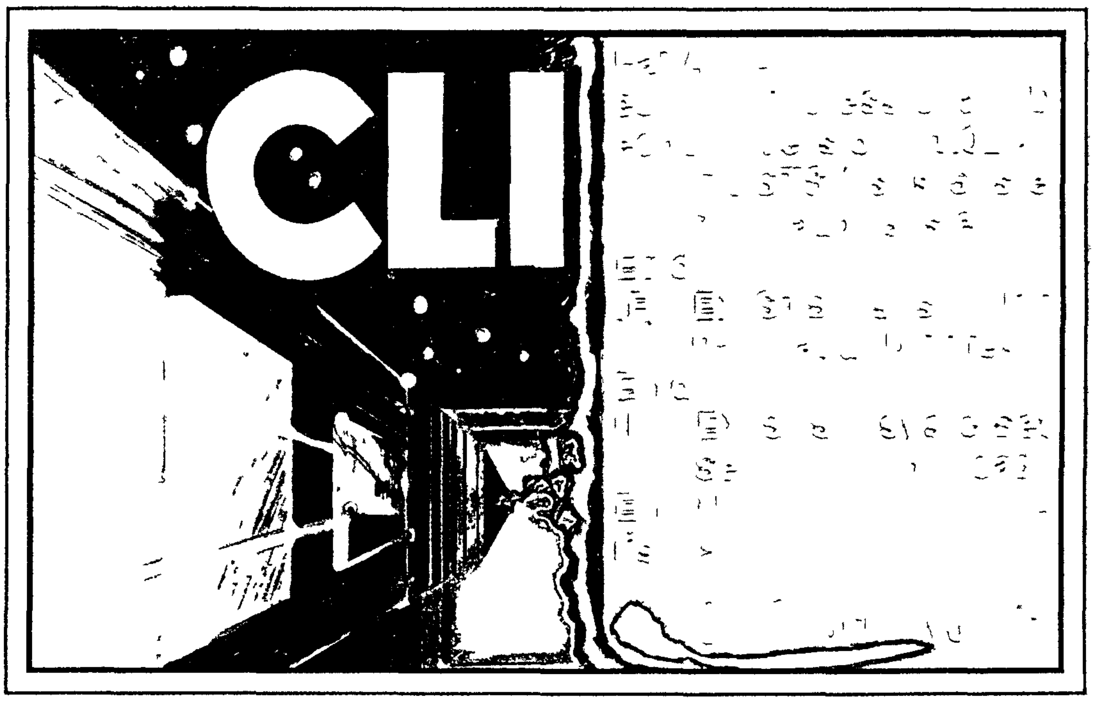
# AmigaDOS® Inside & Out

In-depth [illegible]
AmigaDOS and the



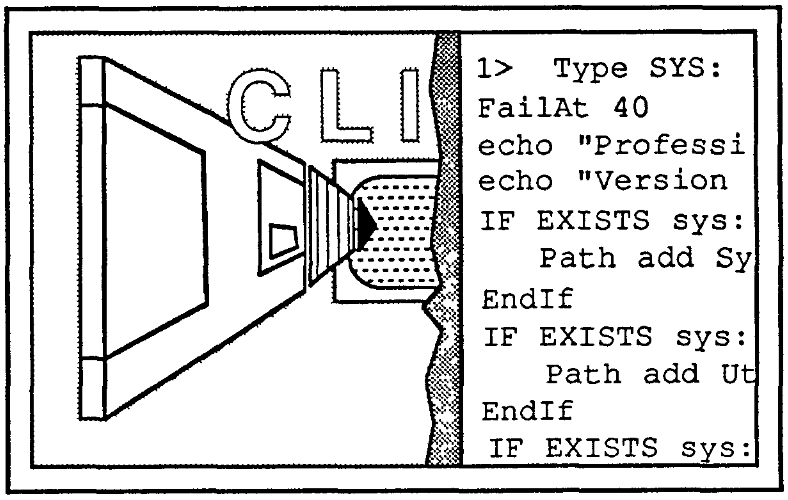
Abacus
A Data Becker Book

Includes WORKBENCH 1.3

AmigaDOS Inside & Out

# AmigaDOS Inside and Out

Kerkloh
Tornsdorf
Zoller



```
1>  Type SYS:
FailAt 40
echo "Professi
echo "Version
IF EXISTS sys:
   Path add Sy
EndIf
IF EXISTS sys:
   Path add Ut
EndIf
IF EXISTS sys:
```

A Data Becker book

Published by

Abacus

# 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 CLI icon on the Workbench disk. A NewCLI window appears and the boring cursor of the Command Line Interface 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 the CLI when the Amiga is turned on. For better or for worse, you now have to work with the CLI (Command Line Interface).

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

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

***Note:*** One final comment: The Amiga is an ever expanding system and the Workbench is constantly being improved. This book covers both Workbench 1.2 and 1.3. These new system disks work so much better than the older versions that we had to make you aware of the additions. Chapter 4 covers Version 1.3 in detail, and any differences between the two Workbench versions are pointed out as they appear in this book. This book supports Workbench/Kickstart 1.2 and Workbench/Kickstart 1.3.

*The Authors    Münster, June 1988*

# 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 `CLI` as easy as possible. In fact, the only `CLI` commands that appear in the following sections of Chapter 1 are the necessary ones. For those who wish to experiment further, Chapter 6 contains more background information on the `CLI`.

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!

***Note for Workbench 1.3 users:***

Workbench 1.3 has two AmigaDOS access programs: The `CLI` (contained in the `System` drawer) and the `Shell`, which is in the `Workbench 1.3` window. Please use the `CLI` for most of your work with this book. The `Shell` is an upgraded version of the `CLI`, which is explained in Chapter 4. Once you have become familiar with the `CLI`, you will probably use the `Shell` exclusively, but please use the `CLI` for the examples in this book.

## 1.1 The Task of DOS

***What is DOS?***

Before we begin working with the `CLI`, we must first explain briefly the function of DOS. *DOS* is the abbreviation for *Disk Operating System*. You may already know the definition of an *operating system*: The program(s) that control the computer (tell 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 (keyboard, disk drive, printer, etc.) control and coordinating various functions. It also makes established program functions available. 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: the DOS commands are short programs that need to be loaded from a disk drive (floppy disk, hard disk or RAM disk) before they can execute. Upon execution, DOS 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. Version 1.3 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 DOS as needed.

The biggest disadvantage of a separate DOS 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.

## 1.2 The Workbench and the `CLI`

The previous section gave a rough description of what DOS does. DOS contains the tools with which the user can preform 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: There is a menu item in the `Disk` menu named `Initialize`. You insert the blank disk, click once on its icon and select the `Initialize` item from the `Disk` menu. 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.

***Command Line Interface***

An alternative to the Workbench is the *`CLI`* (Command Line Interface). The name says it all: 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 `CLI` task.

Isn't the `CLI` 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 the `CLI`. The *startup sequence*, a file made of commands instructing the Amiga what to do or load as it starts up, can only be edited from the `CLI`. This startup sequence is located on the Workbench disk, and the Amiga looks for this file every time you turn the Amiga on.

In addition, some of the filenames on a disk are not visible on the Workbench for a number of reasons (e.g., an invisible file may have no matching `info` file). As a result, the `CLI` provides the only possible way to really look behind the scenes in Amiga disks.

## 1.3 Preparations

***Backup copies***

Before you begin working with the `CLI`, 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 (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.

***Amiga 1000 users only:***

- Place the Kickstart disk in the internal disk drive, sometimes called drive df0: (drive floppy 0). Turn on your Amiga and wait until the hand icon holding the Workbench disk icon appears. Remove the Kickstart disk and insert the original Workbench disk in the internal disk drive (df0:). Skip the next step (for Amiga 500 and Amiga 2000 users) and continue with the following step.

***Amiga 500 and 2000 users only:***

- Place the original Workbench disk in the internal disk drive, sometimes called drive df0: (drive floppy 0). Turn on your Amiga. The loading process begins immediately.
- After a while the Workbench screen appears. The loaded Workbench disk is represented by an icon in the upper right corner of 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. 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 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 `Duplicate` 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 will ask that you exchange the FROM (source) and TO (destination) disks several times. **Never remove a disk from a disk drive when the red light is on** (you could lose data, and even destroy the disk)!!! A window in the upper left corner of the Workbench 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 1.3**, the new disk has the name **Copy of Workbench 1.3**. Remove this extension using the `Rename` item from the `Workbench` menu (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 topically 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 CLI

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 of a hand holding a Workbench disk appears on the screen. Insert your backup copy of the Workbench disk. The system boots and the Workbench screen appears.

***Workbench 1.2 users only***

Workbench 1.2 users will have to make sure that they can access the `CLI`. Double-click on the disk icon. The `Workbench` window opens. Look for the `Preferences` icon (the icon of an Amiga 1000 case with a question mark superimposed over it). Double-click the `Preferences` icon. The `Preferences` window appears.

Look for the text marked `CLI`. There are two gadgets next to the `CLI` text: The `On` gadget and the `Off` gadget. Click on the `On` gadget if it is not already clicked on (it will be a different color from the `Off` gadget if it's on; the `Off` gadget will be a different color if the `CLI` is off). This will allow you to access the `CLI` from the Workbench. Click on the `Save` gadget in the `Preferences` window. The `Workbench` window reappears.

***Workbench 1.2 and 1.3 users***

Double-click on the `System` drawer icon in the `Workbench` window. The `System` window opens. Look for the `CLI` icon. Double-click on this icon.

The `CLI` loads, and a window named `NewCLI` appears on the screen.

The `NewCLI` 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 front gadget and a back gadget (for moving the window into the foreground or background of the screen). However, the `NewCLI` window has no close gadget: You must use a `CLI` command to close the window (more on this in Section 1.8).

The only thing displayed in the `NewCLI` window is the *DOS prompt*. This consists of a number 1 and a greater-than character (`1>`). This character tells the user 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 `CLI` 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 position 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. The `CLI` responds with:

```
Unknown command files
```

Only commands available as programs in the disk drive can be executed. This is the special feature of AmigaDOS. The `CLI` 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 `CLI` can execute programs as well as DOS commands.

Move the mouse pointer to the top of the `NewCLI` 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 `NewCLI` 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.

**Dir**

We'll begin with a relatively simple but very important command, and list other commands as you gain experience in the CLI. 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 CLI. CLI commands even accept mixed case letters.

After a while, the CLI 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>dir
   Trashcan (dir)
   C (dir)
   Prefs (dir)
   System (dir)
   l (dir)
   devs (dir)
   s (dir)
   t (dir)
   fonts (dir)
   libs (dir)
   Empty (dir)
   Utilities (dir)
   Expansion (dir)
   .info                 Disk.info
  Empty.info             Expansion.info
  Prefs.info             Shell
  Shell.info             System.info
  Trashcan.info          Utilities.info
1>
```

## 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. You'll notice that most of the filenames in the Workbench appear in two forms: Once under their usual names (e.g., Preferences for 1.2, Shell for 1.3); and again with an added extension of .info (e.g., Preferences.info for 1.2, Shell.info for 1.3). *Info files* contain icon data, date and time information and comments. You cannot see some files from the Workbench screen because these files don't have matching .info files. However, you don't need .info files when you work in the CLI. The Preferences program, for example, is capable of executing without an .info file.

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 the CLI. You can't see all the data files in the CLI 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 the CLI, 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 can go no 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 see it in the Workbench window. However, we can view the contents of this directory from the CLI. Enter the following command:

```
dir devs
```

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

```
1>dir devs
   keymaps (dir)
   printers (dir)
   clipboards (dir)
 clipboard.device                  MountList
 narrator.device                   parallel.device
 printer.device                    serial.device
 system-configuration
1>
```

You'll immediately see that there are three 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 users will find these printers on the Extras disk; they should enter: dir "Extras 1.3:devs/printers"):

```
1>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>
```

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

The Preferences program retrieves 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 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/Letters/Bills
```

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 CLI 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 are included with CLI commands, but aren't necessarily CLI commands themselves. The actual CLI commands are in a different directory.

You can view the CLI commands by looking in directory c (dir) of the Workbench disk. Enter the following command to view the Amiga-DOS commands available to you:

```
dir c
```

## 1.7 Argument Templates

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

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

You could refer to Chapter 10 of this book to find the correct syntax, but it's often much faster to call the argument template for the command.

Enter the CLI 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 ?
```

The CLI displays:

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

The argument 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 two arguments available: Dir and OPT/K. 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 KEY. This signals the CLI to wait for keyboard input.

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).

Three possible qualifiers can appear in an argument template:

**/A (Argument)** This qualifier 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 certain parameter must appear as well. The parameters allowed and the functions executed depend on the respective CLI command (see Chapter Two 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—turn a command on or off, or switch the command to another mode.

If none of the three 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
```

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 CLI:

```
delete ?
```

The CLI 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. However, no input is required when no qualifiers are surrounded by commas.

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 display of the file deletion process on. The equal sign between the Q and the QUIET means that you can use either 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 an argument template, the cursor reappears in the same line as the argument template following the colon. You can now enter an argument or set of arguments without re-entering the command keyword.

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

## 1.8 Quitting the `CLI`

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

**`EndCLI`**

The `EndCLI` command closes the `CLI` window currently active.

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

```
endcli
```

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

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

# 2. The CLI Commands

# 2. The CLI Commands

This chapter lists each `CLI` command 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. Section 10 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 on the `CLI` (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:*** Workbench 1.3 contains a number of added arguments and commands, making it much more versatile than Workbench 1.2. Any differences in Workbench 1.3 will appear after the general description of the command, preceded by the text **Workbench 1.3 implementation:** in bold type.

## 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

***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. The `Workbench` menu contains an item named `Initialize`. DOS recognizes unformatted disks immediately and places a BAD name under the disk icon on the Workbench screen.

The `CLI Format` command requires more information than the Workbench's `Initialize` item. 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 `CLI`: arguments which cannot contain spaces, or the argument must be enclosed within quotation marks. 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. This `Trashcan` is completely unnecessary when using the `CLI`.

The `DRIVE` and `NAME` arguments must be input every time you use the `Format` command. If the syntax was entered correctly, the `CLI` 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. The `CLI` then responds with a `*** Break.`

If you wish to continue, press the <Return> key alone. The Amiga formats the disk in the drive. The `CLI` 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 hold in 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]
```

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*** 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.

***Boot blocks*** Formatting of the Boot blocks (cylinder 0, side 0, sectors 0 and 1) renews the boot program so the Amiga can eventually autostart. 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 `FileSystem` is used if `NOFFS` is entered.

A partition must be entered in the MountList if you want the Amiga to run under the new FastFileSystem. This MountList is found in the Devs drawer on the Workbench disk. It can be loaded and modified with ED (ed devs: mountlist—see ED for more information). Each partition of the hard drive not autoconfigured (i.e., doesn't already exist elsewhere) 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). In any case, save the contents of the formatted partitions to floppy disks before performing this format.

## 2.1.2 Dir

***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 argument 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 directory 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 (Directories) 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 (Interactive) 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/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 `CLI` window. If you use the <T> key to display programs or `CLI` 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 argument 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 `CLI` prompt. |

If you enter an incorrect command in interactive mode, the `CLI` 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,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 `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>
```

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

| | |
|---|---|
| Com<Return> | Allows the user to execute a `CLI` command either direct 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:
```

## 2.1.3 CD

***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 invoke the `CLI` immediately after startup, this response is the disk drive specifier (e.g., `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., `Workbench:System`).

All `CLI` commands now refer to the current directory. For example, if you enter `Dir` the `CLI` 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, the `CLI` 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 `Bad args` (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 invoking 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 `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.

## 2.1.4 MakeDir

***Syntax:*** `makedir /A`

This command performs much the same function as duplicating the Empty drawer from the Workbench. There you can make a duplicate of the Empty drawer, name the duplicate drawer whatever you want and drag files into the new drawer. 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 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.

## 2.1.5 Delete

***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
```

The CLI cannot delete a directory which still contains data. If you try to delete a directory that still had files in it, the CLI displays the error code 216 (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 3 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 from test1 through test7:

```
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 that it is doing so:

```
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 Delete 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.

## 2.1.6 Copy

***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 the CLI. 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 argument 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 exists, 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 text 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.

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 floppy 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 argument 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 DOS 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
```

## 2.1.7 List

***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 `Dir`, 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 `Preferences`, or if you have an Amiga with a battery-backup realtime 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.

```
>1 list df0:
Directory "df0:" on Sunday 18-Oct-87
Expansion.info                 346 rwed 02-Mar-87 23:29:10
Trashcan                       Dir rwed 02-Mar-87 23:29:10
.info                           82 rwed 09-Jun-87 12:43:46
c                              Dir rwed 02-Apr-87 09:44:21
Clock.info                     338 rwed 02-Mar-87 23:31:49
Demos                          Dir rwed 15-Apr-87 07:57:18
Clock                        19668 rwed 02-Mar-87 23:32:20
System                         Dir rwed 06-Apr-87 15:11:06
l                              Dir rwed 02-Mar-87 23:33:26
devs                           Dir rwed 02-Mar-87 23:35:49
s                              Dir rwed 16-Apr-87 08:43:34
t                              Dir rwed 06-Apr-87 15:08:45
Preferences.info               418 rwed 02-Mar-87 23:36:00
Preferences                  58136 rwed 02-Mar-87 23:36:20
Demos.info                     346 rwed 02-Mar-87 23:36:22
fonts                          Dir rwed 02-Mar-87 23:37:32
libs                           Dir rwed 02-Mar-87 23:37:59
Empty                          Dir rwed 09-Jun-87 12:46:57
Utilities.info                 346 rwed 02-Mar-87 23:38:08
Disk.info                      306 rwed 02-Apr-87 09:05:36
System.info                    346 rwed 02-Mar-87 23:38:13
Empty.info                     346 rwed 02-Mar-87 23:38:16
Trashcan.info                  430 rwed 02-Mar-87 23:38:17
Utilities                      Dir rwed 15-Apr-87 07:57:45
Expansion                      Dir rwed 15-Apr-87 07:35:10
12 files - 13 directories - 196 blocks used
```

There's more to `List` than you might think. Invoking the argument template (`List ?`) 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 ram: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` (subname) argument makes it possible to search for entries arranged according to their *subnames*. A subname 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 up to yesterday
```

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

The following command, would be entered on one line in the Amiga, it searches for the c subdirectory in drive **df0:** and looks for all the commands that end with `CLI`. The command then looks for all these

entries created after September 10, 1986 and sends all these entries to the printer (no dates appear on the printout):

```
list df0:c pat #?CLI nodates to prt: since
10-sep-86 upto today
```

## Workbench 1.3 implementation:

*Syntax:*

```
list DIR,P=PAT/K,KEYS/S,DATES/S,NODATES/S, TO/K,S/K,
SINCE/K,UPTO/K,QUICK/S,BLOCK/S,NOHEAD/S,FILES/SDIRS/S,LFO
RMAT/K
```

There are some very useful arguments added to this version:

**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 each CLI command 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.

## 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 CLI 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.

We found a bug when using the `Rename` command in conjunction with the RAM disk of Workbench 1.2. The following creates two directories named `Test` in the main directory of the RAM disk:

```
makedir ram:test
makedir ram:test/under
rename  ram:test/under ram:test
dir ram:
```

To solve this problem, rename one of the two directories immediately.

***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.

## 2.1.9 Diskcopy

***Syntax:***

```
diskcopy FROM/A,TO/A,NAME
```

The `Diskcopy` command is the `CLI` equivalent of the `Duplicate` item from the `Workbench` pulldown menu.

Unlike the `CLI 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>. The following message appears:

```
*** 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. You must remember to put the original disk back in the drive after aborting the copy procedure.

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 `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 1.2 and Version 1.3 of this command are identical.

## 2.1.10 Relabel

***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.

## 2.1.11 Info

***Syntax:*** `info DEVICE`

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
DF0: 880K   1645   113   93%    0    Read     A500 WB1.2 D
DF1: 880K   534   1224   30%    0    Read/Write TextPro

Volumes Available:
A500 WB1.2 D [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.

## 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, located in ROM on the Amiga 500 and 2000, should let 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 `CLI` 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
t
fonts
libs
```

In addition, you would have to write a *startup sequence* (see Chapter 6 for detailed information about startup sequences 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 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 CLI 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:. 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). In those situations contact your local dealer or a user's group as quickly as possible—they may be able to help you.

## 2.1.13 Type

***Syntax:*** `type FROM/A,TO.OPT/K`

The Type command displays ASCII files on the screen, 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...(|../ ./.
```

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`

## 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 argument 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.

## 2.1.15 Search

***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.

The command has a couple of extra options:

***Wildcards*** Like the new 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:?/#?cli search window
```

would find all the files ending with CLI in any directories containing one letter names, containing the word "window."

Like all of the CLI 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 new 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.

## 2.1.16 Sort

***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.

## 2.1.17 Protect

***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 goes 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
```

## 2.1.18 Filenote

***Syntax:*** `filenote FILE/A,COMMENT/A`

The `Filenote` command allows you to place up to 80 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 02-Mar-87 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.

## 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 realtime clocks in their Amigas—time is set without opening `Preferences`.

**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.

## 2.1.20 DiskDoctor

***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* **dfl:** (the first external disk drive on an Amiga 1000 or 500):

```
diskdoctor dfl:
```

***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 `DiskDoctors` are identical. The 1.3 program works better than the 1.2 program.

## 2.1.21 Diskchange

***Syntax:*** `diskchange DEV/A`

This command deals only with material for Amiga owners who use 5 1/4" disk 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.

### Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

## 2.2 CLI System Commands

The following section describes system commands, including the commands that are related to the CLI itself.

### 2.2.1 NewCLI

***Syntax:*** `newcli WINDOW, FROM`

***Multitasking*** The NewCLI command gives the Amiga user access to multitasking. 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 really runs tasks in alternation instead of in parallel (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 CLI window appears named after the current task (e.g., NewCLI task 2). 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 Chapter 5 for more information about multitasking.

Finally, a hint about 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:

```
newcli con:50/70/250/100/Amiga
```

This option isn't used very often for direct output because CLI windows don't have close gadgets or some other window gadgets. It works best when using the command in conjunction with a startup sequence.

If the size input is missing, the CLI creates a window the full width and half the height of the 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 copies
```

In this example the script file named Copies 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.

The NewCLI command has become obsolete. In the c directory on the new Workbench disk there is a new command called NewShell. This command creates a window port to DOS that has some 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 l: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 qualities:

***Resident commands supported*** DOS can load most of the CLI commands into working memory and is called Resident. These commands are then ready for use by the user. It is covered in detail in the handling of the new CLI commands in Chapter 3. Calling these commands is only possible through the Shell. In a typical CLI window such a command is loaded from the disk.

***Command synonyms allowed***

It's often a good idea to give your CLI 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 returned.

Try to erase your CLI command directory using Delete c. This can't be done because the directory is not empty. Instead of the error message Not Deleted–directory not empty, the message Not Deleted –Error code 216 appears. DOS also makes use of the CLI 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 the Shell 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 the Shell.

The description of the original command is not reduced 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 is put together 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 CLI-Startup. All entered relationships are valid in each Shell.

The relationship alias newcli newcon:0/10/640/100/ AmigaShell is found in this file. A window with the title AmigaShell is nothing other than a Shell that directs its input and output to the new device NewCon:. This device is responsible for you being

able to edit the lines in the Shell. The NewCon device is described in Chapter 4.

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

***Output of current path***

In the new Shell, the prompt represents the actual directory path. This informs 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 the CLI. 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 only 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, instead of the script file CLI-Startup, a script file with the name Shell-Startup is called from NewShell. This file is found in the s directory of the Workbench disk.

Further information on the NewShell command can be found in Chapter 3.

## 2.2.2 EndCLI

***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 in 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 statement `alias endshell endcli` so that `Shell` will accept the `EndShell` command.

## 2.2.3 Run

***Syntax:*** `run PROGRAM_NAME`

This command executes a program or `CLI` command while allowing access to a program running in the background *and* the current `CLI`. Any output from the `Run` command appears in the `CLI` window which started the task. The example below prints three files name `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. The `CLI` 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: +
type 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 `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.

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

## 2.2.4 Status

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

This command displays all the information available about the `CLI` tasks running at that particular time. If you enter `Status` without parameters, or if you enter `Status all`, the `CLI` 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 the `CLI` 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. 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) |

See Chapters 5 and 6 for more information about the Global Vector Table width and task priority.

**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 Amiga returns an error code of 5. This argument is especially helpful in script files for seeing if a background task is running.

## 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 displays the following:

```
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.

## 2.2.6 Break

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

This command halts execution of a DOS command from any CLI window. For example, if one 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 was started entirely from CLI input. 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 CLI command. In most cases, nothing happens.

**ALL**

The ALL argument transmits 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.

## 2.2.7 Path

*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 a directory that might be found in the RAM disk:

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

This list shows the order and directories used for searching a file. If the name of a program is entered (e.g., a `CLI` 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 `CLI` 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 looks like this:

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

Now DOS automatically looks in the `Utilities` drawer.

The `Path` command is especially useful when used in conjunction with the RAM disk. 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).

**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:` device 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, DOS first searches the resident commands. If it cannot find the command in residence, the search operation continues as described above.

## 2.2.8 Assign

*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 DOS. 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 `CLI` 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 accessed from 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 (DOS 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 called from the `CLI`. 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:

The 1.3 version of the `Assign` command allows you to see if a certain device exists. The command must include the device name and the `LIST` argument. The following entry is given if the device exists and the error status is set to zero. The following occurs when you use `Assign` in conjunction with the `fonts:` directory:

Input:

```
assign fonts: list
```

Output:

```
Volumes:
Directories:
fonts   Volume: A500 WB 1.2D
Devices:
```

The `fonts:` device appears under the `Directories:` heading. It is treated as a virtual device.

The `Assign` command returns error code 5 if the device is not found. This error status can be used in a script file (see Chapter 5 for more information about script files). 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: list
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 needs the `If Warn` command if the Extras disk isn't found (`warn` = 5) and displays the specified text.

## 2.2.9 AddBuffers

***Syntax:***

```
addbuffers DRIVE/A,BUFFERS/A
```

This command assigns a large buffer to a specified disk drive. When working in the `CLI`, sometimes a command can be loaded from the drive before it is used 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 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.

The `AddBuffers` command has a small disadvantage: The buffer memory allocated to disk memory is taken from system memory. 544 bytes of memory are required per block of memory. The additional 32 bytes are used for internal memory management. It is clear that the data must eventually be stored in the system memory. There is no clear reason why reserved buffer memory cannot be released again (this is only possible by restarting the system).

### Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

## 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 the CLI 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
```

A glance in the Appendix of this book or entering the command Fault 205 explains the error: Object not found. We purposely misspelled startup-sequence above.

## 2.2.11 Fault

*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 1.2 and Version 1.3 of this command are identical.

## 2.2.12 Date

*Syntax:*

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

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

**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:

```
Wednesday 21-Oct-88 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-Jun-88, you can also enter 1-Jun-88.

## 2.2.13 SetClock

***Syntax:***

```
setclock opt load|save
```

This command places the time and date set by `Date` into the Amiga battery-powered realtime clock (this is an option for Amigas). The realtime clock and the data entered in `Date` are independent of one another.

**OPT LOAD** The `OPT LOAD` argument transfers the realtime clock date and time to the system.

**OPT SAVE** The `OPT SAVE` argument transfers the system date and time to the realtime 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 Workbench disk. 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 realtime clock exists, the computer replies:

```
Internal clock not functioning
```

You will receive this message if you don't have a realtime 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:

Version 1.2 and Version 1.3 of this command are identical.

## 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 `CLI` 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 `CLI` task displayed, the combination `%n` must be entered. Example:

| | |
|---|---|
| Input: | `prompt "I am number %n !"` |
| Output: | `I am number 1 !` |

The old prompt can be restored by entering:

```
prompt %n>
```

### Workbench 1.3 implementation:

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 `CLI` 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 `CLI` task. The actual directory is the `System:` directory of the Workbench disk.

## 2.2.15 Stack

***Syntax:***

```
stack SIZE
```

This command specifies the amount of memory allocated for the stack. Each `CLI` 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 `CLI` 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 crash the computer):

```
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 amount of memory available, use the `Status` command without any parameters.

### Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

## 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 starting time shortens by a couple of seconds.

### Workbench 1.3 implementation:

Version 1.2 and Version 1.3 of this command are identical.

## 17 Mount

`VICE/A`

mand can add new devices to AmigaDOS. The basic configura-
iga recognizes the following devices:

l disk drive
er
allel port
erial port
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:<br>0,1 = Chip or Fast RAM<br>2,3 = Only Chip RAM<br>4,5 = Only Fast RAM |
| `Mount` | 1 = Device connected<br>-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:*** `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 4 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`: `echo "Hello CLI 2. How are you?" to pipe:`

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

Input to 2nd `CLI`: `type pipe:`
Output: `Hello CLI 2. How are you?`

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

**rad:** A 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 |

These statements are only evaluated in conjunction with the new Fast Filing System.

## 2.3 Script File Commands

This section contains information about the commands used in conjunction with script files. Script files (sometimes also called batch files) are simple text files containing any number of CLI commands, written using ED or a word processor. The Execute command runs these commands. Chapter 4 contains detailed explanations and several practical uses for script files.

### 2.3.1 Execute

*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
```

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

As with most `CLI` 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
```

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` command, 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 the case that 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:`).

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

```
                .dol #
```

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
```

## 2.3.2 Echo

***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`

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

## 2.3.3 Failat

***Syntax:*** `failat RClim`

This command halts a command sequence if the Amiga reaches a specified error return code limit. Each `CLI` command and many other programs return an error number if an error occurs during execution. In the `CLI` 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 a `CLI` 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 `CLI`). 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 `CLI`, the limit is also valid in the script file called from the `CLI`. 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 `CLI` called automatically supercedes the error limit of the `CLI` 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.

## 2.3.4 Quit

***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 `CLI` and the following text appears by an error number 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.

## 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, this time 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:

```
.ket 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:

Version 1.2 and Version 1.3 of this command are identical.

## 2.3.6 Ask

*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 `CLI`.

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.

## 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:

Version 1.2 and Version 1.3 of this command are identical.

## 2.3.8 Wait

*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 `CLI` window using the `Dir` command, and the `List df0:` command lists the directory of the same disk in another `CLI` 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 3 it shows how `Wait` can be used to make a `CLI` alarm clock.

## 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:

Version 1.2 and Version 1.3 of this command are identical.

## 2.4 The Editors

The `ED` and `Edit` programs are two large (19K each) programs that make it possible to edit text files.

**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.

**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 40000 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:

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>, 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 |

***<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`

***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. For example, `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.

## 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 the `CLI`. 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 lower case and upper case 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> 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 the `CLI`. The standard devices of the Amiga are listed below:

```
DF0
RAM
PAR
SER
PRT
CON
RAW
```

A colon (:) must always follow the device name, so that the `CLI` can tell devices apart from directories or filenames.

This chapter describes the individual device names and what you can do with these devices.

## 3.1 Floppy Disk Devices (df)

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 `CLI` commands default to drive **df0:** in the basic Amiga configuration (with only one disk drive). If you enter a `CLI` command without a disk drive specifier, the `CLI` 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.

## 3.2 The RAM Disk Device (ram)

The `ram:` device simulates a disk drive with the Amiga's working memory. 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. Since the `Guru Meditation` message can occur at any time, 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 application 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, 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 <u>not</u> 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 <u>did</u> 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 formatted before using it for the first time. If you click on the RAM DISK icon and 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:

```
Initialization failed -
not a disk
```

If you try to format the RAM disk using the `CLI Format` command, the `CLI` displays:

```
Format failed -
not a disk
```

**Relabel** You cannot assign another name to the RAM disk. If you try this, the following message appears:

```
Attempt to rename disk failed
```

**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.

## 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. This device must first be connected before access. 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.

## 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. Only the bottom seven bits of the byte can be transferred. The remaining bit can be sent as the *parity bit* (transfer control bit). After that, 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.

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 problems. There are three ways to achieve correct data transfer:

***Send correct 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 (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.

***Security reservations***

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 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 `Preferences` 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.

Our first try with the serial connection of the Amiga was disappointing. Only a fraction of the parameters that the operating system needs can be set using the `Preferences` program. We tried entering the following command sequence in the `CLI`:

```
Copy ser:command to con:10/10/300/100/output
```

The RS-232 input buffer requires time to fill. The data received first appears in the `Con:` window. The remaining data in the buffer is displayed when a new batch of data is transferred. An end signal cannot be transferred. See Section 5.7 for details about transferring data from Amiga to Amiga using the RS-232 port.

## 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 made in the `Preferences` program. 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 `Preferences` program that made it possible to put together custom drivers for any available printer.

## 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.

## 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 `CLI` window the output can be suppressed by other data. In this case the <Return> key should be pressed in the bottom window.

# 4. Workbench 1.3

# 4. Workbench 1.3

The Amiga developers have prepared Version 1.3 of the Workbench and Kickstart, featuring many added improvements. A few of the improvements:

- New and improved `CLI` commands
- A comfortable `Shell` in addition to the `CLI`
- `CLI` commands can be loaded and remain resident commands
- A `FastFile` system 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
- Boot possibilities 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 **df0:** (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).

Now we'll go on and describe the new functions.

## 4.1 New CLI commands in Workbench 1.3

Some new, very useful commands exist in the c directory on the Workbench 1.3 disk. We want to examine these new commands more closely.

### 4.1.1 Avail

*Syntax:*

```
avail
```

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.

### 4.1.2 FF

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 a boot disk). 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 sending it to the `nil:` device with:

```
ff >nil: -0.
```

The `-N` argument can be entered if the normal output mode is needed. The starting message will then appear without the `Turning on FastText` message.

### 4.1.3 Lock

*Syntax:*

```
lock DRIVE/A,ON/S,OFF/S,PASSKEY
```

This command write protects any partitions of a hard disk drive. The hard disk partitions must function under `FastFilingSystem` (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 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 beethoven:

```
lock dh1: on beethoven
```

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 beethoven
```

### 4.1.4 NewShell

***Syntax:***

```
newshell WINDOW, FROM
```

This command allows you to open another `Shell` window for DOS command entry. The `NewCLI` command also performs this function.

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 additional key combinations are evaluated:

| | |
|---|---|
| <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 special 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 remains invisible. However, the control code is still present and 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 l:Shell-Seg System
```

These commands are usually found in the startup sequence of the Workbench 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 immedieatly 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 `ex`. 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.

## 4.1.5 Remrad

*Syntax:* `remrad`

Abbreviation for (Remove Recoverable RAM Disk). This command erases the contents of the reset-resistant RAM disk 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.

## 4.1.6 Resident

*Syntax:* `resident NAME,FILE,DELETE/S,ADD/S,REPLACE/S,PURE/S,SYSTEM/S`

This command loads the user's favorite `CLI` commands into working memory. `CLI` 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.

Before the `Resident` command existed, the important commands were 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 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 `CLI`, the program is executed from the location in RAM.

A `CLI` 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 `CLI`. Example: The `CLI` window lists the directory of drive **df0:** while the `Dir` command is being used in the `CLI` window 2 for drive **df1:**. Most `CLI` 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, it responds with a 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 DOS that the program can be made resident using the `Resident` command. `List` the new `CLI` 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                       1
Dir                      1
Execute                  1
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 1. A 1 also means that the command is not 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`:

```
resident 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.

**DELETE** The `DELETE` argument eliminates an entry from the list of resident files. The following example removes the `Execute` command from the list:

```
resident execute delete
```

The UseCount value of a system segment is set at -1. Because an entry can only be removed when UseCount is at one, a segment cannot be erased using remove.

**ADD** The `ADD` argument makes it possible to make more commands or segments resident with the same name. It can only call the last entered command from the `Shell`.

**REPLACE** The `REPLACE` argument replaces any command (or segment) with a command (or segment) already in the list. For example, if the `Execute` command was resident, entering the following would replace it with the `Date` command:

```
resident execute date replace
```

## 4.1.7 SetPatch

*Syntax:*

```
setpatch
```

This command is found in the startup sequence on the new Workbench disk. It modifies the Kernal so that a Guru Meditation does not follow a Recoverable Alert. `SetPatch` is a background process started by using `Run`. It can be created with the help of the `Status` command.

## 4.1.8 Setenv/Getenv

*Syntax:*

```
setenv NAME/A,String
getenv NAME/A
```

These commands are not currently implemented. When they are installed, you'll be able to make use of environment variables (the environment handler is still missing). For now the handler can be simulated by the RAM disk, but full use is not yet realized.

## 4.1.9 IconX

This command allows you to call a script file from the Workbench by double clicking on it. 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 `CLI` 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 `Info` menu item from the `Workbench` menu. The `SYS:C:IconX` command must be entered in the `Default Tool` gadget. Save the `Info` 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 Type` string gadget in the `Info` 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.

## 4.2 Workbench 1.3 Devices

The Mount command now allows you to connect new devices in AmigaDOS. This section shows these devices, suggests what to look for in them and how you can interact with them.

### 4.2.1 The NewCon device (newcon)

The Shell uses this new window interface for output and input. The newcon: device is similar to the old con: device from the CLI. Before it can be used it must be mounted, like all other devices, using the Mount command:

```
mount NewCon:
```

The important entry in the MountList file found in the devs drawer on the Workbench disk should look like the following:

```
NewCon:    Handler = L:Newcon Handler
           Priority = 5
           StackSize = 1000
#
```

### 4.2.2 The RAD device (rad)

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 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 befoe you can mount RAD with mount rad:. Each cylinder has a capaciy of 11K. RAD would have a capacity of (21+1) * 11K = 242K if it wee connected using the above entry. RAD must be formatted before it cn be used with the FastFile system.

After a reset, all you have to do is enter mount rad: and the contens of RAD are restored. If you discover that some data is lost, use te DiskDoctor to restore RAD.

When RAD is no longer needed, the largest section of its memory cn be freed by using the Remrad command. It can be removed by usirg the Assign command:

```
assign rad: unmount
```

The entire memory area that was occupied by RAD is then free after te next boot operation.

### 4.2.3 The Pipe device (pipe)

This device opens a communication channel for data exchange betwen different tasks. This communication channel consists of a 4K daa 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 usirg 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 addd to it. In the following example the actual contents of the directory ae 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.

## 4.2.4 The Speak device (speak)

This device controls the Amiga's **speech output. The speak:** device must be mounted using the `Mount` **command:**

```
mount speak:
```

Example for speech output:

```
echo > speak: "nice to see you"
dir > speak df0: opt a
type s:startup-sequence to speak:
```

It is theoretically possible to choose different output modes using `Opt`. The present `speak:` handler does not evaluate the options correctly.

## 4.2.5 The Aux device (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:

```
mount aux:
```

Multi-user operation can be realized with the Amiga (see Chapter 5) through this device. The transfer parameters are set from the `Preferences` program.

## 4.2.6 The `FastFileSystem`

You'll find this new handler program 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 somewhat 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 dh1: 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.

# 5. CLI Tricks and Tips

# 5. CLI 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 the CLI. 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 the CLI.

## 5.1 Input and Output in the CLI

You have probably wished that you could just press a key in the CLI 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 print-out 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

Old Commodore 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., PET, VC20, C64). If the user wanted to erase the files named `test` (`test1`, `test2`, `test3`, etc.) from a disk, the `Scratch` 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., `Scratch` "`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 in at the beginning of a filename sometimes means nothing more than a word or number. A numerical value is also expected behind it sometimes. 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 become 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 should be to determine which of the filenames would be found given the `Dir` commands with the respective parameters.

***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 the CLI

***Keyboard breaks***

The Amiga doesn't have a <Run/Stop> key like the C64. But it's possible to stop the execution of a `CLI` command by pressing <Ctrl><C>. All `CLI` commands react by returning to the program from which they were called (in most cases, the `CLI` 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 come <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 `CLI` 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 `CLI` command within a script file. The `CLI` command 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 `CLI` command of the script file finishes its work and then execution of the text file is stopped. This can cause problems when the `CLI` 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 CLI 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 [CLI2]), the main directory of the disk is displayed in the CLI window. The command doesn't react any more to <Ctrl><C>. The output goes on undisturbed.

There is a way to exit a non-interactive process. The CLI 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 CLI than the one that is presently active. The number of the process that corresponds to the CLI number must simply be entered.

## 5.4 The RAM Disk and the CLI

This section is for the user who has only one disk drive. Difficulties frequently arise out from 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.

***CLI commands in the RAM disk***

The first CLI frustration occurs when the Workbench disk is removed and any other disk is inserted. Then the main directory is searched for by means of the Dir command. A requester appears in the upper left hand corner with the message Please insert the disk Workbench in any drive (or something similar). If you click on the Cancel gadget, the Amiga tells you that no Dir command is available. The Workbench disk causes this problem. If you enter the more specific dir df0:, the same thing happens.

As we mentioned in Chapter 1, every CLI command is nothing more than a short program that must be loaded from the disk before it can be used. In normal cases these commands are found on the disk that was used to start the system. On the Workbench disk the commands are stored in drawer c. The operating system knows exactly which disk is the system disk at power-up. It can be addressed under the name Sys:.

The above example has an error in it. The Dir command wasn't given with the correct name of the disk on which the desired directory is kept. **df0:** is just the label of the drive. DOS finds the Dir command on the Workbench disk and displays the directory of the disk that is in drive 0.

If you enter a specific disk name (e.g., Dir Games:), DOS loads the Dir command from the Workbench disk, asks for a disk with the name Games, and then displays the directory of that disk.

Insert the Workbench disk and enter the following three lines:

```
makedir ram:c
copy from df0:c/dir to ram:c
path ram:c add
```

Now you can insert any disk and read it by using Dir df0:. The Dir command is quickly loaded from the RAM disk. The individual commands have the following functions:

*Line 1:* Creates a directory in the RAM disk (c)

*Line 2:* The CLI copies the Dir command from the current disk to this directory

*Line 3:* DOS looks in the `c` directory of the RAM disk for the `CLI` commands before it looks on the disk that contains the `CLI` commands

The `Assign` command is an alternative to the `Path` command. `Assign` lets you assign a completely new pathname so that you can search for `CLI` commands. Look at the list given when the `Assign` command is invoked: Under the `Directories` heading you'll find the C entry. To the right of this entry a pathname is given (e.g., `A500 WB1.2 D:c`). AmigaDOS looks for the `CLI` commands in the directory given there. When starting up the computer AmigaDOS automatically looks in directory `c` on the system disk. This can be changed very easily using the `Assign` command:

```
makedir RAM:c
assign C: RAM:c
```

Now the Amiga understands only the commands located in the `c` directory of the RAM disk. Immediately following the `MakeDir` command, the important DOS commands should be copied into this directory. If you want to access the commands on the original disk, the entire pathname must be entered:

```
sys:c/assign C: sys:c
```

This disables DOS access to the RAM disk. `Sys:` addresses the system disk. If this disk isn't in the drive, a requester asks for it.

The following lines copy the basic `CLI` commands:

```
;ram_cli
makedir          ram:c
copy c/assign    ram:c
copy c/dir       ram:c
copy c/cd        ram:c
copy c/copy      ram:c
copy c/delete    ram:c
copy c/makedir   ram:c
copy c/rename    ram:c
copy c/newcli    ram:c
copy c/endcli    ram:c
copy c/run       ram:c
copy c/list      ram:c
assign C: ram:c
```

Other `CLI` commands can be copied into the RAM disk if you wish. Limit yourself to only the most important commands, because some commands like `SetClock` or `Date` only take up memory. Users that have only 500K of memory can run into problems with this.

If a command is important and it isn't on the RAM disk, using either `Assign` (`assign C: sys:c`) or the complete pathname of the missing command (`sys:c/date 29-oct-87 11:30:35`) allows it to be loaded from the system disk.

If entering the command list at every session takes too long, you can perform the same process with a script file (see Chapter 6 for details on script files).

If you need further information about the `MakeDir`, `Copy`, `Path` and `Assign` commands, see Chapter 2.

## *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 already been copied to the RAM disk and the system was informed using `Assign` or `Path`. The two commands must be present 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 command for this is `Delete`:

```
delete ram:c all
```

This erases the entire `C` directory that was placed in the RAM disk.

## 5.5 Printing from the CLI

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 the CLI, 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 program. 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 use through Preferences. A program that sends data to the printer needs no more provisions than 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 gotten around 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 your wanted ):

```
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:

```
type >prt: df0:s/startup-sequence
```

### 5.5.3 Printer control characters

The printer drivers in the Preferences program have a few things missing. You cannot 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*). The CLI 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 a CLI 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 the CLI. 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:

The CLI 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 CLI 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 the CLI. The following section should give a few experiments with this device.

Basically, all output created from CLI 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 the CLI, you don't have to open a new window using the NewCLI 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 the CLI than turning a connected printer into a typewriter. The following command is all it takes:

```
copy * to prt:
```

The asterisk 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 the CLI 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 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 appears. 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
```

***A CLI alarm clock***

You may find yourself losing track of time during these CLI 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 clock:

```
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. You should remember that this CLI 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).

Because the CLI 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), the CLI 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 the CLI:

```
type con:300/10/150/50/Converter to * 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 ASCII codes. The input must always be 16 characters. For example, if you want to find the ASCII code for the letters S and J, enter the two characters and press the <Return> key 14 times. The result appears in the `CLI` window. The letters have the 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 also known as a 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 interfaced to the central unit with the correct connection (e.g., Model 100, C64, Atari ST, etc.). Because the 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 an Amiga 500 as the terminal, and an Amiga 2000 with a hard drive and two floppy disk drives as the central unit. We hard soldered the connecting cable ourselves. 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.

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 the CLI we entered and saved the following parameters in both computers with the help of the Preferences program:

| | |
|---|---|
| Baud rate | 9600 |
| Buffer Size | 512 |
| Read Bits | 8 |
| Write Bits | 8 |
| Stop Bits | 2 |
| Parity | None |
| Handshaking | RTS/CTS |

We tried the terminal program that came with the Extras disk first. Both programs told us the modem wasn't ready, and that a Carrier Detect signal error occurred on pin 8. This signal is always active in the telecommunication system if the transfer signal is received from the receiving computer.

We then entered the following command on the central unit (Amiga 2000):

```
run copy ser: to con:10/10/300/100/Receive
```

All data that was received should have gone in the window that was specified. Then we sent the current main directory of the Amiga 500 through the serial connection:

```
dir > ser:
```

Result: It didn't work at first. After the first output from the disk directory the first file finally went to the Amiga 2000. The rest remained in the receiving buffer until we pushed new data in behind it. This is a program error, since no end of file marker was sent. This bug isn't easy to remove.

The transfer took place in principle, and we have the multi-user system to thank for that. There's a simple solution for the central unit: The NewCLI command can enlarge a con window. On one hand the command is received in this window, and on the other hand the CLI command writes its output in this window. The following trick is so simple: Direct the NewCLI command through the serial device as follows:

```
run newcli ser:
```

If the the connection works like it normally is expected to, the 2000 can serve a terminal over the RS-232 interface.

It isn't easy to simulate a true terminal from the CLI. Here we simply sent a small script file from the 500 to the 2000. Because the file was long enough, the first commands from the CLI background process of the 2000 worked. Suddenly the hard drive ran because Dir jh0: was in the program. In addition, the Status command announced its

presence. In any case, the first step in working with the multi-user system is taken. Knowledgeable C programmers will have no problem taming the connection and writing a terminal program.

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 call up from a second computer through the ROM-Wack (a kind of diagnosis program), in case a Task Held or Guru Meditation occurs.

A simple terminal program that is good enough for now is the program found on the Extras disk in the BasicDemos drawer. If the computer hangs up and the right mouse button is pressed (press the right button to cancel/debug), the memory of the computer can be examined with a terminal program from 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:

Computer A and computer B are connected by an RS-232 cable. The following lines create a Task Held condition in computer B:

```
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 from the terminal computer:

```
rom-wack
PC  FE66E8 SR: 0015 USP· 022B32 SSP: 07FFF2 XCPT: 0003 TASK  0221A0
DR  0000FFFF 0000FFFF 0001C815 00000001 00000000 00022394 000221FC 0000001
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 manipulate ROM-Wack on the interrupted computer. Pressing a question mark on the terminal computer 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 jumps into 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 have some form of script file processing capability. AmigaDOS is no exception. Script files are similar to batch file on MS-DOS computers.

This chapter shows you what can be done with this technique andhow 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, CLI 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 CLI commands can only be entered in direct mode; the CLI 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. In addition, there is no program branching allowed—CLI commands can only be executed one after another.

Script files are text files that can be created in a word processing program or text editor. These files consist of a succession of CLI 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 CLI commands, like the ones that you enter in the CLI or 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" CLI commands allowed in script files, a list of special commands exist, whose use in CLI windows wouldn't really make sense. They are the commands:

```
echo
failat
quit
if/else/end
skip/lab
ask
wait
```

If the commands If, Then and Else 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 the CLI 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 execute df0:myscript
```

The operating system separates Execute command sequences so that while the script file is running, further work can be done in the CLI. Eventually, output appears in the CLI 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 CLI 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 batch commands on the screen. The file should contain the names of the commands.

***Solution:*** You must open a CLI 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/then/else"
echo "skip/lap"
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/then/else
skip/lap
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 a CLI window and place the Workbench disk in drive df0:. Enter the Dir df0:S command. A file named startup-sequence is found there. Before this file is displayed it is a good idea to make the CLI window as large as possible. Now enter:

```
Type df0:S/Startup-sequence
```

command 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 halt the scrolling of the screen; release the right mouse button to continue the scrolling. The Workbech 1.2 Amiga startup sequence appears on the screen (An Amiga 2000 startup sequence is given here; the Amiga 500 startup sequence is shorter.):

```
echo "A500/A2000 Workbench disk.  Release 1.2 version
33.61*N"
echo "sample Startup-Sequence for use with a Hard Disk
with Workbench installed"
Sys:System/FastMemFirst
BindDrivers
addbuffers df0: 20
IF EXISTS sys:system
    path sys:system add
ENDIF
IF EXISTS sys:utilities
    path sys:utilities add
ENDIF
SetMap usa1
makedir ram:t
path ram: add
failat 25
assign >NIL: dh0:
IF FAIL
```

```
echo "Transfering control to DH0:*N"
path reset
assign SYS: dh0:
if exists dh0:c
assign C: SYS:c
endif
if exists dh0:l
assign L: SYS:l
endif
if exists dh0:fonts
assign FONTS: SYS:fonts
endif
if exists dh0:s
assign S: SYS:s
endif
if exists dh0:devs
assign DEVS: SYS:devs
endif
if exists dh0:libs
assign LIBS: SYS:libs
endif
if exists dh0:t
assign T: SYS:t
endif
if exists sys:system
path sys:system add
endif
if exists sys:utilities
path sys:utilities add
endif
path ram: add
cd dh0:
ENDIF
LoadWB
SetClock >NIL: Opt load
endcli > nil:
```

If you take a good look at this file, you may notice that only CLI 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 up (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.

The first few lines echo a message to the screen and then check for certain directories and sets the path to these directories. A row of Assign commands framed in If/EndIF constructs follow. These constructs search for the c, s, t, l, libs, devs, fonts, system and Utilities directories. Make sure these files exist by entering the following in the CLI:

```
dir df0:
```

We can remove the IFs, EndIfs and the .info files associated with these directories. The following lines remain in this section of the sequence:

```
echo "A500/A2000 Workbench disk"
Sys:System/FastMemFirst
BindDrivers
addbuffers df0: 20
    path sys:system add
    path sys:utilities add
SetMap usa1
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
LoadWB
endcli > nil:
```

You can decide whether the MakeDir RAM:t line should be deleted or not. If you delete it, no disk icon of the RAM disk appears after the startup sequence is done working. If you leave the line in, then the disk icon of the RAM disk appears on the screen. We decided against the disk icon and deleted this line from the startup sequence. The following Path RAM: add command is also useless to us. We deleted it. Do whatever you did to Dir RAM:.

Is it important for you to know how late it is when your Amiga is turned on? No? Then delete the SetClock opt Load line from the new startup sequence. If you want the clock, leave the line unchanged.

The next line, LoadWb, loads the actual user interface (the Workbench).

The EndCLI > NIL: command closes the CLI—leave it in your new startup sequence.

All of the important commands from the old sequence are contained in this new one. The directories mentioned in the sequence must be on the Workbench disk. If this is not the case, there will be trouble during the startup.

What happens when the new sequence is used instead of the old one? We tried this, and compared the results of the two. The new sequence was 50% faster than the old one (it saved half of the startup time). You can probably profit from this saving of time.

If you have an Amiga 500 and examine its startup sequence you can see that the Assign command is not used.

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 CLI:

```
ed df0:S/Startup-sequence
```

The drive runs for a short time. The editor starts and the old 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 the startup-sequence.

## 6.2.1 The custom startup-sequence

In the previous section you discovered that it is possible to change the startup-sequence. The following shows you how to make the startup-sequence 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 500 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 we hope that you know how this command works and how it is used.

Look on your Workbench disk in directory c to see if you have the command. Different versions of the Workbench don't have this command. You should talk to your computer dealer or a friend if you don't have it to see if they have it on their Workbench disk. Once you have found someone with this command, copy it into the c directory. In the course of this book, you will need the Ask command.

Now back to the startup sequence. Have you ever been annoyed that you had to click on an icon after startup before you could have a CLI 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 a CLI window ? (y/n) "
if warn
 ask "A large (y) or small (n) CLI window?"
  if warn
   newcli "con:0/0/550/200/Startup CLI"
  else
    newcli "con:0/0/160/30/Startup CLI"
  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.

The startup-sequence re-executes. Before it is completely done, a question is asked:

```
Would you like to open a CLI 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, a CLI window with the name Startup CLI opens. <Y> opens a large window. Pressing <N> opens a very small window. This window should really be held in reserve; use it only if you have to.

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 the CLI, 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.

We would like to give you more ideas about this subject in Section 6.3. There you'll find a script file that can be integrated very easily into the startup-sequence.

## 6.2.2 Workbench 1.3 startup-sequences

As the title suggests, Workbench 1.3 handles multiple sequences. We'll discuss the file named startup-sequence first. It has the same function as Version 1.2, but it looks somewhat different. Your startup-sequence may vary since the Amiga system is always expanding and improving.

```
c:run >NIL: c:SetPatch   ;patch system
cd c:
echo "A500/A2000 Workbench disk.   1.3 version *N"
Sys:System/FastMemFirst ; move c00000 memory to last
BindDrivers
AddBuffers df0: 10
FF >NIL: -0              ; start FastFonts
MakeDir Ram:T            ; set up logical T: dir for Execute
Assign T: ram:T
MakeDir Ram:Env          ; set up ENV: directory
Assign ENV: ram:Env
path sys:utilities sys:system ram: sys: s: add
Sys:System/SetMap usa1
LoadWb
SetClock >NIL: load    ;load sys time from real time clock
resident CLI L:Shell-Seg SYSTEM pure ; activate Shell
resident c:Execute pure    ; load the Execute command
mount newcon:              ; mount improved console handler
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 never be entered in the CLI or Shell. Now we come to the second line. It should also be retained. The Echo command takes little time and can be removed. The next line organizes the memory area and should stay. The

BindDrivers command should be eliminated if you don't have any memory expansion in your computer.

The value of the AddBuffers command could be changed from 10 to another value, but don't do it. To obtain faster text output, the FF command must be used. Next, the t directory is placed in the RAM disk. This isn't absolutely necessary. This means that both commands can be removed. That also goes for the next two lines.

The lines containing Env: should be left alone. 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. Whether or not you keep the next line depends on if you have a battery-powered realtime clock in your computer. 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:. This line should be kept. The last line closes the CLI window. Whatever you want to add is up to you. Remember that every line you add increases the duration of the startup sequence. There are two startup files in the s directory. One is called the CLI-startup and the other is the Shell-Startup file. One is for the CLI and the other is for the Shell. The respective file is executed every time a CLI or Shell window is called. In the CLI this is only the command Prompt "%N". In the Shell the following command is used:

```
c:Prompt "%N.%S> "
alias xcopy copy [] clone
alias endshell endcli
```

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 must 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 CLI only with the Shell. Instead, all of the CLI commands can be used.

## 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. First we have written some scripts that work with Workbench 1.2. These also work with Version 1.3. In addition are some scripts that work only with Workbench 1.3.

This book has an optional 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 input goes directly to the printer. You only have to enter the desired command sequence on the keyboard. Since it is difficult to place these escape sequences generated from the keyboard inside a script file, so they can be sent to the printer, you must send the data in a round-about fashion.

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 type style. Enter the following line in the `CLI`:

```
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 type "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). Consult your Amiga User Handbook for the printer escape codes needed for each option.

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 thing you transmit. This resets all styles to normal.

The following script file uses the four script files `NLQ`, `Bold`, `Italics` and `Reset`. 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`. 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`.

```
.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 optional disk, 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 three 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 disk 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 `CLI` 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
    newcli "con:0/0/319/59/A_CLI"
   lab 5
    newcli "con:320/0/319/59/A_CLI"
   lab 4
    newcli "con:0/60/319/59/A_CLI"
   lab 3
    newcli "cc-:320/60/319/59/A_CLI"
   lab 2
    newcli "con:0/120/319/59/A_CLI"
   lab 1
    newcli "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 talk about the last file, 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 computer may generate a `t` directory in the RAM disk on its own.

The last subject that we'll discuss in this chapter is the RAM disk. Here's a script file that copies a few commands into the RAM disk. The commands are copied using shorter names. Here is the script file:

```
.key Parameter
;RAMon
if "<Parameter>" eq "??"
     echo "*n*nCall: execute RAMon*n"
     echo "You don't need to enter any parameters.*n"
     quit
endif
if not "<Parameter>" eq ""
     echo "*nYou must not enter any parameters.*n"
endif
if not exists RAM:d
     makedir RAM:d
     copy sys:c/copy RAM:d/c
     copy sys:c/path RAM:d/p
     assign c: RAM:d
     p add sys:c
else
     copy sys:c/copy RAM:d/c
     c sys:c/path RAM:d/p
endif
c sys:c:dir RAM:d/d
c sys:c:execute RAM:d/ex
c sys:c:delete RAM:d/del
c sys:c:type RAM:d/t
c sys:c:rename RAM:d/r
c sys:c:echo RAM:d/e
```

```
e "Abbreviated commands are now available."
```

Enter these lines in an editor and save them under the name `RAMon`. After you have started the file with `Execute RAMon`, the shortened commands `C`, `P`, `D`, `EX`, `Del`, `T`, `R` and `E` are ready to be used. The commands that these letters represent are found in the lines above.

It is possible to abbreviate more commands. You must be careful that you don't use the same abbreviation twice.

The assigning of the abbreviated commands can also be used in the startup sequence. The lines should appear at the end of the sequence.

To free up more memory and stop using the shortened commands, we have constructed the following script file:

```
.key Parameter
;RAMoff
if "<Parameter>" eq "??"
     echo "*n*nCall: execute RAMoff*n"
     echo "You don't need to enter any parameters.*n"
     quit
endif
if not "<Parameter>" eq ""
     echo "*nYou must not enter any parameters.*n"
endif
if exists RAM:t
     delete RAM:t all quiet
endif
cd ram:d
delete #? quiet
cd df0:
echo "Abbreviated commands are no longer available."
endif
```

This program should be saved under the name `RAMoff`. By entering `Execute RAMoff` the shortened commands are erased and so is subdirectory `t` in the RAM disk. This frees up more memory to be used for other purposes.

You will discover that the `D` directory in the RAM disk isn't erased. The Amiga would give us a message saying that this directory cannot be erased if we tried to delete it. The `RAMon` script file accesses the `d` directory in the RAM disk during execution. The directory must be present.

This is a small price to pay for abbreviated commands.

## 6.3.3 Working Workbench 1.3

This chapter uses the commands of Workbench 1.3. You can try to create the script files with the 1.2 commands since all of the commands are present on Workbench 1.2. But that doesn't mean that it will work. Most 1.3 commands offer more options than 1.2 commands.

The script file in this section creates a working copy of Workbench 1.3. We'll explain shortly what we mean by a working Workbench.

When you sit at your computer, you generally know what you want to do. For this reason you only use a few of the commands or programs on the Workbench. Sometimes you need an additional program, such as a disk monitor or another editor. We recommend you prepare multiple Workbenches that have more or less the same structure, but contain the additional programs. For example, you could have a word processor as the main program on one disk and a database on another disk, etc. The script file below prepares a work disk that has all of the basic Workbench programs needed to run applications.

Before you start the file, make a copy of the original Workbench 1.3. Enter all of your `Preferences` and save them on the copy. This is very important because later it will be impossible to change the disk. The `startup-sequence` and the `MountList` should also be in order before you start with the script file. Place the original Workbench 1.3 in a safe place. The following is the script file, do not enter the line numbers, they are only for reference.

```
 1  ;Workdisk
 2  sys:c/makedir ram:Workdisk
 3  sys:c/makedir ram:Workdisk/c
 4  sys:c/copy sys:c/Copy ram:Workdisk/c
 5  sys:c/cd ram:Workdisk/c
 6  copy sys:c/Addbuffers ram:Workdisk/c
 7  copy sys:c/Ask ram:Workdisk/c
 8  copy sys:c/Assign ram:Workdisk/c
 9  copy sys:c/Binddrivers ram:Workdisk/c
10  copy sys:c/CD ram:Workdisk/c
11  copy sys:c/Delete ram:Workdisk/c
12  copy sys:c/Dir ram:Workdisk/c
13  copy sys:c/Echo ram:Workdisk/c
14  copy sys:c/Endcli ram:Workdisk/c
15  copy sys:c/Execute ram:Workdisk/c
16  copy sys:c/FF ram:Workdisk/c
17  copy sys:c/If ram:Workdisk/c
18  copy sys:c/Install ram:Workdisk/c
19  copy sys:c/Lab ram:Workdisk/c
20  copy sys:c/Loadwb ram:Workdisk/c
21  copy sys:c/Makedir ram:Workdisk/c
22  copy sys:c/Mount ram:Workdisk/c
```

```
23  copy sys:c/Newshell ram:Workdisk/c
24  copy sys:c/Path ram:Workdisk/c
25  copy sys:c/Prompt ram:Workdisk/c
26  copy sys:c/Resident ram:Workdisk/c
27  copy sys:c/Run ram:Workdisk/c
28  copy sys:c/Setpatch ram:Workdisk/c
29  copy sys:c/Setclock ram:Workdisk/c
30  copy sys:c/Skip ram:Workdisk/c
31  copy sys:c/Type ram:Workdisk/c
32  copy sys:c/Iconx ram:Workdisk/c
33  ;Additional commands may be placed here
34    copy sys:c/date ram:WOrkdisk/c
35    copy sys:c/failat ram:Workdisk/c
36    copy sys:c/wait ram:Workdisk/c
37    copy sys:c/list ram:Workdisk/c
38    copy sys:c/break ram:workdisk/c
39  copy sys:.info ram:Workdisk
40  copy sys:Clock#? ram:Workdisk
41  copy sys:Disk.info ram:Workdisk
42  copy sys:Shell#? ram:Workdisk
43  copy sys:System.info ram:Workdisk
44  makedir ram:Workdisk/System
45  copy sys:System ram:Workdisk/System all
46  makedir ram:Workdisk/L
47  copy sys:L ram:Workdisk/L all
48  makedir ram:Workdisk/Devs
49  copy sys:Devs/#? ram:Workdisk/Devs
50  makedir ram:Workdisk/Devs/Keymaps
51  ;optional copy sys:Devs/Keymaps/other_keymap
ram:Workdisk/Devs/Keymaps
52  copy sys:Devs/Keymaps/usa1
ram:Workdisk/Devs/Keymaps
53  makedir ram:Workdisk/Devs/Printers
54  copy sys:Devs/Printers/generic
ram:Workdisk/Devs/Printers ;add your own
55  makedir ram:Workdisk/S
56  copy sys:S ram:Workdisk/S all
57  makedir ram:Workdisk/T
58  makedir ram:Workdisk/Fonts
59  copy sys:Fonts/Topaz.font ram:Workdisk/Fonts
60  makedir ram:Workdisk/Fonts/Topaz
61  copy sys:Fonts/Topaz/11 ram:Workdisk/Fonts/Topaz
62  makedir ram:Workdisk/Libs
63  copy sys:Libs ram:Workdisk/Libs all
64  makedir ram:Workdisk/Utilities
65  echo "*n*nPlease insert a disk in drive df0:."
66  echo "This disk will be formatted and become your
working"
67  echo "copy of workbench 1.3.*n"
68  wait 30 ; wait 30 second to change disks
69  ram:Workdisk/System/format drive df0: name
"Workdisk_workbench_1.3" noicons
70  install df0:
71  copy ram:Workdisk df0: all
72  cd "Workdisk_workbench_1.3:"
73  delete ram:Workdisk all quiet
```

```
74  echo "Ready!"
```

This script file is different in one way from the files we have mentioned previously. The other script files required parameters to be entered. This script file will be executred using the ICONX program. The `Workdisk` file in the `Script_files` directory on the optional disk available for this book can be activated with the mouse with the help of the `IconX` command of Workbench 1.3. Using the `IconX` command has the disadvantage that only the commands in the script file can be used. You probably know how the lines in this file work, but here is an explanation of them.

| | |
|---|---|
| ***Lines 1-3*** | Creates a `Workdisk` directory and `c` directory in the RAM disk. |
| ***Lines 4-5*** | The `Copy` command is copied into the `C` directory and the `ram:c` directory is made the active directory. This has the advantage that the following commands do not have to be loaded from disk. |
| ***Lines 6-32*** | Each line copies a command from the `c` directory of the Workbench into the `c` directory of the RAM disk. Why so many, you may ask. The large number of commands is necessary because they are all used either in the `startup-sequence` or inside the script file. Some commands are used so often that they are not allowed to be missing from the list. These are, for example, the `Dir` or `Type` commands. |
| ***Lines 33-38*** | Place any more commands required by your `startup-sequence` after this line. |
| ***Lines 39-42*** | Here all required files are copied from the Workbench disk to the `Workdisk` directory of the RAM disk. |
| ***Lines 43-47*** | The `System` and `l` directories are provided in the `Workdisk` directory. Then the contents (files) of the directory of the same name on the Workbench are copied there. |
| ***Lines 48-49*** | All of the files (not the subdirectories) from the `devs` directory on the Workbench disk are copied into the RAM disk. This directory contains a file with the name `System-configuration`. All information that was entered in `Preferences` is stored here. The `Preferences` of the computer cannot be changed because the `Preferences` program was not copied. Another file has the name `MountList`. This should be modified to your liking before using this script file. |
| ***Line 50-52*** | The `Keymaps` directory is prepared by the `devs` directory. Then the file `usa1` (for the American keyboard setup) is copied into the RAM disk from the Workbench. There is also an optional line for adding any other keymap files you might want. |

| | |
|---|---|
| ***Line 53-54*** | Functions like lines 44-46 except that the `Printers` directory and the `generic` printer driver are copied. You must copy the printer driver you have specified in `Preferences`. The `generic` file is used here as an example. If you want other printer drivers, use the `Install-Printer` program found on the Extras 1.3 disk. |
| ***Lines 55-56*** | Directory `s` is created on the RAM disk. The `startup-sequence` and the `CLI` and `Shell-Startup` files are copied into this directory. |
| ***Line 57*** | A `t` directory is prepared for the temporary files. |
| ***Lines 58-61*** | These lines show how a font is copied from the Workbench. Should you want to use more fonts, add more lines like the one we show. |
| ***Lines 62-63*** | Prepares the `libs` directory. Then the contents of the `libs` directory is copied into the RAM disk. |
| ***Line 64*** | We have created this directory because a path to this directory is placed in the `startup-sequence`. We don't copy any files into this directory. The `calculator`, `Notepad` or other programs can be copied into this directory if you use them often. |
| ***Lines 65-68*** | Text is displayed on the screen and the Amiga waits 30 seconds for you to insert the correct disk. |
| ***Lines 69-70*** | After you remove the Workbench from drive **df0:** and replace it with a blank disk, the disk is formatted and then installed (made bootable). |
| ***Line 71*** | This line copies the entire `Workdisk` directory of the RAM disk, along with all of the subdirectories and files, to the new disk **Working_Workbench_1.3:**. |
| ***Lines 72-73*** | The directory changes to the **Working_Workbench_1.3: disk** and the RAM disk could be deleted here (**don't forget the colon at the end of** the device name). |
| ***Line 74*** | An ending message appears on the screen. |

The file needs only one drive to run. Because the memory needed to run is very large, you should have at least a megabyte of RAM. When this is not possible, don't copy as many commands from the `c` directory. You can leave off all of the commands not used inside the script file. These commands must be copied after the file is through running.

To use this script file, boot your Amiga with a copy of Workbench 1.3 that is write protected. Then start the program as described above.

When you have made a work disk with the script file, you can copy the programs that you need onto the disk. Be careful that these programs do not need other files that are not present on the disk. These files must be copied onto your **Working_Workbench_1.3:** disk in the correct directory.

## 6.3.4 Starting the script file 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 the correct drawer, then open a CLI window. A script file can be started with a mouse click. Workbench 1.3 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 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 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. Select the Info item from the Workbench menu. An Info window appears. The word TYPE appears in the upper left hand corner. Right next to it is the word Project. When Project or TOOL appears next to the word TYPE, this icon can be used for our purposes.

Open the Shell again. Enter the following to copy the Shell icon information to the Batch program 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 Info item from the Workbench menu. This displays the Info 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 in the above case 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. 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. IconXable script files can only contain commands that can be entered directly in the CLI or Shell. 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.

## 6.3.5 The Types script file

Hopefully you saved your work in the s subdirectory on the Workbench if you've created script files before working through this book. That is where the Amiga looks for a script file if it is not found in the main 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 optional disk available for this book, for the optional 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"`.

## 6.3.6 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). 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. This is because the reset-resitant RAM disk is always present and may be remounted using `mount rad:` after a warm start. The Workbench files don't need to be copied again. The additional lines take care of this.

## 6.4 Using Alias with NewShell Commands

Workbench 1.3 has a very efficient program called the Shell. The Shell has many advantages over the 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 CLI 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 optional 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. 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 are 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 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"
   ```

You see the changes affect more than the output. The `Alias` command can bring about 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. Every `Shell` window has a border and a title line. How does it look when no title and border appear. Type the following line if you would like to know. Then the `noborder` command is available for use.

   ```
   Alias noborder echo "*e[80u*e[0x*e[0y*e[31t"
   ```

   After you enter this line, size the window at its full size. Enter the `noorder` command. The window is hardly changed. Pressing <Ctrl><L> and <Return> achieves the desired result: The border is gone. To bring back the border, press the <Esc> key (you won't see the input in the `Shell` window) and then the <C> key. Resize the window to restore the border.

5. 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
   ```

6. 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 Kickstart 1.3 to use this command. This h bit is ignored when Kickstart 1.2 is used.

7. Like in 6, 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.

8. 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`.

9. Section 6.3.2 described a script file that copies some `CLI` commands into the RAM disk and assigns them abbreviated command words. The `CLI/Shell` 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 *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. 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 the `CLI` 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 the `CLI` 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 demo 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 the `CLI` and Workbench at the same time.

Before we show you the many possibilities for multitasking with the Workbench and the `CLI`, we must mention that the Workbench doesn't really have "multitasking capabilities." To see this, format a disk with `Initialize` and try to do something else while the disk is being formatted. It doesn't work. The mouse pointer turns into the wait pointer and doesn't let you do anything else. Also, the Workbench can start programs one after another, but it cannot execute more tasks through each other. This is why the `CLI` exists, so you can execute several tasks at the same time.

It is important to have the correct commands in RAM if you are using an Amiga 500 with one disk drive. What happens when `Intuition` loads a program from the disk and AmigaDOS looks for the desired command from the same disk can only be endured with patience. If you have two disk drives, the system disk with the `CLI` commands should be in one drive and the program should be started from the Workbench in the other drive. Copying the `CLI` commands into RAM using script files was covered in Chapter 6.

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 the `CLI` with:

```
endcli >nil:
```

The `CLI` 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 on with your work on the Amiga. How can you work with the Workbench and `CLI` at the same time?

***An example:***

Pretend 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 the CLI. You can get out of this by resetting the computer, but then you lose the data in the RAM disk. Had you placed a CLI 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. The CLI 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 a CLI window. Remember a tiny CLI window in the background doesn't use that much memory and can save a lot of frustration.

Another use for the combination of the Workbench and the CLI 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 disk. You could make the BeckerText window very small and place all data files to his disk using the Workbench. It is simpler to click on the CLI window and use the Copy command to give your friend a copy of your files. And if his 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 the CLI. 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 the newest game.

***Deleting files***

This can go for 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 any more. You could use the Workbench to open the desired drawer, mark all the text and then erase it with Discard. While the files are being erased you cannot work with the Workbench. It is much easier and faster to click on the CLI 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 CLI window might get in your way when you work with the Workbench and CLI. 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 CLI window. Open a new CLI window with the desired dimensions and close the old CLI window. This section of an improved startup sequence can look like this:

```
.....
........
newcli "con:10/10/60/60/MyCLI"
........
endcli >nil:
```

## 7.3 Multitasking with NewCLI

You don't have to use the CLI for everything. Executing important tasks through the CLI 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 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. Start a new CLI with NewCLI. Position the second CLI 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 CLI, change to the second CLI, 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 CLI window and execute the command in another CLI window. When you don't need the other window anymore, you can get rid of it by entering EndCLI.

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 make another CLI. 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 taste 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 CLI open and only have a little memory remaining in the Amiga. It's the same thing with the Workbench: If the main CLI is being used or is hung up by a program, you can continue to work with the second CLI window. If you innocently enter Copy text PRT: when working with one CLI window you have to wait until the text file is printed or else open a second CLI window through the Workbench. It's much better to have a second CLI open and simply be able to continue.

AmigaDOS can affect new CLI windows. Say you have chosen df0:Texts/Private as the current directory, then opened another CLI with NewCLI. New input is automatically sent to the directory of the previous CLI. In our case, the current directory of the second CLI 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 CLI window. A small CLI window can be created in the upper left hand corner with:

```
newcli con:10/10/60/60/MyCLI
```

It is best to enter this long line only once using ED and save it on the disk under the name NC (for NewCLI). Then the new CLI window can be called by entering:

```
execute df0:nc
```

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 CLI to appear is:

```
e nc
```

## 7.4 Multitasking using Run

The possibility of using NewCLI 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 CLI 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 a new CLI is furnished that comes without its own window. Let's see how this works. Place the Workbench disk with the CLI commands in **df0**. Then enter the following two lines:

```
run copy df0:c/run ram:c
dir df0:
```

Immediately after you press the <Return> key a new line appears:

```
[CLI 2]
```

and then the 1> (when more CLI processes have been started, this is a different number). Then you entered the second line (hopefully quick enough that the drive was still working) and the CLI 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:

```
type df0:s/Startup-sequence
run dir df0:
```

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 a CLI command started with Run doesn't have a window to use for its output. It puts all of its output in the CLI window.

In spite of this restriction, there are many uses for multitasking using Run. These include all of the CLI commands that don't generate any screen output and require relatively little time (Format, DiskCopy, Copy). 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 the CLI and would like to continue to work during the time required for the program to load. One option would be to start a new CLI with NewCLI and call the program from the second CLI 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 CLI with:

```
run program
```

A typical example uses the editor. For example, you write a C program and have to enter corrections in the program. If you are using only one CLI, you must leave the editor, call the compiler and then start the editor again. Starting the editor from a new CLI 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 again. This saves a lot of time.

There is another restriction to Run that can have unpleasant results. Remember that 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 CLI commands don't need any more input". But that is not completely true. The CLI commands allow <Ctrl><C> or another <Ctrl> function to be input. When a program or CLI command is started with Run, <Ctrl><C> doesn't go to the command, but instead to the CLI. To put it another way, you have a 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 CLI window with:

```
run type text
```

you can enter:

```
break 2
```

to stop the text output from the second CLI, CLI 2. The complete command is discussed in Chapter 3. It's enough for us to show here how to stop commands started with Run.

To close this section we want to give some of the basic differences between NewCLI and Run. Have you created the new CLI command Task from Chapter 9? In contrast to the Status command, the Task command doesn't only show the existing CLI processes, but also all of the existing tasks in the Amiga. Every NewCLI creates a new task with the name NewCLI, while every task started with Run creates a task called Background CLI. The Task command from Chapter 9 will show you all of the tasks in a short list. This could look like this for example:

```
CON               5
Initial CLI       0
input.device     20
File System      10
Workbench         1
trackdisk.device  5
CON               5
Background CLI    0
```

An entry Background CLI appears in the last line of this example. This command was started with Run and has no input or output window.

```
CON               5
New CLI           0
CON               5
Initial CLI       0
```

```
Workbench         1
File System      10
input.device     20
trackdisk.device  5
RAM               0
CON               5
```

In this example a NewCLI is in the second position. Here a second CLI window was opened. This window allows keyboard input.

## 7.5 Using the CLI

You probably got an impression of how much time and work could be saved by using the full range of AmigaDOS options in Sections 5.2 through 5.4. 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 the CLI to avoid long pauses when copying and formatting disks. In addition, a CLI 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 the CLI, but you can also look in iconless drawers. That is important when you have a disk that doesn't show anything on the Workbench.

As you work more with the CLI, you should plan your use of the CLI 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 CLI window in the background at all times for security.

An especially meaningful use for multitasking with the CLI is correct planning of 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 CLI commands are most frequently used and copy these into the RAM disk. In this case you would have the text to be printed in drive **df0:**, while the other CLI processes call their commands from the RAM disk.

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 machines have it easier. They can put important programs and a generous amount of commands in RAM, 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 CLI) 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, with all of the printer command codes is printed, it can be sent to the printer from the CLI while 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:

```
newcli
```

and you will have a CLI 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 afterwards.

Pretend that 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 is 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 the `CLI` processes. This command sets the priority for `CLI` tasks at a number between -128 and +127. You can see this with:

```
status full
```

and the actual information about the `CLI` processes appears. The display for a `CLI` looks like the following:

```
task 1: stk 1600, 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 `CLI`s 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 1600, gv 150, pri 3 loaded as command: status
```

We see that something has changed, the `pri 3`. Our `CLI` has a priority of 3. Now you may think that you will wait forever to access the `CLI`. 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 `CLI` window with `NewCLI` 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 any more 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, you can use the difference very drastically. Enter `ChangeTaskPri 0` in `CLI 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 `Editor` window. Then move the `CLI` window so that you can input next to the editor. Display the direciory of `df0:c` in this window with:

```
dir df0:c
```

and try to produce the cursor inside the `Editor` window at the same time. 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 the `CLI` 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 with Workbench 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
```

Don't let `pri 157` worry you. This is really be -99. The error lies in the 1.2 `Status` command. It can only show numbers from 0-255 and the priorities have numbers from -128 to +127. So a -99 is shown as +157. This was corrected in Workbench 1.3. Now we want to go through the same test with the changed values. Start the output of the long directory in the `CLI` and try to produce the cursor in the editor.

You'll soon determine that the work with `ED` is not hindered any more. The output of the filenames is stopped completely when a key is pressed in the `Editor` 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 the `CLI` processes that were called from the command can be changed. This `CLI` 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. This was corrected in Workbench 1.3.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.
- Only `CLI` processes can be influenced with `ChangeTaskPri`. It is not possible to change the priority of a program started from the Workbench. In Chapter 9 we describe the new `CLI` command `TaskPri` that can change not only the priority of programs started from the Workbench, but of all tasks: An important help for complete control over the multitasking of the Amiga.

## 7.7 What to Watch For

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 two halves, 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 `CLI` command from the same disk. That happens in the following example:

```
run copy df0:text prt:
cd df0:
```

While `Copy` is reading from the disk to print the `Text`, the `CLI` must read the next command (`CD`) from the disk. In this case the movement of the drive head does not last very long. The use of a second drive is the only way to help here. 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 the `CLI` 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 `CLI` 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 you want to 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 is the cause of the error message `CLI error: Unable to open redirection file.` when the output is directed to `Datafile1` while another `CLI` 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 +5. 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 come into line through these routines. The next two examples will show this, but be sure to save all of your important work before trying these examples.

***Attention:*** When you try out the following two examples, you lose control over the Amiga. Save all of your data beforehand, including the contents of the RAM disk.

1. Set the priority of the `CLI` process to `50` (`ChangeTaskPri 50`). Start AmigaBASIC with `Run` (`Run AmigaBasic`). Click in the left window and write some text in it. Press the <Return> key.

   You'll suddenly notice that no more keyboard input is possible. The mouse can be used if you are lucky. Click in a couple of windows and suddenly the mouse is lost. AmigaBASIC has a higher priority than all other tasks, so working with them is no longer possible. Because AmigaBASIC waits directly for its results, without using the wait routine of the operating system, these results never arrive. The system hangs up even though there is nothing wrong with it.

2. You need Workbench 1.2 and the new `CLI` command `TaskPri` from Chapter 9 for the second example. Start the `Lines` program in the `Demos` drawer. Enter:

   ```
   TaskPri Lines 50
   ```

   Click in the `Lines` window. Now no more keyboard input or mouse movement is possible.

We hope that these two examples illustrate that you can really take charge by changing the priorities. So we advise that you work only with priorities from -5 to +5 so that the important system tasks work as they were meant to work.

# 8. AmigaDOS Internals

# 8. AmigaDOS Internals

## *Global Vector Table*

It's fun to work with the `CLI`, but it's even more interesting to see how the `CLI` and AmigaDOS really function. It's especially useful if you want to write your own programs. You can add some tips and tricks to your own programs that you learned from examining existing `CLI` commands. We'll now discuss the internal functions that we discovered. Much of this information cannot be found in any other book, such as a description of the internal construction of the `CLI` commands and their use of the *Global Vector Table*. These global vectors provide mysterious "gv 150" after a `Status Full` command.

This book doesn't provide complete information about the Amiga—that would require many volumes. We don't cover tasks and their structures, message ports, etc.; we just mention them briefly. We want to give you the most interesting information about AmigaDOS.

To understand and use all of the information in this chapter, you should be familiar with the C programming language and 68000 machine language. We'll try to present the information so that an Amiga user without experience in these programming languages can get at least some idea of what's happening.

## 8.1 DOS, Devices, Handler, Packets

Maybe you've wondered how AmigaDOS really functions. For example, maybe you have pondered how the same file can be sent to a disk, the RAM disk or a printer. In this section we want to explain how this occurs and what the devices and handlers have to do with this.

*DOS and files*

Say you want to open a file for reading. You would use the `Open()` function of the `DOS.library`, which requires two parameters: the filename and type of access (read/write). DOS then looks for the filename and tries to open the file. The following special filename cases should be noted:

- The filename is `*`. In this case DOS makes sure that the input/output goes to the actual `CLI` window. This means that the message port of the Con handler is used for the input/output message port.
- The filename `NIL`. No routine exists for input or output. Nothing is displayed when `Read()` or `Write()` calls are made.
- The filename contains a colon (`:`). AmigaDOS checks to see if the device preceding the colon is in the device list. A disk device (**df0:**,**df1:**, etc.) must also be present in a disk drive. When this is the case, the message port of this device opens for later input and output. The handler for this device is loaded and started if it isn't in memory.

A volume name (e.g., Workbench 1.2) can also precede the colon if this device or volume isn't found. In this case AmigaDOS uses a requester to ask that this volume be placed in any drive.

When an access with `Read()` is made to the opened file, AmigaDOS checks in an internal buffer to see if the characters are there. If no characters are present in the buffer DOS sends a message requesting that the buffer be filled with data. AmigaDOS doesn't access the disk directly, but demands the data from the device. Many other actions of DOS really occur with the device, for example, renaming the disk, renaming files or directories, building a new directory and so forth.

*DOSPackets*

How is DOS notified by a device? For this DOSPackets are used. The include file `dosextens.h` contains the exact method. The `dp_Type` entry is important because the desired action is stored as a number here. The possible actions and their numbers are found in the include file. For example, if DOS wants to rename a disk, then a packet is sent with `dp_Type` = `Rename_disk` (=9) to the device and waits for the Reply

packet, the answer of the device. Then DOS can determine if the renaming took place or if an error was encountered.

*Devices and Handlers*

Up until now we've said that DOS communicates with a device and exchanges information in this manner. This was incorrect in one respect but we have used it temporarily. To be more accurate, DOS works mainly with devices and handlers, and DOS usually works only with the respective handler. Amiga ports are handlers. The `Port` handler is used for the serial and parallel ports, the `Con` handler for keyboard and screen and the `FileSystem` for the disks. A device belongs to every handler and uses the port for connection. The `Port` handler works with three devices: `parallel.device`, `printer.device` and `serial.device`, the `FileSystem` with `trackdisk.device`, and the `Con` handler with the the `console.device`.

Basically a device is closer to hardware level than the respective handler and can only execute much simpler actions. We see this, for example, in the `FileSystem` (handler) and `trackdisk.device`. The `FileSystem` uses a "higher" disk structure (files and directories), while the `trackdisk.device` works much "deeper" (tracks and sectors). In our example we want to `Read()` a file that was opened from DOS. DOS knows that no valid data is present in its buffer. It sends a packet to the respective `FileSystem` and asks for more data (`dn_Type` = `Action_read`). The `FileSystem` knows which sectors contain the next valid data of this file. After that it checks if these sectors are present in the buffer list. (The buffer holds a number of sectors which can be increased with `AddBuffers`). It picks up the desired data from the buffer and copies it into the given DOS buffer.

When the data is not in the buffer of the `FileSystem` a message is sent to the respective `trackdisk.device` which reads these sectors into the buffer of the `FileSystem`. The `trackdisk.device` reads an entire track. It first looks to see if the desired sectors are in the track buffer. When this is the case, the sectors are copied out of the track buffer and into the buffer of the `FileSystem`. Otherwise it prepares the track on which the sectors are found and reads them into memory.

Accessing the disk data uses many stations (DOS, `FileSystem` (handler), `trackdisk.device`, disk hardware), and each station takes another step towards the actual hardware. This is why disk access on the Amiga is so slow. Every station stores something in between, searches through its buffer for the desired data, and sends information to other tasks. There are more reasons for the slow disk access speed, for example the manner of distribution of data on the disk and the fact that the features of the Amiga hardware are not used to full advantage by the `trackdisk.device` (index synchronization).

There is one exception in this task distribution (DOS, handler, device, hardware) the RAM disk of Version 1.2. For there is only one task, RAM, which functions as a handler as well as a device, but does not support all possible actions. The 1.2 RAM disk cannot be formatted for example. Version 1.3 has the RAM disk named RAD: that isn't erased when the computer is reset. This consists of a RAM handler and the `ramdrive.device`. With this new construction all actions (including formatting) can be executed by the new RAM disk.

The handlers are found in directory `l` and the devices in `devs`.

## 8.2 Relocatible Programs, Segments, BPTR Pointer

### *Relocatible Programs*

Because the Amiga has a multitasking operating system, the addresses where programs are loaded are not fixed. Otherwise a second program could not be loaded while another program was in its memory location. AmigaDOS searches for a large enough section of memory before it loads a program and fits all of the addresses in this memory section. A programmer doesn't usually have to worry about this because the C compiler or assembler stores the necessary information on disk. Trying to debug a program that always loads to a different address can be confusing. When a program is loaded and examined with a monitor or debugger, it's in a different position the next time it's loaded. This fact makes the absolute address location useless.

A great help for debugging programs and the Amiga operating system is the *AssemPro* assembler from Abacus. It contains a good assembler, debugger and a reassembler. What's the advantage of this system? When you load a program into the debugger you can create a source file with the reassembler. You can then comment this source code for your own purposes. In most cases *AssemPro* can translate from the assembler into a running program.

### *Segments*

The Amiga operating system allows multiple programs to run at the same time, which can cause a few difficulties with the free memory. This free memory doesn't lie in one piece (segment), but in many pieces between the sections that are taken up by programs. There can be 500K free, but if the largest section is 80K, a 90K program could not normally be loaded. The operating system controls this and places the program in many small segments. This allows the program to be stored in small memory sections and the sections to be built into a running program. Here again AmigaDOS is careful that the addresses of the single program section fits in the proper position of the segment. The basic division into segments happens with the help of a compiler or assembler.

AmigaDOS links the individual segments for each program to be loaded. Two long words (BPTR pointer) are placed before each segment. This long word points to the address of the next segment. With the help of this segment list it's possible to find all the sections of a program in memory and to reconstruct them into the source code with the help of a reassembler.

***BPTR pointer***

C programs, include files or documentation of the Amiga operating system always contain the mysterious *BPTR pointer*. The BPTR pointer is basically the invention of AmigaDOS in BCPL. In C a pointer indicates the direct memory location where the object is. A BPTR pointer is such a pointer divided by four. The BPTR points to 2500 if a pointer indicates an object at memory location 10000. This way no information is lost because the BPTR points only to objects that are at a long word address. When you divide such an address by four and then multiply by four again, the same address is received.

You want to make this concept concrete by using an example. When you have a segment list that ties the various segments of a program together, the pointer for the next segment is a BPTR pointer. Segments can only lie at long word addresses. To get the correct address from the BPTR pointer you must multiply the value by four.

## 8.3 The CLI Program

Although we mostly speak about the CLI as a user interface including the CLI window and CLI programs in drawer C:, there is also a CLI program, which is the starting program for this user interface. It is found either in the System drawer or directly in the main directory of the Workbench disk.

The CLI program is written in C, unlike most of the other CLI commands. Looking at the structure of this program, you can see that it has astonishingly few tasks.

***CLI program structure***

The beginning contains the usual introduction that every C program has. It's first tested to see if the program was started from the Workbench or the CLI. The command line is analyzed and split apart into the parameters if it was started from the CLI.

The actual Main() program tests to see if a CLI window already exists. If is does not, 10000 bytes of memory are reserved, the prompt becomes %N>, and the actual directory is set to SYS:

In each case a window is opened. This is done by the DOS function Open() and is assigned the name Con:0/50/640/80/NewCLI window.

***COS & CIS***

Then the DOS function Execute is called. This requires three parameters: a command string, and an input and output handle. A null is assigned to the string, a null is assigned to the output handle and the opened CON window is assigned to the input handle. Execute sets CIS (Command Input Stream) and COS (Command Output Stream) to the assigned file handle, loads and starts the CLI command C:Run. This reads input from the CON window, executes the command, and writes the results to the CON window until an EndCLI is entered. Then CIS and COS are restored and Execute returns.

The CLI program then closes the window, frees the memory and ends.

***The Run command***

You may have noticed that the CLI program does very little. The actual work is finished with the CLI command Run. Even more surprising is the fact that the CLI program is not capable of running without the CLI command Run. We can easily show this by an example:

Create a c directory on the RAM disk and copy only the Assign command into this directory. Set c: to ram:c with the following.

```
makedir ram:c
copy sys:c/assign to ram:c
assign c: ram:c
```

Now start the CLI program from the Workbench. The CON window appears shortly, but then it disappears again. That is because c:run was not found. Do not erase the c:Run command from the Workbench disk. Re-assign the C directory with the following:

```
assign c: sys:c
```

With this knowledge about the functions of the CLI program it is very simple to write your own CLI program. The important points are:

- Open a CON window
- Call the DOS function Execute with three parameters:
  - a 0 for the Execute string in D1
  - the CON window for Input handle in D2
  - a 0 for Output handle in D3
- Close the CON window

The important machine language commands (not a complete program) look like this:

```
; ------------------- OpenCliWindow ---------------------
OpenCliWindow:PEA $3ED ; Mode: Old_File
PEA conwindow          ; "CON:0/50/640/80/New Cli Window
JSR call_Open          ; calling DOS-Function OPEN
MOVE.L D0,D2           ; Filehandle from Open "CON:"
ADDQ.L #8,A7           ; Stack restoring
BEQ LC2C47E            ; Open error -> no Execute
CLR.L -(A7)            ; = Outputhandle
MOVE.L D2,-(A7)        ; = Inputhandle
PEA executestring      ; $C2B8C0 = 0
JSR call_Execute       ; loads "C:RUN"
                       ; assigns CIS and COS
                       ; Unload "RUN", CIS/COS restoration

RTS

conwindow: dc.b "CON:0/50/640/80/Mein Cli Window", 0
executestring: dc.b 0,0,0,0
```

As we said, the program is not complete because the DOS library must be opened beforehand. But this example does point out that it is very easy to write your own CLI program. Unfortunately complete control is assigned to C:Run and we can only wait until Run reads EndCLI as a command and returns control.

## 8.4 Internals of the DOS Library

Have you ever examined a C program with a disassembler or written your own machine language program? If so the following commands should look familiar:

```
MOVE.L Pointer for library,A6
JSR    OFFSET(A6)
TST.L  D0
BEQ    Error
```

This command sequence calls the routines of the operating system libraries. Look at a CLI command like Run. There you can't find such a call. Instead the following command sequence emerges:

```
LC188E8:MOVE.L D1,D6 ; save number GLOBAL VECTORS (=$95)
 ADDI.L #$32,D1      ; address $32 for AllocMem (=$C7)
 SUBA.L A0,A0        ; sets A0 to 0
 MOVE.L $74(A2),A4   ; = Allocate Memory (MEMF_PUBLIC)
                       (length=D1),Return BPTR(D1)
 MOVEQ #$C,D0
 JSR (A5)            ; AllocateMem
 TST.L D1            ; memory contents for Vector table?
 BEQ.L LC189DE       ; no -> end
 ADD.L #$32,D1       ; yes, further
```

Don't worry if you don't understand all the comments. This is only a small section of the CLI command Run. The following commands are interesting:

```
MOVE.L OFFSET(A2),A4
JSR   (A5)
TST.L  D1
```

What's interesting is that you find different offsets every time the same subprogram is called with JSR(A5). Here the CLI commands reach into an internal library. The basic address of the library must be stored in A2, the address of the routine in A4, and the subroutine in A5 calls this routine. This has two interesting consequences:

- CLI commands are relatively short and compact, always shorter than a comparable C program

- CLI commands need more information than a C program. For example, the address of the internal library. Because this address is not assigned from the Workbench, CLI commands cannot be started from the Workbench without help. For example, if you give the Run command an icon and then double-click on this icon from the Workbench, you would get a Guru Meditation.

How do CLI commands work with the Global Vector Table? The next section will go into more detail, but for now the following is interesting: Most CLI commands copy the Table into a free memory area at the beginning and later add some routines there. So every CLI command has its own table. The standard table contains $96 entries ($00-$95), see the comments in the first line of the previous program section.

Have you ever used the Status Full command, which displays important information?

```
status full
task 1: stk 3200, gv 150, pro 0 loaded as command: status
```

Here the mysterious entry gv appears. It usually has a value of 150. That is the decimal value for $96. GV displays the size of each process Global Vector Table. It is questionable as to why this value is shown because it is rather uninteresting for users and up until now we have not found any information about this table. This table is usually called the Global Vector Table because it holds all CLI processes as ordered.

The section of the Run program that you printed is the preparation **for** copying this table. The important memory allocations for this is **also** found here.

The CLI commands get more information from the processor **register.** There important variables can be accessed without large **computing** costs. Our investigations have revealed the following contents:

| | |
|---|---|
| D0 | Number of parameter characters in the command line |
| D2 | Size of the program stack |
| A0 | Pointer for parameter characters |
| A2 | Pointer for internal DOS library |
| A5 | Pointer for routine to call the functions |
| A6 | Pointer for return routine |

For example, if the Run command was called without any parameters (like the DOS function Execute does), the register assignment would look like this:

```
;-------------------- Example-Register ------------------
; D0 = 00000001 -> Only command name "Run"
; D1 = 00C5AA7C
; D2 = 00000FA0    = 4000 = Program stack for calling CLI
; D3 = 00000FA8
; D4 = 00000001 -> length of the parameter strings ?
; D5 = 0000003E
; D6 = 00309107
; D7 = 00C5A794

; A0 = 00C27411 -> Pointer:1 Parameter after "run" (010A)
; A2 = 00C04CA0 -> Pointer for DOS-Library
; A3 = 00C6F52C
; A4 = 00C6BB48
; A5 = 00FF44B4
; A6 = 00FF44A8 -> Routine that ends the function call
; A7 = 00C721C4 -> return address
```

It's interesting how the parameter string is stored. This isn't done by the usual C convention (character string concluded by null), but instead the string length is in the first byte and then the characters follow. The assigned string in our example reads:

```
$01,$0A     (1 character, and then a linefeed)
```

***BPTR***

Pointers for the strings are known as *BPTR*. BPTR are pointers that must be multiplied by four before they are used.

We believe that you can use the Global Vector Table with your routines for some C commands with no problems, as long as you handle the existing CLI commands in the conventional method. Here Commodore can hardly make a change, because all of the CLI commands that are supplied would no longer function.

## 8.5 The Run Command

In this section we want to look at the Run command. We can show more about how the DOS commands are built with this example. We are especially interested in how vectors can be entered in the copy of the Global Vector Table (GVT). There is a completed routine in the GVT for this. It is easier to analyze other CLI commands, by doing this maybe we can find some tricks to put in our own programs.

Basically, most CLI commands are in two segments (see Section 8.1). In the first segment the preparation for the actual running of the program is done:

- First a loop checks all the segments of the program for the size of the Global Vector Table (GVT). The last long word in the segment contains this size. The minimum value is #$95 = 149 because the table contains 150 vectors. If this check finds a higher value, it reserves the necessary size.
- Then the necessary memory is prepared for the table. #$32 long words are added for this and this memory is set aside using AllocMem.
- The GVT is copied into the free memory from ROM. An internal routine of the DOS library is used for this

```
MOVE.L $70(A2),A4    ; = Fill in a Global Vector Table
MOVEQ #$C,D0
JSR (A5)             ; Fill the table
```

This is called twice because the GVT is stored in two different places. The 15 long words are copied into the free memory from the CLI structure of the actual tasks. This structure is in the GVT at $218 and occupies the same place in the new table. Now this routine is called for the second segment of the same program. The routine "fill in Global Vector Table" waits in D1 for a BPTR from the segment list of the program and returns a -1 if everything is OK, otherwise it returns a 0.

The construction of the segment must be as follows: The last long word of the segment contains the maximum size of the table. This is also the largest possible vector number. Before that is the offset of the entered Vector and before that the Vector number. An offset of null ends this operation. The end of the second segment of Run looks like this:

```
$00000000 $00000001 $00000024 $00000096
End       Vector #  Offset    Max. table size
```

In this case Vector 1 was entered, the address of the respective subroutine begins at segment 2+$24 and the following $00000000 as the offset ends this segment. Because no more segments exist, the routine "Fill in a Global Vector Table" ends.

- Vector 1 is overwritten by the address of a routine that is in the first segment and cannot be entered from the copy routine into the table. This routine frees up the memory of the newly placed Vector Table and ends the CLI command. After closing the command, the Vector Null is called. This points to the program start of the second segment so that the program section stored there is processed. The program code of the second segment starts and a null doesn't appear at the beginning, but at $24 because the name of the CLI command starts before that. That looks like this:

```
;-----------------------------------
; RUN Segment 2
Offset: $00
L00C1A610:DC.B $30,$39          ; "09"
DC.B $11,$52,$55,$4E            ; ".RUN"
DC.B $20,$20,$20,$20
DC.B $20,$20,$20,$20
DC.B $20,$20,$20,$20
DC.B $20,$20,$00,$00            ;$20,$20
DC.B $07,$73,$74,$61            ; ".sta"
DC.B $72,$74                    ; "rt"
```

The code for Vector 0 of the Run command is $04 in Version 1.3. The most important routine of Vector 0 like the CLI program is very short. Beforehand important preparation takes place, like calling Input(), Output() and AllocMem(). The fixed portion looks like this:

```
LC1A860:MOVE.L $3C(A1),$68(A1)
 LEA $3F4(A4),A3
 MOVE.L A3,D4          ;get pointer for TASK NODE names
 LSR.L #2,D4           ;BPTR: TASK NODE names in pointer
 MOVE.L $34(A1),D3     ;get priorities of the processes
 MOVE.L #$320,D2       ;size of the stacks in long words
 MOVE.L $20(A1),D1     ;BPRT from segment list for process
 MOVEQ #$58,D0
 MOVE.L $84(A2),A4     ;Create a Process
 ;                    D1 = BPTR to SegList array for process
 ;                    D2 = Stack for process in long word
 ;                    D3 = Priority of Process
 ;                    D4 = BPTR to task node name
 JSR (A5)
 MOVE.L D1,$1C(A1)      ; Pointer for process Message Port
 TST.L D1               ; was process furnished?
 BNE.S LC1A892          ; No-> "Can't Create Background
Task"
 BRA.L LC1A9C0          ; yes, further, free up memory, End!
```

***Background Task***

In reality, the `CLI` command `Run` furnishes a new task with the name *Background Task* that you can examine it if you have used the `Task` command.

We hope this information has helped you understand the `CLI` commands. We hope that through this knowledge you will understand the efficiency and capabilities of the AmigaDOS and use the commands in your own programs. We know that with commands like `Assign` and `AddBuffers` better commands can be created that will make work with AmigaDOS and the `CLI` even easier and more efficient for the average user.

# 9. Creating `CLI` Commands

# 9. Creating CLI 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. Because the CLI commands are not integrated into the operating system but instead are small programs on the Workbench disk, you can add new commands. As a CLI expert, you have to look in the c directory on the Workbench disk for the CLI commands. DOS 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 CLI command can be found this way.

The number of commands is not limited or set. You can copy your favorite CLI 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 CLI 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 CLI command, can usually only be created by using an assembler. The language the commands were written in is BCPL, which most people don't have for their Amiga. The C language is an alternative. It was derived from BCPL and a large portion of the Amiga operating system was programmed with it. When creating new commands you should stick to a particular programming style. You should at least pay attention to the following items:

1. All CLI commands are "non-interactive". That 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 CLI window from which they were called. A command should not open its own window for output.

3. True CLI 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 fitting remark. 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
```

## 9.1 CLI Commands in C

A command written in C and a CLI command cannot be told apart if the programming is done correctly. 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 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 the CLI 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. (Use -lm -lc options to link.

```
/* 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 the CLI 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 the CLI 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 is still possible to receive data from other devices. The following example program shows how to do this:

```
/* RedirectInput 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 is 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 the CLI) 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 the CLI command line switches the standard input device to the given device.

## 9.2 Task

***The Task command***

You have probably wanted to know what the Amiga really does when you don't give it any tasks. There is the Status command, but this command shows only the CLI processes and their tasks and not programs that were started from the Workbench or the internal activities of the Amiga. It would be better if you could see which tasks the Amiga really executed internally. With this information you could influence these tasks. To do this we have created two new CLI commands: Task, which notifies you of tasks, and Taskpri, which allows you to influence these tasks. We'll begin with Task. This command notifies you of not only active tasks, but also gives their priorities.

***Start address***

Before we give the Task command as a C program, we want to give you some basic information about its functions. The Amiga has an operating system that uses totally relocatable programs. No address is certain; every program can be placed at any location. Relocatible programs can be loaded into any address. There is one permanent address, and that is the start address of Execbase. This is the *start address* of the basic table of the Amiga operating system and the address is always at memory location $000004. Only the address of Execbase is stored here. This is a different location for an Amiga 500 with memory expansion than it is for an Amiga without one. Inside of Execbase there are many important pointers for all devices, libraries, etc. It also contains the address of the list where the tasks are managed. Basically, there are three status possibilities for each task:

| | |
|---|---|
| *Running* | The task is active and being executed |
| *Ready* | The task can be worked on next |
| *Waiting* | The task is waiting for a call and cannot be executed until it is called |

There is only one task in the Running condition, while many can be Ready or Waiting. If a call comes for a task in the waiting list, it's automatically placed in the Ready list. The construction of this list is not simple and we won't get into it in this book. It isn't completely true when we say that Task shows all tasks. It displays only those in the Waiting list. That is hardly a limitation because only one task can be Running and several more tasks Ready. Should you be interested in more information, look in ***Amiga System Programmer's Guide*** by Abacus. The following is a program for the new CLI command name Task.

```
#include <exec/types.h>
#include <exec/execbase.h>
#include <exec/tasks.h>
```

```
#include <exec/exec.h>
#include <exec/execname.h>
#include <exec/lists.h>
extern struct ExecBase *SysBase;
main()
{
    struct Task *task;
    char *names [20];
    int pri [20];
    int count,i;
    count = 0;
    Disable();
    for (task = (struct Task *)SysBase->TaskWait.lh_Head;
    task->tc_Node.ln_Succ;
       /* End, when the pointer for the next entry = 0 */
   task = (struct Task *)task->tc_Node.ln_Succ)
       {
          names[count] = task->tc_Node.ln_Name;
          pri[count++] = task->tc_Node.ln_Pri;
       }
    Enable();
    for (i = 0; i < count; i++)
       {
          printf ("%s",names[i]);
          printf ("\r\t\t\t%d\n",pri[i]);
       }
}
```

***How the program works***

First, all important header files are included into the source file and a pointer with the name `SysBase` is defined, it points to `Execbase`. Now the important variables must be defined. You need a pointer for the structure `Task`. You don't need to worry about searching for the structure because it's already present in the task list. Because the contents of this `Task` structure can be changed, you can't directly display the names on the screen, but instead the address must be entered in its own field. The `Disable()` function is useful for disabling task switching. From this point on do not change the active task and the list. Go through the list until you establish its end.

```
task->tc_Node.ln_Succ; /* End, if pointer for next entry = 0 */
```

***The Strncpy function***

Then use `Enable()` to turn the task switching back on and display the list. Take the chance that a name in the list has changed in the meantime, because you only saved the pointer for this name. You must save the name with the copy function `Strncpy`. The maximum number of entries for this program is 20, and this is sufficient in most cases.

Started from a third `CLI`, the output of our program looks like the following, your display will differ:

```
Background CLI          0
CON                     5
CON                     5
New CLI                 0
New CLI                 0
Workbench               1
File System            10
input.device           20
trackdisk.device        5
File System            10
trackdisk.device        5
CON                     5
```

***Background CLI***

The task named `Background CLI` is responsible for the TextPro program in this case. It was started from the `CLI` using `Run`. The three `Con` tasks take care of window input and output. The `NewCLI` task controls `CLI` processes started with `NewCLI`. The Workbench also furnishes a task and places it in the tasks that the Amiga has to finish. The names `File System` and `trackdisk.device` will be repeated for each disk drive, they will appear twice if you own two disk drives.

## 9.3 TaskPri

This new command is an extension of the ChangeTaskPri command discussed in Section 7.6. TaskPri is capable of more: A disadvantage of ChangeTaskPri is that only the priority of the actual CLI can be changed. It cannot change the priority of another CLI. TaskPri makes it possible to manipulate not only the priority of any process but to manipulate any task displayed by the Task command. Because this command needs input from the user (the name and new priority of a task), it's quite large. Here is the C listing for our new CLI command (ChangeTaskPri.C):

```
#include <exec/types.h>
#include <exec/execbase.h>
#include <exec/tasks.h>
#include <exec/exec.h>
#include <exec/execname.h>
#include <exec/lists.h>
extern struct ExecBase *SysBase;
long FindTask();
int settaskpri();
main(argc, argv)
int argc;
char *argv[];
   {
   struct Task *MyTask;
   long pri;
   int result;
   char oldpri;
   long task;
   pri = 0L;
   if (argc != 3)
      {
      printf(" Task name  priority\n");
      exit(FALSE);
      }
   result = Value(argv[2], &pri);
   if (result == FALSE)
      {
      printf ("Not in range!\n");
      printf (" 128 <= Taskpri <= 127 : %ld\n", pri);
      exit(FALSE);
      }
   task = FindTask (argv[1]);
   if (task == 0)
      {
      printf ("Task <%s> not found !\n", argv[1]);
      exit(FALSE);
      }
```

```
   oldpri = ( char ) SetTaskPri (task, pri);
   printf ("Old Priority was: %d  new is %ld\n", oldpri,
pri);
   exit (TRUE);

}

/* Up to three numbers 0 - 255  */
Value(string, value)
char *string;
long *value;
   {
   int i;
   int numdigits;
   char sign;
   int numeral;
   int tens;
   tens = 1;
   numdigits = 0;
   if (*string == '-') /* negative number */
      {
      tens = -1; /* change to negative number */
      string++; /* overlook minus sign */
      }
      for (i = 1; i <= 4; i++) /* number of digits */
      {
      if (*string == 0)
         break;
      string++;
      numdigits++;
      }
  if ((numdigits == 0) || (numdigits > 3))
                    /* only 3 digits are allowed */
     return (FALSE);
  for (i = numdigits; i > 0; i-- )  /* create number */
     {
     string--;
     sign = *string; /* read string back */
     if (sign < '0' || sign > '9')
        return (FALSE);
     numeral = sign - '0';
     *value = *value + numeral * tens;
     tens = tens * 10;
     }
  if (*value < -128 || *value > 127)
     return (FALSE);
  return (TRUE);
  }
```

***How the program works***

The first seven lines set up the task of the program. After that the external function `FindTask()`, which is found in the `Exec` library, is declared. Its job is to look in the task list for an entry whose name has been entered on the command line. The function must be given a pointer for the name to find. A pointer to the correct file is returned as a result if the file is found. Unlike many other functions on the Amiga, the `FindTask` function requires the correct use of upper case and lower case letters. `FindTask` doesn't find a task if it isn't entered exactly as it appears in the output of the `Task` command.

In this program, when multiple tasks have the same name, the pointer for the first one is returned and that is all. This problem can be solved, but would require a large amount of programming time fixing it.

***The SetTaskPri() function***

The `SetTaskPri()` function is declared as the second external function. The name describes what the function does: The priority of a task can be set to any new value. The function needs the new priority and the structure pointer returned by `FindTask` as a parameter. The return value contains the previous priority of this task.

***The Value() function***

The main function of the `TaskPri` program is like that of the previously mentioned `Task` command. The evaluation of the given parameters is programmed exactly as it was described in the beginning of this chapter. This time there is also a small syntax check that gives a message when incorrect input is made: The command should only be given two additional parameters (`argc !== 3`) otherwise the `main` function exits. `Taskname Priority` appears on the screen to inform the user of the required parameters. This information will be displayed when you enter a space and a question mark following the command, thus giving you the argument template.

The correct parameters are converted to long format by the `Value()` function because `SetTaskPri` cannot process ASCII values. `Value()` expects a pointer to the input data as a parameter (`argv[2]`) and also the address of a variable where the converted result can be stored (`&pri`). `Value()` returns `False` if a matching function isn't found or if the permissible parameters are not in the correct range. The remainder of the main function explains itself: In case the desired task structure is found—`FindTask` supplies a value other than null—the new priority is entered here and the previous value is displayed in the `CLI` window.

The `Value()` conversion function is a C problem that we will not describe in detail here. The job of the function is simply to convert the ASCII input into a value between -128 and 127.

***Using the program***

Calling our new `CLI` function is very easy: use the `Task` command to display the tasks and their priorities. Use the `TaskPri` command followed by a task name and a new priority value to change the priority of the task. The task name must be placed in quotation marks if it contains spaces (for example `File System`).

The following example demonstrates this command:

Use the `Dir` command to display the contents of the Workbench disk in the `CLI` window. Move the mouse arrow around the screen at the same time. Result: Movement of the mouse is completely normal. Now enter `TaskPri Input.device -1` from the `CLI`. The message `Old priority was: 20 new is -1` is displayed on the screen. Try the `Dir` command again and move the mouse. You'll see that the mouse arrow moves only after the directory is output.

This isn't a very meaningful example, but it does show how the new command functions. Enter `TaskPri Input.device 20` to return to normal. There is a better example:

Pretend that you would like to print out a lengthy file while you are writing a new letter with your word processor. In some cases the speed of the word processor will be slowed down considerably. It may not be able to keep up with the processing of the letter and printing. In this case it is helpful to set the priority for the word processor at +5 and the priority of the `Printer.device` at -5. The printing may take a few minutes longer, but you won't notice this because you are writing the letter.

A similar problem arises with compiler languages such as C: If you try to make changes to a source file during the compilation operation, `ED` reacts very slowly to input. Setting the compiler's priority at a lower number prevents this effect.

## 9.4 TaskStop

Now we'll present a way of stopping all actions on the Amiga for a period of time by just pressing a key. What good is this? Game players find this useful: The phone rings in the middle of a game, and by pressing the left <Shift> key the game goes "on hold." Say it is your boss on the phone and he wants to know how your work with the new Professional DataRetrieve program is progressing. You can move the game window to the back until you are finished talking to your boss. When you are ready to continue, move the game screen to the front and press the right <Shift> key. The game continues to play.

This works for programs other than games. Every program can be paused with our new CLI command. Someone who just had an idea can bring up another screen and write it down while the original program is paused. A relatively small C program (TaskStop.c) for the command follows:

```
#include <exec/types.h>
#include <exec/execbase.h>
#include <exec/tasks.h>
#include <exec/exec.h>
#include <exec/execname.h>
#include <exec/lists.h>
#include <functions.h>
#define SHIFTLEFT 0x3f
#define SHIFTRIGHT 0x3d
#define CONTROL 0x39

extern struct ExecBase *SysBase;
main()
{
    struct Task *task;
    char *keys;
    int i;

    i = 0;
    keys = 0xbfec01;     /* Address of the special keys */
    printf ("\n Shift Left = Stop, Shift Right =
Continue, Control = End\n ");
    task =  FindTask(0L);
    if ( task == 0L)
    {
    printf (" I can't find the Task \n");
    exit (0L);
    }
    SetTaskPri (task,127L);
    do
    {
```

```
    if ( *keys == SHIFTLEFT )
    {
    do
    {
    i++;
    } while ( (*keys != SHIFTRIGHT) && (*keys != CONTROL)
);
    i++;
    }
    Delay(10L);
    } while ( *keys != CONTROL );

}
```

### *How the program works*

After the integration of the required Header files the three macros are defined. They are assigned the respective key values. The pointer for Execbase is also needed for this program. In the main function, a pointer for the Task structure is defined first. The *keys pointer is set to the hardware register, from which the keyboard codes are read. This is not the best way, but for our program it is the simplest because it can react quickly to keyboard input. The 0xbfec01 address tells whether the special key was pressed or not. The key pointer is supplied with this address.

When the user presses one of the control keys, a pointer for the Task structure in the task pointer is picked by using FindTask. The program is ended prematurely if this structure isn't found: The Guru remains in the background. The task is given the highest priority (+127) if the structure is found.

So that other tasks receive enough computing time, program execution can be held up for 10-50 seconds with the Delay command. During this time other programs can be placed in the Running list. The <Shift> key doesn't work during this time. When the task is active and the left <Shift> key pressed, the program goes into a loop that cannot be interrupted by another task. When the right <Shift> key or <Ctrl> is pressed, the loop is stopped. Pressing <Ctrl> exits the program.

It isn't very easy to give concrete examples for all the uses of the TaskStop command. A small tip: As a background process that can be started with Run, the new command functions flawlessly. It can easily be integrated into the startup sequence of a self-loading game disk.

## 9.5 Delay

When a game program stopped with the `TaskStop` command is reactivated by pressing the right <Shift> key, the game immediately continues. Our new `Delay` command makes it possible to make the game speed more efficient. The advantage to this is that the program is actually slower, but not the reaction to our input.

*<Shift> keys*

The <Shift> keys are not used in the same manner as they were in the `TaskStop` command: They allow the program to be delayed dynamically. That means that the longer they are held down, the longer the delay is. There is an additional feature built in the program. All programs can be delayed with the new command. The command must have a high priority if the game program also has a high priority. Otherwise the command should have as low a priority as possible so that it doesn't interfere with the system tasks. For example, if the priority is higher than 20, the delay also works for the mouse. This is an unacceptable condition for a game. All of these problems vanish when using the `Delay` command. But first, the program....

The C listing for `Delay`:

```
/* Delay.c   */
/* Control = End, Shiftleft = slow, Shiftright = fast */

#include <exec/types.h>
#include <exec/tasks.h>
#include <functions.h>
#define CONTROL 0x39
#define SHIFTLEFT 0x3f
#define SHIFTRIGHT 0x3d
extern struct ExecBase *SysBase;
main(argc, argv)
int argc;
char *argv[];
   {
   char *key;
   struct Task *task;
   long Count;
   long pri;
   int result;
   long i;
   Count = 1L;
   key = 0xbfec01;
   pri = 0L;              /* Priority of 1 first */
   if ((argc != 2) && (argc != 3))
      {
      printf(" Delay [Priority] \n");
      exit(FALSE);
```

```
      }
   result = Value(argv[1], &Count);
   if(result == FALSE)
      {
      printf (" 0 < Wait time < 999, out of range:
%ld\n", Count);
      exit(FALSE);
      }
   if (argc == 3)
      {
      result = Value(argv[2], &pri);

      if( (result == FALSE) || (pri < -127) || (pri >
127) )
         {
         printf (" -127 < Priority < 127, out of range:
%ld\n", pri);
         exit(FALSE);
         }
      }
   task = FindTask(0L);
   if (task == 0L)
   {
      printf ("Did not find a task!\n");
      exit(0L);
   }
   SetTaskPri (task, pri);

      do
      {
        i = 0L;
        Delay( 5L );
        for ( i = 0L; i < Count * 10L; i++ )
           {
           while ( (*key == SHIFTLEFT ) && (Count < 995L
) )
              {
              Count++;
              Count++;
              Count++;
              Count++;
              Count++;
              Delay(1L);
              }
        while ( (*key == SHIFTRIGHT ) && (Count > 5L ) )
              {
              Count--;
              Count--;
              Count--;
              Count--;
              Count--;
              Delay(1L);
              }
           }
      } while (*key != CONTROL);
      printf (" Count = %ld\n", Count);
```

```
}

/* Up to three characters 0 - 999 */

Value(string, value)
char *string;
long *value;
   {
   int i;
   int numdigits;
   char sign;
   int numeral;
   int tens;
   tens = 1;
   numdigits= 0;
   *value = 0L;
   if (*string == '-')                    /* negative number */
      {
      tens = -1;            /* change to negative number */
      string++;                     /* ignore minus sign */
      }
      for (i = 1; i <= 4; i++)   /* number of numerals */
      {
      if (*string == 0)
         break;
      string++;
      numdigits++;
      }
  if ((numdigits== 0) || (numdigits> 3)) /* only
                                three numerals allowed */
      {
      printf (" Too many numerals %d\n", numdigits);
      return (FALSE);
      }
  for (i = numdigits; i > 0; i-- )  /* create number */
      {
      string--;
      sign = *string;    /* read string back*/
      if (sign < '0' || sign > '9')
         {
         printf ("Incorrect numeral %c \n", sign );
         return (FALSE);
         }
      numeral = sign - '0';
      *value = *value + numeral * tens;
      tens = tens * 10;
      }
/*   if (*value <= 999 || *value > 999)
     return (FALSE);                                     */
  return (TRUE);
  }
```

***How the program works***

The beginning of the program is taken directly from `TaskStop`. Three cases could take place after evaluating the input line:

1. Two or three parameters are not entered. The program stops and the correct parameters are displayed.
2. Exactly two parameters are entered. The delay is converted into a number and the given priority null remains unchanged.
3. Exactly three parameters are entered. Then the second parameter is converted into a value, is tested for a valid value, and the variable `Pri` is assigned to it.

***The Do/While loop***

Then a pointer for the task structure is found using `FindTask`, and the new priority is set using `SetTaskPri`. Without a given priority, our task always has the priority null. The real work takes place in the Do While loop. This is made up of three components:

1. A call of the wait function `Delay`. This makes sure that the high priority system task has enough time.
2. A delay loop that is dependent on the delay value. During this time, all lower priority tasks are blocked.
3. A double keyboard reading block where both <Shift> keys are controlled. When a key is pressed, the program checks to see if the delay value is in the allowable range. The delay value is changed if it is in the range. The five lines "`Count++`" and "`Count--`" and the following `Delay(1L)` may make you wonder. `Delay` prevents the minimum or maximum value from being reached by pressing a <Shift> key. The five lines have a double responsibility: Make sure that the value is changed quickly enough because a delay as small as `1L` is not possible. It also makes sure that the program does not go on uncontrollably. The program would not be slowed down if the lines were replaced by a `Delay`. Because the program code always has a higher priority, the program is slowed down at least a little. It would be good to investigate what happens to a program with a delay of 500. You could then determine that every time the <Shift> key is pressed the program actually speeds up.

## 9.6 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 the CLI Search command that searches for characters strings in files, but you can't replace anything with this command. We have created a new CLI 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 CLI 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 if you need 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:aaaaaaaaaaaaaaaaaaaaaaaaaa
```

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 if the string has been found yet or not. Such a string can be utilized in C using the Modulo command. We use a more complicated yet faster method. The sub-program 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 =
 }   /* 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();

   }
```

# 10. Quick Reference

# 10. Quick Reference

This chapter contains all the information you have seen so far about CLI commands and Shell 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 form.

Section 10.3 lists the CLI/Shell commands in a similar form. Hopefully this clear format will help you find commands much faster.

## 10.1 The ED Program

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 it's 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 line end 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> | toggle 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 |

## 10.2 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 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)/skl)* | searches line that contains the string sk (forward search) |
| *BF((q)/skl)* | searches line that contains the string sk (reverse search) |
| *DF((q)/skl)* | searches line containing the string sk is (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 are shown 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 .zl.* | 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 break-points) |
| *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 CLI/Shell Commands

This section briefly covers the `CLI/Shell` commands. First the correct 1.2 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 1.3 they are described. The new Version 1.3 commands are marked with the identifier `(1.3 only)`.

**`AddBuffers (DRIVE) Drive (BUFFERS) n`**

Reserves a buffer on a drive with a certain amount of memory.

| | |
|---|---|
| *DRIVE* | Optional |
| *Drive* | The drive assigned the buffer. |
| *BUFFERS* | Optional |
| *n* | The size of the buffer to be allocated |

**`Ask "Text"`**

Asks a question answered with only (Y)es or (N)o: `y` returns an error code of 5 and `n` returns no error code.

| | |
|---|---|
| *"Text"* | Contains text displayed on the screen, usually in the form of a question. |

**`Alias Name String`** **`(1.3 Shell only)`**

This command can only be used in conjunction with the `Shell`in 1.3. 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. |

**`Assign ((NAME) Name1 ((DIR) Name2)) (List)`**

Assigns a logical device to a directory.

| | |
|---|---|
| *NAME* | Optional |
| *Name1* | The logical device. |
| *DIR* | Optional |
| *Name2* | The directory assigned the logical device. |
| *List* | Lists the assignments of the logical devices. |

**V1.3 (Exists Name) (Dismount Name) (Unmount Name)**

| | |
|---|---|
| *Exists Name* | Searches for `Name` in the `Assign` list. The error code 5 is returned if `Name` is not present. |
| *Dismount Name* | Removes `Name` from the `Assign` list (developer's version only). |
| *Unmount Name* | See `Dismount` `Name`. |

**Avail (CHIP) (FAST) (TOTAL) (1.3 only)**

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. |

**BindDrivers**

Introduces additional device drivers to the system.

**Break (TASK) Number (All) (C) (D) (E) (F)**

Stops a task in process.

| | |
|---|---|
| *TASK* | Optional |
| *Number* | Points to the process to be broken off. |
| *All* | Sets the break level at C, D, E and F. |
| C,D,E,F | Sets break level. |

**V1.3 (Process)**

| | |
|---|---|
| *Process* | See V1.2 `TASK`. |

**CD Name**

Changes the directory or displays the current directory.

| | |
|---|---|
| *Name* | The drive or the directory which should be accessed. |

**ChangeTaskPri (Pri ) n**

Changes the priority of a process started from the `CLI`.

| | |
|---|---|
| *Pri* | Optional, determined by `Status` command. |
| *n* | Contains the new priority (-128 to 127). |

**V1.3 (Process n)**

| | |
|---|---|
| *Process n* | The new priority is assigned to `Process` number n. See the `Status` command. |

**Copy Name1 to Name 2 (All) (Quiet)**

Creates a copy of files or a directory.

| | |
|---|---|
| *Name1* | The source file. |
| *Name2* | The target file. |
| *All* | Copies the entire directory. |
| *Quiet* | Displays no output to the screen. |

**V1.3 (Buffer n) (Buf n) (Clone) (Date) (NoPro) (Com)**

| | |
|---|---|
| *Buffer n* | Uses n 512K buffers for copying. |
| *Buf n* | See buffer. |
| *Clone* | Date, Status bits, and comments are also copied. |
| *Date* | Date is also copied. |
| *NoPro* | The Status bits are reset when copied. |
| *Com* | The comments are also copied. |

**Date (DATE) Date (TIME) Time (To Name) (Ver Name)**

Input or output of date and/or time.

| | |
|---|---|
| *DATE* | Optional |
| *Date* | The date to be input. |
| *TIME* | Optional |
| *Time* | The time to be input. |
| *To Name* | The name of the file into which the date or the time is written. |
| *Ver Name* | See To Name. |

**Delete Name (All) (Quiet) (Q)**

Erases files and/or directories.

| | |
|---|---|
| *Name* | Gives the file or directory name to be deleted. |
| *All* | The entire directory is deleted. |
| *Quiet* | There is no message output to the screen. |
| *Q* | Abbreviation for `Quiet`. |

**Dir Name (OPT (A) (I) (AI) (D))**

Displays the directory of a disk.

| | |
|---|---|
| *Name* | Name of the disk drive or the directory (pathname). |
| *OPT A* | Shows all files in the directory including it's subdirectories and their contents. |

| | |
|---|---|
| *OPT I* | The contents are interactively output. After each file or directory the following inputs can be made.<br>? Displays the possible commands.<br>B Back up the directory (directory only).<br>E Enter the displayed directory (directory only).<br>T Type the file (files only).<br>Del The file is deleted.<br>Q Quit the `Dir` command. |
| *OPT AI* | The A and I options are combined. |
| *OPT D* | Only the directories are listed. |

**`V1.3 (ALL) (DIRS) (FILES) (INTER)`**

| | |
|---|---|
| *ALL* | See V1.2 OPT A |
| *DIRS* | See V1.2 OPT D |
| *FILES* | Only files, not directories are listed |
| *INTER* | See V1.2 OPT I |

***Note:*** When using these arguments (`ALL, DIRS, FILES, INTER`) do not include the OPT argument.

### `DiskChange Drive`

Tells AmigaDOS that a disk has been changed.

| | |
|---|---|
| *Drive* | Which drive has experienced a disk change. |

### `Diskcopy Drive1 to Drive 2 (NAME Name)`

Creates a copy of a disk.

| | |
|---|---|
| *Drive1* | The source drive. |
| *Drive2* | The destination drive. |
| *NAME Name* | Names the copy `Name`. |

### `DiskDoctor (DRIVE) Drive`

Attempts to repair errors on a disk. Damaged files may or may not be removed.

| | |
|---|---|
| *DRIVE* | Optional. |
| *Drive* | The drive the program will access. |

### `Echo "Text"`

Sends a text to the current output path.

| | |
|---|---|
| *"Text"* | Text that is output to the current output path, usually the screen. |

**`V1.3 (NoLines) (First n) (Len n)`**

| | |
|---|---|
| *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. |

### `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`).

### `EndCLI`

Exits `CLI` or `Shell` window.

### `EndIF`

Ends an `IF/EndIF` construct in a script file (see `IF`).

### `EndSkip`

Script file resumes execution at line following this command during a `Skip`.

### `Eval Value1 Operator Value2 (TO) (Lformat)` (1.3 only)

Evaluates simple expressions.

| | |
|---|---|
| *Value1* | Decimal, hex or octal value |
| *Operator* | math operator: +, -, *, /, mod, &, \|, <<, >> |
| *Value2* | Decimal, hex or octal value |
| *To* | Optional |
| *Lformat* | Specifies output format:<br>%Xn hex (n is number of digits)<br>%On octal (n is number of digits)<br>%N decimal<br>%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 (n)`**

Sets the return error code limit or returns the current return error code limit.

| | |
|---|---|
| *n* | Contains the size of the new return error code limit. |

**`Fault n`**

Prints information about a specific error.

| | |
|---|---|
| *n* | The valid error number. |

**`FF (-0) (-n)`** `(1.3 only)`

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`). |

**`Filenote (FILE) Name (COMMENT ) Text`**

Inserts a comment into a file.

| | |
|---|---|
| *FILE* | Optional. |
| *Name* | Which file will receive the comment. |
| *COMMENT* | Optional. |
| *Text* | The comment of the file. |

**`Format DRIVE Drive NAME "Name" (NOICONS)`**

Formats a disk and gives it a name.

| | |
|---|---|
| *DRIVE* | Required to specify drive. |
| *Drive* | Location of the drive containing the disk to be formatted. |
| *NAME* | Required to specify Name. |
| *Name* | The formatted disk receives the name "Name." |
| *NOICIONS* | Optional (the disk will not have an icon if this option is used). |

`V1.3 (Quick) (FFS) (NoFFS)`

| | |
|---|---|
| *Quick* | Only formats root and boot blocks. |
| *FFS* | The `FastFileSystem` is used to format. |
| *NoFFS* | The `FastFileSystem` is not used. |

**`Getenv Name`** `(1.3 only)`

This command reads the contents of an environment variable.

| | |
|---|---|
| *Name* | The label of the variable whose contents should be read. |

**`IconX`** `(1.3 only)`

Assigns icon and data to a script file. This lets you access the script file from the Workbench using the mouse with the help of this program (see chapter 6).

**`If (Not) (Warn) (Error) (Fail) (Text1 EQ Text2) (Exists Name) Command1 (Else Command2) Endif`**

This command 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`. |
| *Exists Name* | Condition fulfilled when file `Name` is accessible. |
| *Command1* | Executes `CLI Command1` when a condition is fulfilled. |
| *Else Command1* | Executes `CLI Command2` when the condition is not fulfilled. |
| *EndIF* | Ends the `IF` block. |

**`Info`**

Displays information on the screen about connected disk drives.

`V1.3 (Device)`

| | |
|---|---|
| *Device* | Specifies a device. |

**`Install (DRIVE) Drive`**

Converts a blank formatted disk into a boot disk.

| | |
|---|---|
| *DRIVE* | Optional. |
| *Drive* | The drive which contains the disk to be installed. |

`V1.3 (NoBoot) (Check)`

| | |
|---|---|
| *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. |

**`Join Name1 Name2 (AS) Name3`**

Joins two or more files together.

| | |
|---|---|
| *Name1* | First of the two files to be joined together. |
| *Name2* | Second of the two files to be joined together. |
| *AS* | Optional. |
| *Name3* | The file to which the joined files are written. |

**`V1.3 (To)`**

| | |
|---|---|
| *To* | Functions exactly like `AS`. |

**`Lab Text`**

Defines a string as the branch label for a script file.

| | |
|---|---|
| *Text* | The string to be defined as a label. |

**`List (Name) (Pat Pattern) (P Pattern) (Keys) (Dates) (NoDates) (To Name) (S Text) (Since Date) (Upto Date) (Quick)`**

Lists data about files.

| | |
|---|---|
| *Name* | Displays only information about the file `Name`. |
| *Pat Pattern* | Displays only the files specified in `Pattern`. |
| *P Pattern* | See `Pat Pattern`. |
| *Keys* | Displays the number of header blocks of the file or directory. |
| *Dates* | Displays the date. |
| *NoDates* | Suppresses the date. |
| *To Name* | Sends the output to the file `Name`. |
| *S Text* | Displays information about file whose name is contained in `Text`. |
| *Since Date* | Displays only the files created since `Date`. |
| *Upto Date* | Displays only the files created before `Date`. |
| *Quick* | Displays the filename only. |

**`V1.3 (Block) (NoHead) (Files) (Dirs) (LFormat="Text")`**

| | |
|---|---|
| *Block* | The file size is given in blocks |
| *NoHead* | The information is suppressed |
| *Files* | Lists only the files |
| *Dirs* | Lists only the directories |
| *LFormat="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. |

**`LoadWB -Debug`**

Loads the Workbench from the `CLI` or the `Shell`.

| | |
|---|---|
| *-Debug* | Adds a hidden menu with the debugging commands `Debug` and `FlushLibs`. |

**`Lock (DRIVE) Drive (On Password) (Off Password) (1.3 only)`**

Prevents or allows access to a hard drive partition.

| | |
|---|---|
| *DRIVE* | Optional |
| *Drive* | Contains the protected hard disk partition. |
| *On Password* | Prevents access to the hard drive partition. Access is restored after entering the password (max. 4 characters). |
| *Off Password* | Removes an existing password. This command functions only with Kickstart 1.3. |

**`MakeDir Name`**

Creates a new directory with the name `Name`.

| | |
|---|---|
| *Name* | The name of the new directory. |

**`Mount (Device) Name`**

Creates a device.

| | |
|---|---|
| *Device* | Optional |
| *Name* | A new device name. |

**`V1.3 (From Name)`**

| | |
|---|---|
| *From Name* | Removes parameters from the file `Name` instead of the `Devs/Mount-list` file. |

**`NewCLI (Con:x/y/Width/Height(/Text)) (From Name)`**

Opens a new `CLI`.

| | |
|---|---|
| *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 `CLI` window opens; if no filename is given the default file is `s:CLI-startup`. |

**`NewShell (Newcon:x/y/width/height(/Text)) (From Name)`** **`(1.3 Shell only)`**

This command opens a new `Shell` window.

| | |
|---|---|
| *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`. |

**`Path (Name ADD) (Show) ((Name) Reset)`**

Displays or changes the pathname.

| | |
|---|---|
| *Name ADD* | Adds a path to the directory `Name`. |
| *Show* | Shows the current path. |
| *Name Reset* | Deletes all paths up to the `c` directory and the path `Name`. |

**`V1.3 (Quiet)`**

| | |
|---|---|
| *Quiet* | Suppresses output from the current output channel. |

**`Prompt Text`**

Changes the `CLI` or `Shell` prompt string. The `Shell` in V1.3 can use `%s` to display the currently directory.

| | |
|---|---|
| *Text* | Formats the prompt's appearance; `%n` displays the `CLI` process number. |

**`Protect (FILE) Name (FLAGS) Status`**

Determines what sort of protection data should have.

| | |
|---|---|
| *File* | Optional. |
| *Name* | The name of the file to protect. |
| *Flags* | Optional. |
| *Status* | 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. |

**`V1.3 (+) (-) (Add) (Sub)`**

| | |
|---|---|
| + | Sets the status of the given `Status` bit. |
| - | Removes the status of the status bit. |
| *Add* | See + |
| *Sub* | See - |

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.

**`Quit (n)`**

Stops execution of a script file and returns an error code.

| | |
|---|---|
| *n* | Error code. |

**`ReLabel (DRIVE) Drive (NAME) Name`**

Changes the name of a disk.

| | |
|---|---|
| *DRIVE* | Optional. |
| *Drive* | The drive containing the disk to be renamed. |
| *NAME* | Optional. |
| *Name* | The new name of the disk. |

**`Remrad`** **`(1.3 only)`**

This command erases all files from the reset-resistant RAM disk. The `Ramdrive.Device` is also removed after the next boot.

**`Rename (FROM) Name1 (TO (AS) Name2`**

Renames files.

| | |
|---|---|
| *FROM* | Optional. |
| *Name1* | Name of the data which is to be renamed. |
| *TO* | Optional. |
| *AS* | Optional. |
| *Name2* | The new name. |

**`Resident (NAME) File (Remove) (Add) (Replace) (Pure) (System)`** **`(1.3 only)`**

This command erases, replaces, or includes a new command in the list of resident commands.

| | |
|---|---|
| *NAME* | Optional. |
| *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. |

**`Run Command`**

Runs a program in the background.

| | |
|---|---|
| *Command* | An AmigaDOS command to run in the background. |

**`Search (FROM) Name (SEARCH ) Text (All)`**

Searches data for a string.

| | |
|---|---|
| *FROM* | Optional. |
| *Name* | The file to be searched. |
| *SEARCH* | Optional. |
| *Text* | The string to be searched for. |
| *All* | Searches all directories and subdirectories. |

**`V1.3 (NoNum) (Quiet) (Quick) (File)`**

| | |
|---|---|
| *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. |

**`SetClock (OPT Load) (OPT Save)`**

Transfer the system date and time to and from the clock.

| | |
|---|---|
| *OPT Load* | Loads date and time from the internal clock. |
| *OPT Save* | Saves system time and date to the internal clock. |

**`V1.3 (Load) (Save)`**

| | |
|---|---|
| *Load* | Loads date and time from the internal clock. |
| *Save* | Saves system date and time to the internal clock. |

**`SetDate (FILE) Name (DATE) Date ((TIME) Time)`**

Inserts a date or time into data.

| | |
|---|---|
| *FILE* | Optional. |
| *Name* | File into which the date and time are inserted. |
| *DATE* | Optional. |
| *Date* | The date assigned to the file. |
| *TIME* | Optional. |
| *Time* | The time assigned to the file. |

**`Setenv Name String`** **`(1.3 only)`**

Assigns a string to an environment variable.

| | |
|---|---|
| *Name* | The label of the variable |
| *String* | The character string to be assigned to the variable |

**`SetPatch`**

Patches ROM in 1.2/1.3 Kickstart, enabling recoverable alerts.

**`Skip Text`**

Jumps within a script file to a defined label.

| | |
|---|---|
| *Text* | Contains the string defined as a label. |

**`Sort (FROM) Name1 (TO) Name2 (Colstart n)`**

Alphabetically sorts a file and saves it to another file.

| | |
|---|---|
| *FROM* | Optional. |
| *Name1* | The source filename. |
| *TO* | Optional. |
| *Name2* | The new file the sorted data is written to. |
| *Colstart n n* | The line after which the text is sorted. |

**`Stack (n)`**

Changes the stack size or returns the current size.

| | |
|---|---|
| *n* | The stack size in bytes. |

**`Status (Process) Number (Full) (TCB) (CLI) (All)`**

Outputs information about `CLI` processes.

| | |
|---|---|
| *Process* | Optional. |
| *Number* | Selects the task number which should be displayed. |
| *Full* | Combines the `TCB` and `CLI` options. |

| | |
|---|---|
| *TCB* | Displays information about priority, stack size and global vector size. |
| *CLI* | Displays the status of the current command process. |
| *All* | See `CLI`. |

**`V1.3 (Com=Command) (Command=Command)`**

| | |
|---|---|
| *Com=Command* | Searches for the `CLI` command `Command`. |
| *Command=Command* | See `Com=Command`. |

**`Type (FROM) Name1 ((TO) Name2) (OPT(N) (H))`**

Displays the contents of a file.

| | |
|---|---|
| *FROM* | Optional. |
| *Name1* | The source file. |
| *Name2* | The destination file to which `Name1` is copied. If a name isn't given the file appears on the screen. |
| *OPT N* | The lines are displayed with line numbers. |
| *OPT H* | The characters are displayed in hex and ASCII characters. |

**`V1.3 (TO) (Hex) (Number)`**

| | |
|---|---|
| *TO* | Entered in conjunction with `Name2`. `Name2` is overwritten without question if it already exists. A message saying that `Name2` exists already appears if `TO` is omitted. |
| *Hex* | See V1.2 `OPT H`. Do not use `OPT` with this argument. |
| *Number* | See V1.2 `OPT N`. Do not use `OPT` with this argument. |

**`Wait (n) (Sec) (Secs) (Min) (Mins) (Until  Time)`**

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 Time* | Waits until the input time. |

**`Which Name (Nores) (Res)`** **`(1.3 only)`**

This command searches for and displays the path of a command (helps locate the command's location on disk).

| | |
|---|---|
| *Name* | Name of the command to search for. |
| *Nores* | Suppress search in resident list. |
| *Res* | Limits the search to the resident list. |

**`Why`**

**Returns information about the last error that occurred.**

# Appendix

# Appendix

## *Command and editor sequences in the CLI/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 is 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 |

# Index

# PROFESSIONAL DataRetrieve

## File your other databases away!

**Professional DataRetrieve,** for the Amiga 500/1000/2000, is a friendly easy-to-operate *professional level data management package with the features of a relational data base system.*

**Professional DataRetrieve** has complete relational data management capabilities. Define relationships between different files (one to one, one to many, many to many). Change relations without file reorganization.

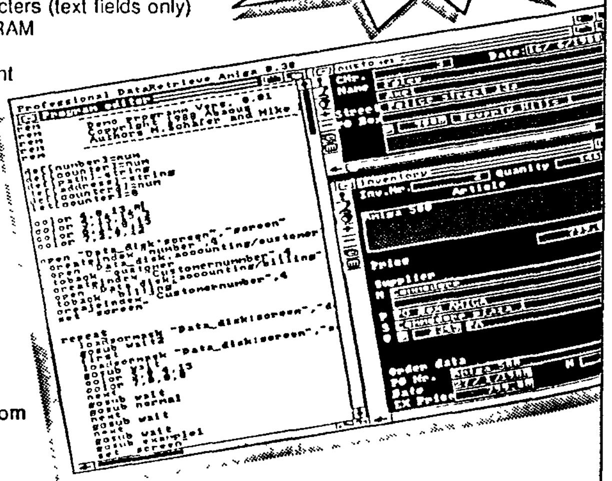
**Professional DataRetrieve** includes an extensive programming language which includes more than 200 BASIC-like commands and functions and integrated program editor. Design custom user interfaces with pulldown menus, icon selection, window activation and more.

**Professional DataRetrieve** can perform calculations and searches using complex mathematical comparisons using over 80 functions and constants.

**$295.00**

### Professional DataRetrieve's features:

- Up to 8 files can be edited simultaneously
- Maximum size of a data field 32,000 characters (text fields only)
- Maximum number of data fields limited by RAM
- Maximum record size of 64,000 characters
- Maximum number of records disk dependant (2,000,000,000 maximum)
- Up to 80 index fields per file
- Up to 6 field types - Text, Date, Time, Numeric, IFF, Choice
- Unlimited number of searches and sub-range criteria
- Integrated list editor and full-page printer mask editor
- Index accuracy selectable from 1-999 characters
- Multiple file masks on-screen
- Easily create/edit on-screen masks for one or many files
- User-programmable pulldown menus
- Operate the program from the mouse or from the keyboard
- Calculation fields, Date fields
- IFF Graphics supported
- Mass-storage-oriented file organization
- Not Copy Protected, no dongle. can be installed on your hard drive



Abacus

**5370 52nd St. SE Grand Rapids MI 49508 - Order Toll Free! 800-451-4319**

# New Software

*All Abacus software runs on the Amiga 500 Amiga 1000 or Amiga 2000. Each package is fully compatible with our other products in the Amiga line*

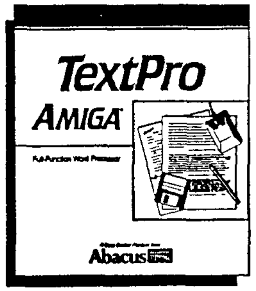
The *Ideal* AMIGA wordprocessor

## TextPro AMIGA

**TextPro AMIGA** upholds the true spirit of the AMIGA: it's powerful, it has a surprising number of "extra" features, but it's also very easy to use. **TextPro AMIGA**—the Ideal AMIGA word processor that proves just how easy word processing can be. You can write your first documents immediately, with a minimum of learning—without even reading the manual. But **TextPro AMIGA** is much more than a beginner's package. Ultra-fast onscreen formatting, graphic merge capabilities, automatic hyphenation and many more features make **TextPro AMIGA** ideal for the professional user as well. **TextPro AMIGA** features:

- High-speed text input and editing
- Functions accessible through menus or shortcut keys
- Fast onscreen formatting
- Automatic hyphenation
- Versatile function key assignment
- Save any section of an AMIGA screen & print as text
- Loading and saving through the RS-232 interface
- Multiple tab settings
- Accepts IFF format graphics in texts
- Extremely flexible printer adaptations. Printer drivers for most popular dot-matrix printers included
- Includes thorough manual
- Not copy protected

**TextPro AMIGA** sets a new standard for word processing packages in its price range. So easy to use and modestly priced that any AMIGA owner can use it—so packed with advanced features, you can't pass it up.



**Suggested retail price: $79.95**

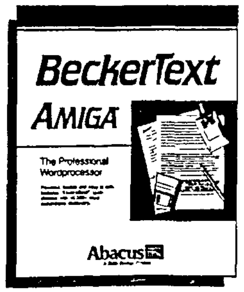
More than word processing...

## BeckerText AMIGA

This is one program for <u>serious</u> AMIGA owners. **BeckerText Amiga** is more than a word processor. It has all the features of **TextPro AMIGA**, but it also has features that you might not expect:

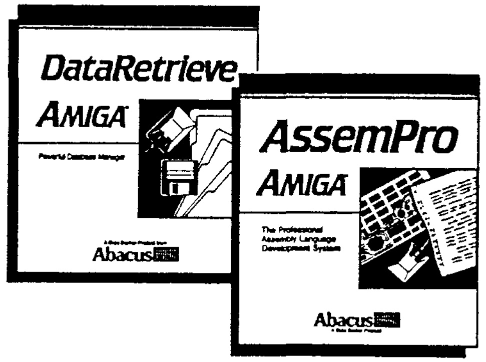
- Fast WYSIWYG formatting
- Calculations within a text—like having a spreadsheet program anytime you want it
- Templates for calculations in columns
- Line spacing options
- Auto-hyphenation and Auto-indexing
- Multiple-column printing, up to 5 columns on a single page
- Online dictionary checks spelling in text as it's written
- Spell checker for interactive proofing of documents
- Up to 999 characters per line (with scrolling)
- Many more features for the professional

**BeckerText AMIGA** is a vital addition for C programmers—it's an extremely flexible C editor. Whether you're deleting, adding or duplicating a block of C source-code, **BeckerText AMIGA** does it all, automatically. And the online dictionary acts as a C syntax checker and finds syntax errors in a flash.



**BeckerText AMIGA.** When you need more from your word processor than just word processing.

Suggested retail price: **$150.00**



# How to Order

**Abacus** 5370 52nd Street SE Grand Rapids, MI 49508

All of our Amiga products–application and language software, and our Amiga Reference Library–are available at more than 2000 dealers in the U.S. and Canada. To find out the location of the Abacus dealer nearest you, call:

**Toll Free 1-800-451-4319**

8:30 am-8:00 pm Eastern Standard Time

Or order from Abacus directly by phone with your credit card. We accept Mastercard, Visa and American Express.

Every one of our software packages is backed by the **Abacus 30-Day Guarantee**—if for any reason you're not satisfied by the software purchased directly from us, simply return the product for a full refund of the purchase price.

**Order Blank**

**Send your completed order blank to:**

Abacus Software, Inc.
5370 52nd St. S.E.
Grand Rapids, MI 49508

**Your order will be shipped within 24 hours of our receiving it**

Name: ____________________

Address: ____________________

City __________ State ______ Zip ______ Country ______

Phone: / ____________________

| Qty | Name of product | Price |
|---|---|---|
| | | |
| | | |
| | | |
| | **Mich. residents add 4% sales tax** | |
| | **Shipping/Handling charge** (Foreign Orders $12 per item) | |
| | **Check/Money order TOTAL enclosed** | |

Credit Card# ____________________

Expiration date ______

For extra-fast 24-hour shipment service, order by phone with your credit card

VISA MasterCard American Express

# Table of Contents