIRS/S,
LFORMAT/K:

Lists data about files.

DIR Displays only information about the file in DIR.
P=PAT Displays only the files specified in Pattern.
KEYS Displays the number of header blocks of the file or directory.
DATES Displays the date.
NODATES Suppresses the date.
TO Sends the output to the file Name.
SUB Displays information about file whose name is contained in Text.
SINCE Displays only the files created since Date.
UPTO Displays only the files created before Date.
QUICK Displays the filename only.
BLOCK The file size is given in blocks.
NOHEAD The information is suppressed.
FILES Lists only the files.
DIRS Lists only the directories.
LFORMATt="Text"
The option causes the text in Text to be displayed. Entering %s serves as a place holder for the actual file name. Entering a second %s causes the filename to be displayed a second time. Entering three %s causes the first one to display the path description of the current file. The next two contain the filename. Entering four %s produces the path description for the first and third ones and the filename for the second and fourth.

V2.0 ALL/S

All Lists ALL files.

LoadWB -Debug

Loads the Workbench from the CLI or the Shell.

-Debug AmigaDOS 1.3 adds a hidden menu with the debugging commands Debug and FlushLibs.

V2.0 Delay

DELAY The DELAY option waits three seconds before continuing.
-Debug was removed as an option.

LOCK DRIVE/A,ON/S,OFF/S,PASSKEY:

Prevents or allows access to a hard drive partition.

DRIVE Contains the protected hard disk partition.
ON Prevents access to the hard drive partition. Access is restored after entering the password (max. 4 characters).
OFF Removes an existing password. This command functions only with Kickstart 1.3.
PASSKEY Four character password required for access.

MAKEDIR DIR/A

Creates a new directory with the name Name.

DIR The name of the new directory.

V2.0 DIR/M

DIR Multiple directories can be created.

MAKELINK FROM/A,TO/A,HARD/S: (AmigaDOS 2.0)

Creates a file that points to another file. When the first file is specified, the linked file is called.

FROM The name of the original file.
TO The name of the linked file.
HARD Files will not be linked across volumes.

MOUNT DEVICE/A, FROM/K

Mounts a device.

Device A new device name.
From Name Removes parameters from the file Name instead of the Devs/Mount-list file.

NEWCLI WINDOW FROM:
NEWSHELL WINDOW FROM:

Opens a new CLI.

WINDOW	(Con:x/y/Width/Height(/Text))
x	The X-position of the upper left corner of the new window.
y	The Y-position of the upper left corner of the new window.
Width	Window width in pixels.
Height	Window height in pixels.
Text	Title of the new window.
FROM Name	Accesses the script file Name after the new SHELL window opens; if no filename is given the default file is S:Shell-startup.

V2.0 Command made an AmigaDOS internal command.

PATH ,,,,,,,,,,ADD/S,SHOW/S,RESET/S,QUITE/S

Displays or changes the pathname.

ADD	Adds a path to the directory Name.
SHOW	Shows the current path.
RESET	Deletes all paths up to the C: directory and the path Name.
QUIET	Suppresses output from the current output channel.

V2.0 **PATH/M, REMOVE/S**

PATH	Multiple paths may now be added.
REMOVE	Individual paths may be removed.

Command made an AmigaDOS internal command.

PROMPT PROMPT:

Changes the Shell prompt string. The Shell in V1.3 can use %s to display the current directory.

PROMPT	Formats the prompt's appearance; %n displays the process number.

V2.0 Command made an AmigaDOS internal command.

PROTECT FILE/A,FLAGS,ADD/S,SUB/S

Determines the protection bits a file should have.

FILE	The name of the file to protect.
FLAGS	Sets the protection status.
R	The file can be read.
W	The file can be written to.
D	The file is deletable.
E	The file is executable.
	In V1.3 the Hidden (H), Script (S), Pure (P) and Archive (A) bits can be set or reset.
H	Hidden file.
S	The file can be started without execute (script files only).
P	The file can be placed in the `Resident` list.
A	The file is archived.
	The H and A bits function only with Kickstart 1.3.
+, *ADD*	Sets the status of the given `Status` bit.
-. *SUB*	Removes the status of the status bit.

V2.0 ALL/S,QUITE/S

ALL	Multiple files may now be protected.
QUIET	No messages are displayed.

QUIT RC

Stops execution of a script file and returns an error code.

RC	Return error Code.

V2.0 Command made an AmigaDOS internal command and RC specified as numeric.

RELABEL DRIVE/A,NAME/A

Changes the name of a disk.

DRIVE	The drive containing the disk to be renamed.
NAME	The new name of the disk.

REMRAD (AmigaDOS 1.3)

This command erases all files from the reset-resistant RAM disk. The `ramdrive.device` is also removed after the next boot.

RENAME FROM/A,TO=AS/A

Renames files.

FROM Name of the data which is to be renamed.
TO=AS The new name.

V2.0 FROM/A/M,QUIET/S

FROM Multiple files may now be protected.
QUIET No messages are displayed.

RESIDENT NAME,FILE,REMOVE/S,ADD/S,REPLACE/S,
PURE/S,SYSTEM/S: (AmigaDOS 1.3)

This command erases, replaces or includes a new command in the list of resident commands.

NAME The resident name.
FILE Contains the command that should be activated in the `Resident` list.
REMOVE Deletes the command from the list.
ADD The command is included in the list.
REPLACE Replaces an existing command of the same name in the list with the new version of the command.
PURE Checks Pure bit of the command to see if it is set.
SYSTEM Files added to the system portion of the resident list cannot be removed.

V2.0 Command made an AmigaDOS internal command and FORCE can be used instead of PURE.

RUN COMMAND

Runs a program as a background process.

COMMAND
An AmigaDOS command to run as a background process.

V2.0 Command made an AmigaDOS internal command.

SEARCH FROM/A,SEARCH,ALL/S,NONUM/S,QUIET/S,
QUICK/S,FILE/S:

Searches data for a string.

FROM The file to be searched.
SEARCH Text
The string to be searched for.
ALL Searches all directories and subdirectories.
NONUM Displays no line numbers if string is found.
QUIET No output is displayed.

QUICK	The output format is more compact.
FILE	Searches for the specified file then the character string.

V2.0 **FILE/A/M,QUIET/S**

FILE	Multiple files may now be searched.
QUIET	No messages are displayed.

SETCLOCK LOAD/S,SAVE/S,RESET/S

Transfer the system date and time to and from the clock.

Load	Loads date and time from the internal clock.
Save	Saves system date and time to the internal clock.

V2.0 **RESET/S**

RESET	Resets clock completely.

SETDATE FILE/A,DATE/A,TIME:

Inserts a date or time into data.

FILE	File into which the date and time are inserted.
DATE	The date assigned to the file.
TIME	The time assigned to the file.

V2.0 **ALL/S**

ALL	Multiple files can have their dates set.

SET/SETENV NAME, STRING/F: (AmigaDOS 1.3)

Assigns a string to an environment variable.

NAME	The label of the variable.
STRING	The character string to be assigned to the variable.

V2.0 **SET**

SET	Command also accessed by SET.

Command made an AmigaDOS internal command.

SETPATCH

Patches ROM in Kickstart, enhancing system software.

SKIP LABEL,BACK/S:

Jumps within a script file to a defined label.

LABEL	Contains the string defined as a label.
BACK	Jumps to the start of the script file before searching for the label.

V2.0 Command made an AmigaDOS internal command.

SORT FROM/A,TO/A,COLSTART/K:

Alphabetically sorts a file and saves it to another file.

FROM	The source filename.
TO	The new file the sorted data is written to.
COLSTART	The line after which the text is sorted.

V2.0 CASE/S,NUMERIC/S

CASE	The sort is case sensitive, uppercase first.
NUMERIC	The sort is numeric sensitive, letters first.

STACK SIZE:

Changes the stack size or returns the current size.

SIZE	The stack size in bytes.

V2.0 Command made an AmigaDOS internal command and `SIZE` parameter specified as numeric.

STATUS PROCESS,FULL/S,TCB/S,CLI=ALL/S,COM=COMMAND/K:

Outputs information about `CLI` processes.

PROCESS	Selects the task number which should be displayed.
FULL	Combines the `TCB` and `CLI` options.
TCB	Displays priority, stack size and global vector size.
CLI=ALL	Displays the status of the current command process.
Com=COMAND	Searches for the `CLI` command `COMMAND`.

V2.0 Command made an AmigaDOS internal command and `PROCESS` parameter specified as numeric.

TYPE FROM/A,TO/S,OPT/K,HEX/S,NUMBER/S:

Displays the contents of a file.

FROM	The source file.
TO	The destination file to which `Name1` is copied. If a name isn't given the file appears on the screen.
OPT	Allows using H an N abbreviation for Hex and Number.
NUMBER	The lines are displayed with line numbers.
HEX	The characters are displayed in hex and ASCII characters.

V2.0 Multiple files may be input.

UNALIAS NAME **(AmigaDOS 2.0)**

Removes an alias from the alias list.

NAME	The name of the alias to remove.

V2.0 AmigaDOS 2.0 internal command.

UNSET/UNSETENV NAME: **(AmigaDOS 2.0)**

Unsets an environmental variable.

NAME	The name of the variable to remove.

V2.0 AmigaDOS 2.0 internal command.

VERSION NAME,VERSION,REVISION,UNIT:

Displays the version and revision number of a device, library or Workbench diskette.

NAME	Library name.
VERSION	Set condition flag based on version number.
REVISION	Set condition flag based on revision number.
UNIT	Specify unit, for multi-unit devices.

WAIT /N,SEC=SECS/S,MIN=MINS/S,UNTIL/K:

Shifts the system to a pause mode.

N	Waiting time in n units.
SEC=SECS	Specifies the unit as seconds.
MIN=MINS	Specifies the unit as minutes.
UNTIL	Waits until the input time.

WHICH FILE/A,NORES/S,RES/S: (AmigaDOS 1.3)

This command searches for and displays the path of a command (helps locate the command's location on disk).

FILE	Name of the command to search for.
NORES	Suppress search in resident list.
RES	Limits the search to the resident list.

V2.0 ALL/S

ALL	You can look for multiple files.

WHY

Returns information about the last error that occurred.

V2.0 Command made an AmigaDOS internal command.

Appendix

Appendix

Command and editor sequences in the Shell

Using the <Ctrl> and <Esc> keys, sequences can be entered directly in the `CLI/Shell` or by using the `Echo` command inside a batch file that can effect the output. When the `Echo` command is used, the <Esc> key can be set using the character combination `*e`.

Escape sequences

<Esc>c	The contents of the `CLI/Shell` window are erased and all other modes are turned off.
<Esc>[0m	All other modes are turned off.
<Esc>[1m	Bold text is turned on.
<Esc>[2m	Color number 2 becomes the text color (black).
<Esc>[3m	Italic text is turned on.
<Esc>[30m	Color number 0 becomes the text color (blue).
<Esc>[31m	Color number 1 becomes the text color (white).
<Esc>[32m	Color number 2 becomes the text color (black).
<Esc>[33m	Color number 3 becomes the text color (orange).
<Esc>[4m	The text is underlined.
<Esc>[40m	Color number 0 becomes the background color (blue).
<Esc>[41m	Color number 1 becomes the background color (white).
<Esc>[42m	Color number 2 becomes the background color (black).
<Esc>[43m	Color number 3 becomes the background color (orange).
<Esc>[7m	The text becomes inverted.
<Esc>[8m	The text becomes invisible (blue).
<Esc>[nu	The `CLI/Shell` window becomes `n` characters wide.
<Esc>[nt	Number of lines in the `CLI/Shell` window is set to `n`.
<Esc>[nx	The left border is set at `n` pixels.
<Esc>[ny	The distance from the top is set at `n` pixels.

Control sequences

When entering control sequences you must press the <Ctrl> key and the corresponding letter key.

<Ctrl><h>	Deletes last character entered.
<Ctrl><i>	Moves cursor one tab position to the right.
<Ctrl><j>	Linefeed.
<Ctrl><k>	Moves cursor up one line.
<Ctrl><l>	Clears `CLI/Shell` window.
<Ctrl><m>	Same as <Return>.
<Ctrl><n>	Enables Alt character set.
<Ctrl><o>	Enables normal character set.
<Ctrl><x>	Deletes current line.
<Ctrl><\>	Marks the end of a file.

Printer Escape Sequences

The following printer escape sequences are translated using the printer drivers included in the `Preferences` editors.

Printer Escape sequence	Meaning
<Esc>c	Initialize (reset) printer
<Esc>#1	Disable all other modes
<Esc>D	Line feed
<Esc>E	Line feed + carriage return
<Esc>M	One line up
<Esc>[0m	Normal characters
<Esc>[1m	Bold on
<Esc>[22m	Bold off
<Esc>[3m	Italics on
<Esc>[23m	Italics off
<Esc>[4m	Underlining on
<Esc>[24m	Underlining off
<Esc>[xm	Colors (x=30 - 39 [foreground] or 40 - 49 [background])
<Esc>[0w	Normal text size
<Esc>[2w	Elite on
<Esc>[1w	Elite off
<Esc>[4w	Condensed type on
<Esc>[3w	Condensed type off
<Esc>[6w	Enlarged type on
<Esc>[5w	Enlarged type off
<Esc>[2"z	NLQ on
<Esc>[1"z	NLQ off
<Esc>[4"z	Double strike on
<Esc>[3"z	Double strike off
<Esc>[6"z	Shadow type on
<Esc>[5"z	Shadow type off
<Esc>[2v	Superscript on
<Esc>[1v	Superscript off
<Esc>[4v	Subscript on
<Esc>[3v	Subscript off
<Esc>[0v	Back to normal type
<Esc>[2p	Proportional type on
<Esc>[1p	Proportional type off
<Esc>[0p	Delete proportional spacing
<Esc>[xE	Proportional spacing = x

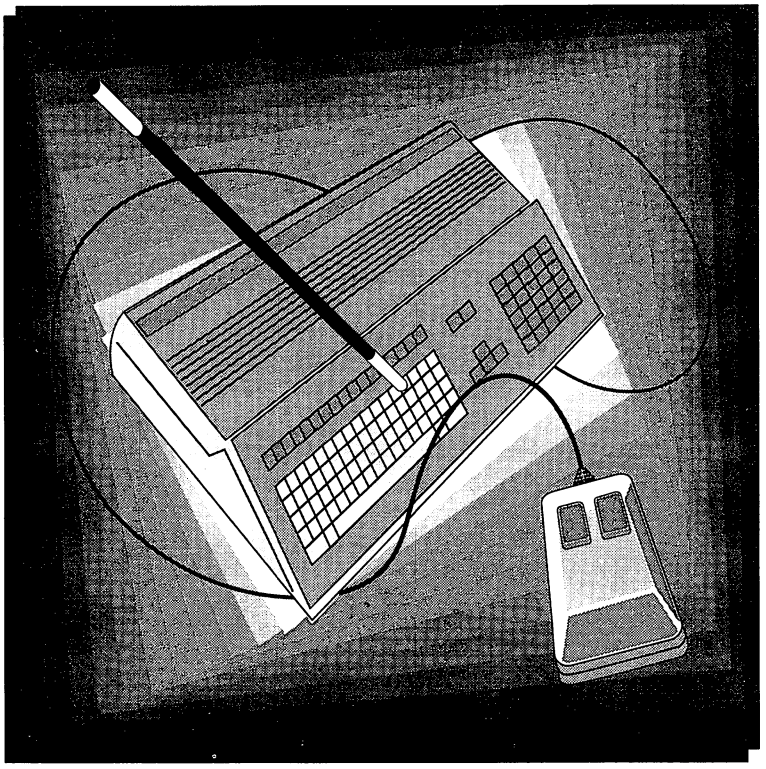
Printer

Escape sequence	Meaning
<Esc>[5F	Left justify
<Esc>[7F	Right justify
<Esc>[6F	Set block
<Esc>[0F	Set block off
<Esc>[3F	Justify letter width
<Esc>[1F	Center justify
<Esc>[0z	Line dimension 1/8 inch
<Esc>[1z	Line dimension 1/6 inch
<Esc>[xt	Page length set at x lines
<Esc>[xq	Perforation jumps to x lines
<Esc>[0q	Perforation jumping off
<Esc>(B	American character set
<Esc>(R	French character set
<Esc>(K	German character set
<Esc>(A	English character set
<Esc>(E	Danish character set (Nr.1)
<Esc>(H	Swedish character set
<Esc>(Y	Italian character set
<Esc>(Z	Spanish character set
<Esc>(J	Japanese character set
<Esc>(6	Norwegian character set
<Esc>(C	Danish character set (Nr.2)
<Esc>#9	Set left margin
<Esc>#0	Set right margin
<Esc>#8	Set header
<Esc>#2	Set footer
<Esc>#3	Delete margins
<Esc>[xyr	Header x lines from top; footer y lines from bottom
<Esc>[xys	Set left margin (x) and right margin (y)
<Esc>H	Set horizontal tab
<Esc>J	Set vertical tab
<Esc>[0g	Delete horizontal tab
<Esc>[3g	Delete all horizontal tabs
<Esc>[1g	Delete vertical tab
<Esc>[4g	Delete all vertical tabs
<Esc>#4	Delete all tabs
<Esc>#5	Set standard tabs

Index

Abacus

Amiga Catalog

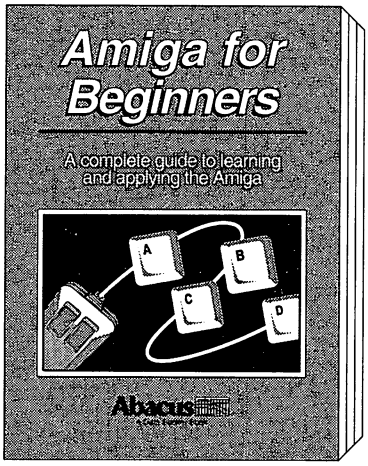


Order Toll Free 1-800-451-4319

Amiga for Beginners

Revised for 2.0

A perfect introductory book if you're a new or prospective Amiga owner. **Amiga for Beginners** introduces you to Intuition (the Amiga's graphic interface), the mouse, windows, the versatile CLI. This first volume in our Amiga series explains every practical aspect of the Amiga in plain English. Clear, step-by-step instructions for common Amiga tasks. **Amiga for Beginners** is all the info you need to get up and running.



Topics include:

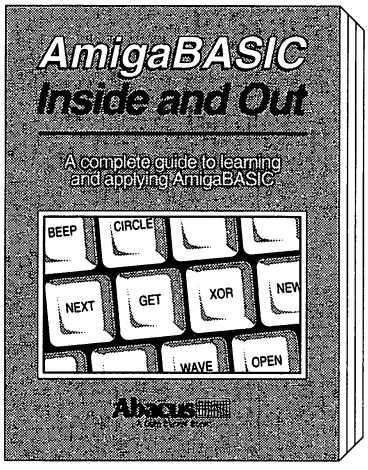
- Unpacking and connecting the Amiga components
- Starting up your Amiga
- Customizing the Workbench
- Exploring the Extras disk
- Taking your first step in AmigaBASIC programming language
- AmigaDOS functions
- Using the CLI to perform "housekeeping" chores
- First Aid, Keyword, Technical appendixes
- Glossary

Item #B021 ISBN 1-55755-021-2. Suggested retail price: $16.95

Companion Diskette not available for this book.

Amiga BASIC: Inside and Out

Amiga BASIC: Inside and Out is the definitive step-by-step guide to programming the Amiga in BASIC. This huge volume should be within every Amiga user's reach. Every Amiga BASIC command is fully described and detailed. In addition, **Amiga BASIC: Inside and Out** is loaded with real working programs.



Topics include:

- Video titling for high quality object animation
- Bar and pie charts
- Windows
- Pull down menus
- Mouse commands
- Statistics
- Sequential and relative files
- Speech and sound synthesis

Item #B87 ISBN 0-916439-87-9. Suggested retail price: $24.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

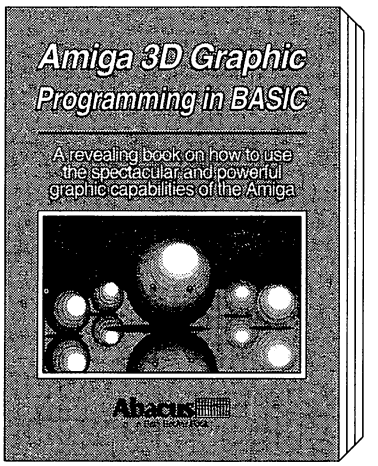
See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga 3D Graphic Programming in BASIC

Amiga 3D Graphic Programming in BASIC- shows you how to use the powerful graphics capabilities of the Amiga. Details the techniques and algorithm for writing three dimensional graphics programs: ray tracing in all resolutions, light sources and shading, saving graphics in IFF format and more.

Topics include:

- Basics of ray tracing
- Using an object editor to enter three-dimensional objects
- Material editor for creating parameters of color, shading and mirroring of objects
- Automatic computation in different resolutions
- Using any Amiga resolution (low-res, high-res, interlace, HAM)
- Different light sources and any active pixel
- Save graphics in IFF format for later recall into any IFF compatible drawing program
- Mathematical basics for the non-mathematician



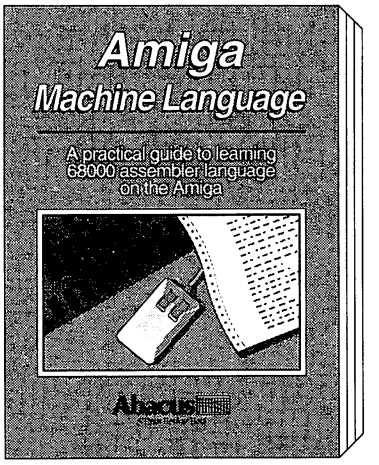
Item #B044 ISBN 1-55755-044-1. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

Amiga Machine Language

Amiga Machine Language introduces you to 68000 machine language programming presented in clear, easy to understand terms. If you're a beginner, the introduction eases you into programming right away. If you're an advanced programmer, you'll discover the hidden powers of your Amiga. Learn how to access the hardware registers, use the Amiga libraries, create gadgets, work with Intuition and more.

- 68000 microprocessor architecture
- 68000 address modes and instruction set
- Accessing RAM, operating system and multitasking capabilities
- Details the powerful Amiga libraries for access to AmigaDOS
- Simple number base conversions
- Text input and output - Checking for special keys
- Opening CON: RAW: SER: and PRT: devices
- Menu programming explained
- Speech utility for remarkable human voice synthesis
- Complete Intuition demonstration program including Proportional, Boolean and String gadgets



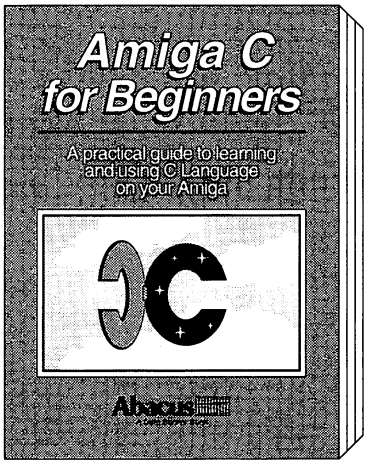
Item #B025 ISBN 1-55755-025-5. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga C for Beginners

Amiga C for Beginners is an introduction to learning the popular C language. Explains the language elements using examples specifically geared to the Amiga. Describes C library routines, how the compiler works and more.



Topics include:

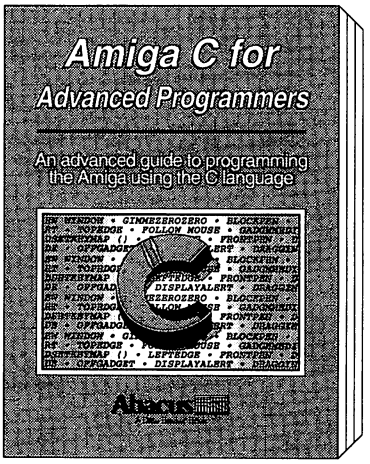
- Beginner's overview of C
- Particulars of C
- Writing your first program
- The scope of the language (loops, conditions, functions, structures)
- Special features of the C language
- Input/Output using C
- Tricks and Tips for finding errors
- Introduction to direct programming of the operating system (windows, screens, direct text output, DOS functions) Using the LATTICE and AZTEC C compilers

Item #B045 ISBN 1-55755-045-X. Suggested retail price: $19.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

Amiga C for Advanced Programmers

Amiga C for Advanced Programmers contains a wealth of information from the C programming pros: how compilers, assemblers and linkers work, designing and programming user friendly interfaces utilizing the Amiga's built-in user interface Intuition, managing large C programming projects, using jump tables and dynamic arrays, combining assembly language and C codes, using MAKE correctly. Includes the complete source code for a text editor.



Topics include:

- Using INCLUDE, DEFINE and CAST
- Debugging and optimizing assembler sources
- All about programming Intuition including windows, screens, pulldown menus, requesters, gadgets and more
- Programming the console device
- A professional editor's view of problems with developing larger programs
- Debugging C programs with different utilities

Item #B046 ISBN 1-55755-046-8. Suggested retail price: $34.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

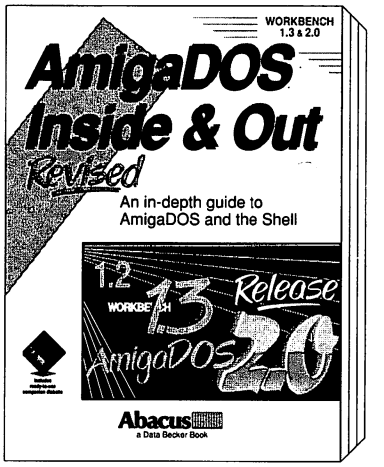
AmigaDOS: Inside & Out Revised

AmigaDOS: Inside & Out covers the insides of AmigaDOS, everything from the internal design to practical applications. **AmigaDOS Inside & Out** will show you how to manage Amiga's multitasking capabilities more effectively. There is also a detailed reference section which helps you find information in a flash, both alphabetically and in command groups. Topics include getting the most from the AmigaDOS Shell (wildcards and command abbreviations) script (batch) files - what they are and how to write them.

More topics include:

- AmigaDOS - Tasks and handling
- Detailed explanations of CLI commands and their functions
- In-depth guide to ED and EDIT
- Amiga devices and how the AmigaDOS Shell uses them
- Customizing your own startup-sequence
- AmigaDOS and multitasking
- Writing your own AmigaDOS Shell commands in C
- Reference for 1.2, 1.3 and 2.0 commands
- Companion diskette included

Item #B125 ISBN 1-55755-125-1. Suggested retail price: $24.95



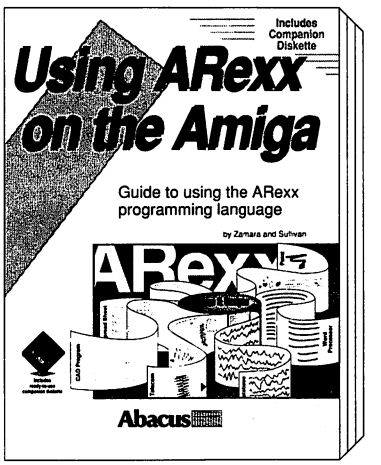
Using ARexx on the Amiga

Using ARexx on the Amiga is the most authoritative guide to using the popular ARexx programming language on the Amiga. It's filled with tutorials, examples, programming code and a complete reference section that you will use over and over again. **Using ARexx on the Amiga** is written for new users and advanced programmers of ARexx by noted Amiga experts Chris Zamara and Nick Sullivan.

Topics include:

- What is Rexx/ARexx - a short history
- Thorough overview of all ARexx commands - with examples
- Useful ARexx macros for controlling software and devices
- How to access other Amiga applications with ARexx
- Detailed ARexx programming examples for beginners and advanced users
- Multi-tasking and inter-program communications
- Companion diskette included
- And much, much more!

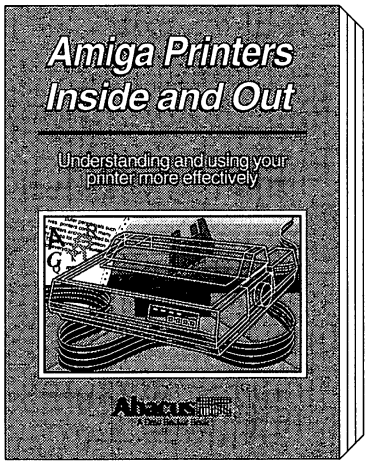
Item #B114 ISBN 1-55755-114-6. Suggested retail price: $34.95



See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga Printers: Inside & Out

Your printer is probably the most used peripheral on your Amiga system and probably the most confusing. Today's printers come equipped with many built-in features that are rarely used because of this confusion. This book shows you quickly and easily how to harness your printer's built-in functions and special features.



Topics include:

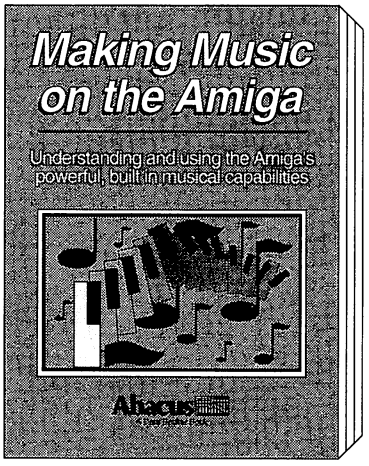
- How printers work, and why they do what they do
- Basic printer configuration using the DIP switches
- AmigaDOS commands for simple printer control
- Printing tricks and tips from the experts
- Recognizing and fixing errors
- WORKBENCH Printer drivers explained in detail
- Amiga fonts as printer fonts and much more!

Item #B087 ISBN 1-55755-087-5. Suggested retail price: $34.95

Companion Diskette Included at no additional cost: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings.*

Making Music on the Amiga

The Amiga has an orchestrs deep within it, just waiting for you to give the downbeat. **Making Music on the Amiga** takes you through all the aspects of music development on this great computer. Whether you need the fundamentals of music notation, the elements of sound synthesis or special circuitry to interface your Amiga to external musical instruments, you'll find it in this book.



Topics include:

- Basics of sound generation
- Music programming in AmigaBASIC
- Hardware programming in GFA BASIC
- IFF formats (8SVX and SMUS)
- MIDI fundamentals: Concept, function, parameters, schematics and applications
- Digitization: Capture and edit sound, schematics, applications
- Applications: Using Perfect Sound, Aegis Sonix, Deluxe Music Construction Set, Deluxe Sound Digitizer, Audio Master and Dynamic Drums

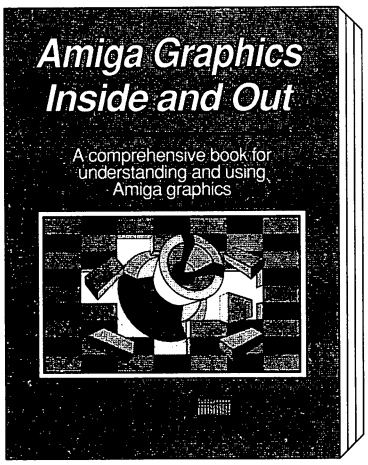
Item #B094 ISBN 1-55755-094-8. Suggested retail price: $34.95

Companion Diskette Included at no additional cost: *Contains public domain sound sources in AmigaBASIC, C, GFA BASIC and assembly language.*

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Amiga Graphics: Inside & Out

Amiga Graphics: Inside & Out will show you the super graphic features and functions of the Amiga in detail. Learn the graphic features that can be accessed from AmigaBASIC or C. The advanced user will learn how to call the graphic routines from the Amiga's built-in graphic libraries. Learn graphic programming in C with examples of points, lines, rectangles, polygons, colors and more. Complete description of the Amiga graphic system- View, ViewPort, RastPort, bitmap mapping, screens, and windows.



Topics include:

- Accessing fonts and type styles in AmigaBASIC
- Loading and saving IFF graphics
- CAD on a 1024 x 1024 super bitmap, using graphic library routines
- Access libraries and chips from BASIC- 4096 colors at once, color patterns, screen and window dumps to printer
- Amiga animation explained including sprites, bobs and AnimObs, Copper and blitter programming

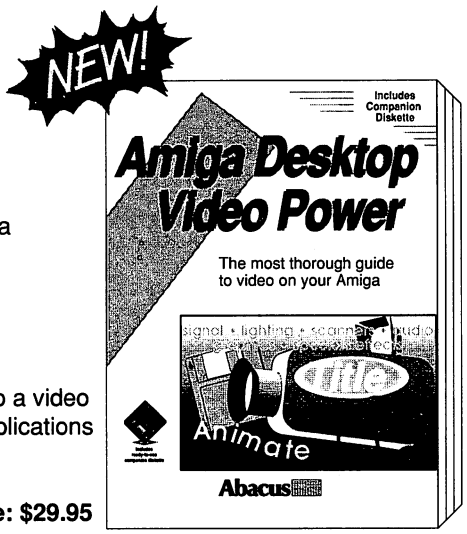
Item #B052 ISBN 1-55755-052-2. Suggested retail price: $34.95

Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **$14.95**

Amiga Desktop Video Power

Amiga desktop Video Power is the most complete and useful guide to desktop video on the Amiga. **Amiga Desktop Video Power** covers all the basics- defining video terms, selecting genlocks, digitizers, scanners, VCRs, camera and connecting them to the Amiga.

Just a few of the topics described in this excellent book:



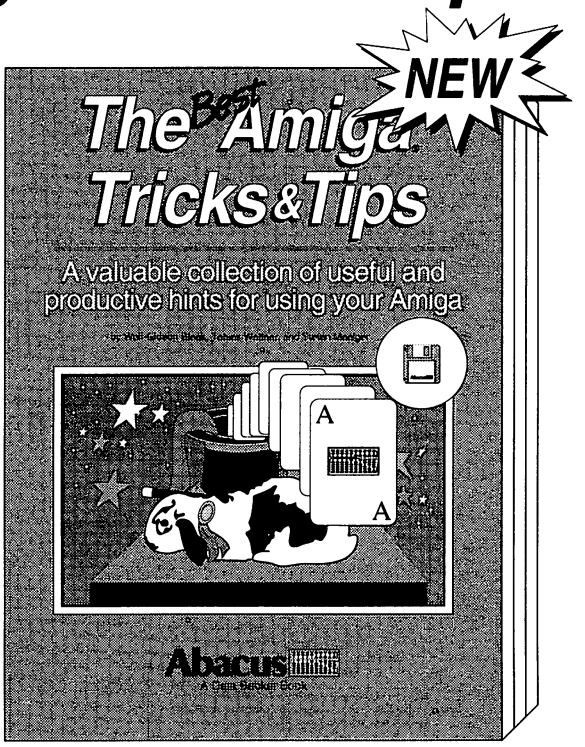
- Now includes DCTV, Video Toaster info
- The basics of video
- Genlocks
- Digitizers and scanners
- Frame Grabbers/ Frame Buffers
- How to connect VCRs, VTRs, and cameras to the Amiga
- Animation
- Video Titling
- Music and videos
- Home videos
- Advanced techniques
- Using the Amiga to add or incorporate Special Effects to a video
- Paint, Ray Tracing, and 3D rendering in commercial applications
- Companion diskette included

Item #B057 ISBN 1-55755-122-7. Suggested retail price: $29.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

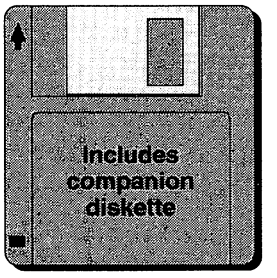
The Best Amiga Tricks & Tips

The Best Amiga Tricks & Tips is a great collection of Workbench, CLI and BASIC programming "quick-hitters", hints and application programs. You'll be able to make your programs more user-friendly with pull-down menus, sliders and tables. BASIC programmers will learn all about gadgets, windows, graphic fades, HAM mode, 3D graphics and more.



The Best Amiga Tricks & Tips includes a complete list of BASIC tokens and multitasking input and a fast and easy print routine. If you're an advanced programmer, you'll discover the hidden powers of your Amiga.

- Using the new AmigaDOS, Workbench and Preferences 1.3 and Release 20
- Tips on using the new utilities on Extras 1.3
- Customizing Kickstart for Amiga 1000 users
- Enhancing BASIC using ColorCycle and mouse sleeper
- Disabling FastRAM and disk drives
- Using the mount command
- Writing an Amiga virus killer program
- Changing type-styles
- Learn kernal commands
- BASIC benchmarks
- Disk drive operations and disk commands
- Learn machine language calls.



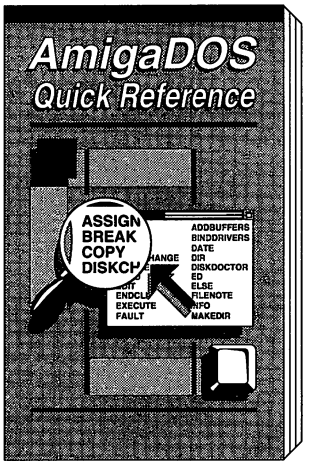
The Best Amiga Tricks & Tips includes companion disk. 410 pp.
Item # B107 ISBN 1-55755-107-3. Suggested retail price $29.95
Authors: Wolf-Gideon Bleek, Tobias Weltner, and Stefan Maelger.

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

AmigaDOS Quick Reference

AmigaDOS Quick Reference is an easy-to-use reference tool for beginners and advanced programmers alike. You can quickly find commands for your Amiga by using the three handy indexes designed with the user in mind. All commands are in alphabetical order for easy reference. The most useful information you need fast can be found including:

- All AmigaDOS commands described with examples including Workbench 1.3
- Command syntax and arguments described with examples
- CLI shortcuts
- CTRL sequences
- ESCape sequences
- Amiga ASCII table
- Guru Meditation Codes
- Error messages with their corresponding numbers



Three indexes for instant information at your fingertips! The **AmigaDOS Quick Reference** is an indispensable tool you'll want to keep close to your Amiga.

Item #B049 ISBN 1-55755-049-2. Suggested retail price: $9.95
Companion Diskette not available for this book.

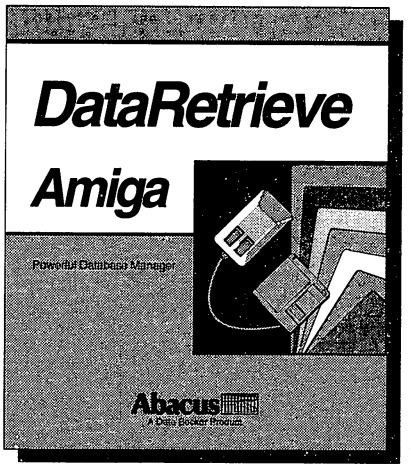
Abacus Amiga Book Summary		
Amiga for Beginners	Item #B021 1-55755-021-2	$16.95
AmigaBASIC: Inside and Out	Item #B87 0-916439-87-9	$24.95
Amiga 3D Graphic Programming in BASIC	Item #B044 1-55755-044-1	$19.95
Amiga Machine Language	Item #B025 1-55755-025-5	$19.95
AmigaDOS: Inside and Out	Item #B041 1-55755-041-7	$19.95
Amiga Disk Drives: Inside and Out	Item #B042 1-55755-042-5	$29.95
'C' for Beginners	Item #B045 1-55755-045-X	$19.95
'C' for Advanced Programmers	Item #B046 1-55755-046-8	$34.95
Amiga Graphics: Inside & Out	Item #B052 1-55755-052-2	$34.95
Amiga Desktop Video Guide	Item #B057 1-55755-057-3	$19.95
Amiga Printers: Inside & Out w/ disk	Item #B087 1-55755-087-5	$34.95
Making Music on the Amiga w/disk	Item #B094 1-55755-094-8	$34.95
The Best Amiga Tricks & Tips w/ disk	Item #B107 1-55755-107-3	$29.95
AmigaDOS Quick Reference	Item #B049 1-55755-049-2	$9.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

DataRetrieve

A Powerful Database Manager for the Amiga

Imagine a powerful database for your Amiga: one that's fast, has a huge data capacity, yet is easy to work with. Now think **DataRetrieve.** It works the same way as your Amiga- graphic and intuitive, with no obscure commands. Quickly set up your data files using convenient on-screen templates called masks. Select commands from the pulldown menus or time-saving shortcut keys. Customize the masks with different text fonts, styles, colors, sizes and graphics. If you have any questions, Help screens are available at the touch of a button. **DataRetrieve** is the perfect database for your Amiga.



Features inclde:

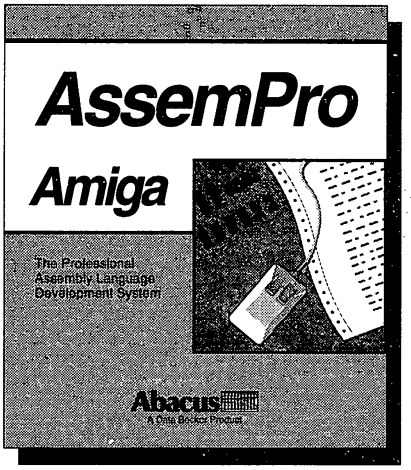
- Enter data into convenient screenmasks
- Work with 8 databases concurrently
- Define different field types: text, date, time, numeric and selection
- Customize 20 function keys to store macro commands and text
- Specify up to 80 index fields for superfast access to your data
- Perform simple or complex data searches
- Create subsets of a larger database for even faster operation
- Exchange data with other packages: form letters, mailing lists
- Produce custom printer forms: index cards, labels, Rolodex cards, etc. Adapts to most dot-matrix and letter-quality printers
- Protect your data with passwords
- Get Help from online screens
- Not copy protected

Item #S028 ISBN 1-55755-028-X. Suggested retail price: $79.95

AssemPro

Assembly Language Development System for the Amiga

AssemPro also has the professional features that advanced programmers look for. Lots of "extras" eliminate the most tedious, repetitious and time-consuming machine language programming tasks. Like syntax error search/replace functions to speed program alterations and debugging. And you can compile to memory for lightning speed. The comprehensive tutorial and manual have the detailed information you need for fast, effective programming.



Features inclde:

- Integrated editor, debugger, disassembler and reassembler
- Large operating system library
- Runs under CLI and Workbench
- Produces either PC-relocatable or absolute code
- Create custom macros for nearly any parameter
- Error search and replace functions
- Menu-controlled conditional and repeated assembly
- Full 32-bit arithmetic
- Advanced debugger with 68020 single-step emulation
- Fast assembly to either memory or disk
- Written entirely in machine language
- Runs on any Amiga with 512K or more

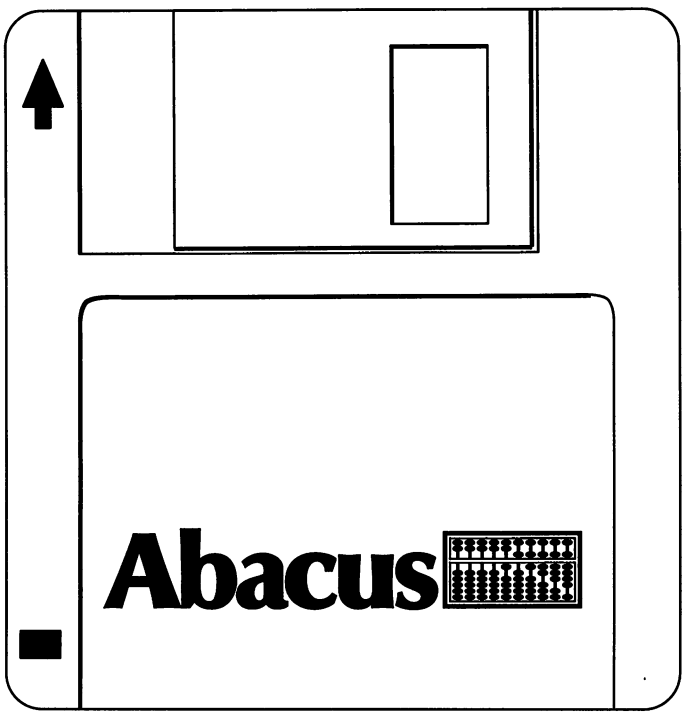
Item #S030 ISBN 1-55755-030-1. Suggested retail price: $99.95

Machine language programming requires a solid understanding of the Amiga's hardware and operating system. We do not recommend this package to beginning Amiga programmers.

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Cut carefully along this line

Companion Diskette Enclosed



Book/companion diskette packages:

- Save hours of typing in source code from the book.
- Provide **all script files and ARExx listings described in the book**; help avoid printing and typing mistakes.
- The companion diskette contains all of the **script files and ARExx listings** in this book..

If you bought this book without a diskette, call us today to order an economical companion diskette and save yourself valuable time.

Abacus

5370 52nd Street SE • Grand Rapids, MI 49512
Call 1-800-451-4319

Companion diskette contents

```
ARexx (dir)
      FindWord.rexx            GuessNum.rexx
      LoadLibrary.rexx         PrintFile.rexx
      RandNum.rexx             Rolldice.rexx
      Substitute.rexx          Test.rexx
C_Programs  (dir)
      .info                    DATAFILE
      evaluation               evaluation.c
      evaluation.o             READ-ME
      READ-ME.info             redirectinput
      redirectinput.c          redirectinput.o
      replace                  replace.c
      replace.o
Shareware-PD (dir)
      Misc Shareware AmigaDOS programs, see the
      read-me file for a complete description.
Printer_routines (dir)
      .info                    Bold
      Italics                  Nlq
      Reset                    Under_CLI_Only
      Under_CLI_Only.info
Script_Files (dir)
      .info                    Alias.BAT
      Backup                   Commands
      printer                  Radcontrol
      RamControl               Read_Me
      Read_Me.info             Types
      Window                   workdisk
      workdisk.info
```

WORKBENCH 1.3 & 2.0

AmigaDOS® Inside & Out Revised

AmigaDOS Inside & Out Revised is for Amiga users who want to make the most of their Amigas. **AmigaDOS Inside & Out Revised** covers the insides of AmigaDOS from internal design up to practical applications.

You'll learn abut the CLI (Command Line Interface), the user interface of AmigaDOS. **AmigaDOS Inside & Out Revised** explains AmigaDOS in plain, simple English so you'll be a CLI expert in no time!

AmigaDOS Inside & Out Revised takes you step by step through the practical aspects of AmigaDOS. **AmigaDOS Inside & Out Revised** will show you how to manage Amiga's multi-tasking capabilities more effectively.

There is also a quick reference section providing you with fast access to information you need immediately. This quick reference section is arranged alphabetically for finding a command easily.

AmigaDOS Inside & Out Revised topics include: detailed explanations of CLI commands and their functions, getting the most from the CLI (wildcards and command abbreviations), script (batch) files, what they are and how to write them.

US $24.95

ISBN 1-55755-125-1

An in-depth guide to AmigaDOS and the Shell

Workbench 1.3 commands explained:

- AmigaDOS - tasks and handling
- Amiga devices and how the CLI uses them
- Customizing your own startup sequence
- AmigaDOS and multi-tasking
- Writing your own CLI commands in C
- Resetting priorities - the TaskPri command
- In-depth guides to ED and EDIT
- Reference for 1.2 and 1.3 CLI commands
- Includes ARexx command summary and sample application programs.

AmigaDOS Inside & Out Revised will teach you how to use CLI commands to take full control of your Amiga.

Includes ready-to-use companion diskette

Abacus
5370 52nd Street SE • Grand Rapids, MI 49512

Amiga is a registered trademark of Commodore-Amiga Inc.