The AmigaDOS Manual

Bantam Computer Books
Ask your bookseller for the books you have missed

THE AMIGADOS USER'S MANUAL
by Commodore-Amiga, Inc.
THE APPLE //c BOOK
by Bill O'Brien
THE COMMODORE 64 SURVIVAL MANUAL
by Winn L. Rosch
COMMODORE 128 PROGRAMMER'S REFERENCE GUIDE
by Commodore Business Machines, Inc.
EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR APPLE II
by Tim Hartnell
EXPLORING ARTIFICIAL INTELLIGENCE ON YOUR COMMODORE 64
by Tim Hartnell
EXPLORING THE UNIX ENVIRONMENT
by The Waite Group / Irene Pasternack
FRAMEWORK FROM THE GROUND UP
by The Waite Group / Cynthia Spoor and Robert Warren
HOW TO GET THE MOST OUT OF COMPUSERVE, 2d ed.
by Charles Bowen and David Peyton
HOW TO GET THE MOST OUT OF THE SOURCE
by Charles Bowen and David Peyton
THE MACINTOSH
by Bill O'Brien
THE NEW *jr*: A GUIDE TO IBM'S PC*jr*
by Winn L. Rosch
ORCHESTRATING SYMPHONY
by The Waite Group / Dan Shafer
PC-DOS / MS-DOS
User's Guide to the Most Popular Operating System for Personal Computers
by Alan M. Boyd
POWER PAINTING: COMPUTER GRAPHICS ON THE MACINTOSH
by Verne Bauman and Ronald Kidd / illustrated by Gasper Vaccaro
SMARTER TELECOMMUNICATIONS
Hands-On Guide to On-Line Computer Services
by Charles Bowen and Stewart Schneider
SWING WITH JAZZ:
Lotus Jazz on the Macintosh
by Datatech Publications Corp. / Michael McCarty
USER'S GUIDE TO THE AT&T PC 6300 PERSONAL COMPUTER
by David B. Peatroy, Ricardo A. Anzaldua, H. A. Wohlwend,
and Datatech Publications Corp.

The AmigaDOS Manual

Commodore-Amiga, Inc.

BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

AMIGADOS MANUAL
A Bantam Book / February 1986

Cover design by J. Caroff Associates

ISBN 0-553-34294-0

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

H 0 9 8 7 6 5 4 3 2 1

Contents

Preface

This book, *The AmigaDOS Manual,* is a combination of three separate publications:

The AmigaDOS User's Manual
The AmigaDOS Developer's Manual
The AmigaDOS Technical Reference Manual

The *User's Manual* contains information of interest to every Amiga user. There are many more commands that AmigaDOS understands than are accessible from the Workbench. If a user uses Preferences to turn on the CLI, these new commands become accessible.

The *Developer's Manual* describes how to use AmigaDOS from within a program rather than from a command line interface. It also fully documents the Amiga Macro Assembler and Linker. (Note that the Amiga Macro Assembler is available as a separate product.)

The *Technical Reference Manual* describes the data structures that AmigaDOS uses internally. It includes descriptions of how DOS disk data is stored, and the format of the "object-files" that AmigaDOS uses. A developer or expert user would find the information in this technical section to be very useful.

Together these three publications, in this single volume, comprise the essential guide to AmigaDOS.

AmigaDOS User's Manual

Introduction

This manual describes the AmigaDOS and its commands. The Command Line Interpreter (CLI) reads AmigaDOS commands typed into a CLI window and translates them into actions performed by the computer. In this sense, the CLI is similar to more "traditional" computer interfaces: you type in commands and the interface displays text in return.

Because the Workbench interface is sufficient and friendly for most users the Workbench diskettes are shipped with the CLI interface "disabled". To use the commands in this manual you must "enable" the CLI interface. This puts a new icon, labeled "CLI" on your Workbench. When you have selected and opened this icon, a CLI window becomes available, and you can use it to issue text commands directly to AmigaDOS.

How to Enable the Command Line Interface

Boot your computer using the Kickstart and Workbench diskettes. Open the diskette icon. Open the "Preferences" tool. Near the left-hand side of the screen, about two-thirds of the way down you will notice "CLI" with a button for "ON" and a button for "OFF". Select the "ON" button. Select "Save" (lower right part of the Preferences screen) to leave Preferences.

How to Open a CLI Window

To use the CLI commands, you open a CLI window. Open the "System" drawer. The CLI icon (a cube containing "1>") should now be visible. Open it.

Using the CLI

To use the CLI interface select the CLI window and type the desired CLI commands. The CLI window(s) may be sized and moved just like most others. To close the CLI window, type "ENDCLI".

Workbench and CLI, Their Relationship and Differences

Type "DIR" to display a list of files (and directories) in the current disk directory. This is a list of files that makes up your Workbench. You may notice that there are more files in this directory than there are icons on the Workbench. Workbench only displays file "X" if that file has an associated "X.info" file. Workbench uses the ".info" file to manipulate the icon.

For example, the diskcopy program has two files. The file "Diskcopy" contains the program and "Diskcopy.info" contains the Workbench information about it. In the case of painting data files like "mount.pic" the file "mount.pic.-info" contains icon information and the name of the program (default) that should process it (GraphiCraft). In this case, when the user "opens" the data file (mount.pic) Workbench runs the program and passes the data file name (mount.pic) to it.

AmigaDOS subdirectories correspond to Workbench drawers. Random access block devices such as disks (DF0:) correspond to the diskette icons you have seen.

Not all programs or commands can be run under both Workbench and the CLI environment. None of the CLI commands described in Chapter 2 of this manual can be run from Workbench. For example, there are two separate Diskcopy commands. The one in the :c/ directory is run from AmigaDOS (CLI). The one in the system directory (drawer) is run from Workbench.

Chapter 1

Introducing AmigaDOS

This chapter provides a general overview of the AmigaDOS operating system, including descriptions of terminal handling, the directory structure, and command use. At the end of the chapter, you'll find a simple example session with AmigaDOS.

1.1 Chapter Overview

AmigaDOS is a **multi-processing** operating system designed for the Amiga. Although you can use it as a multi-user system, you normally run AmigaDOS for a single user. The multi-processing facility lets many jobs take place simul-

taneously. You can also use the multi-processing facility to suspend one job while you run another.

Each AmigaDOS **process** represents a particular process of the operating system—for example, the filing system. Only one process is running at a time, while other processes are either waiting for something to happen or have been interrupted and are waiting to be resumed. Each process has a **priority** associated with it, and the process with the highest priority that is free to run does so. Processes of lower priority run only when those of higher priority are waiting for some reason—for example, waiting for information to arrive from the disk.

The standard AmigaDOS system uses a number of processes that are not available to you, for example, the process that handles the serial line. These processes are known as private processes. Other private processes handle the terminal and the filing system on a disk drive. If the hardware configuration contains more than one disk drive, there is a process for each drive.

AmigaDOS provides a process that you can use, called a **Command Line Interface** or **CLI.** There may be several CLI processes running simultaneously, numbered from 1 onward. The CLI processes read **commands** and then execute them. All commands and user programs will run under any CLI. To make additional CLI processes, you use the NEWCLI or RUN commands. To remove a CLI process use the ENDCLI command. (You can find a full description of these commands in Chapter 2 of this manual.)

1.2 Terminal Handling

You can direct information that you enter at the terminal to a Command Line Interface (CLI) that tells AmigaDOS to load a program, or you can direct the information to a program running under that CLI. In either case, a **terminal** (or **console**) **handler** processes input and output. This terminal handler also performs local line editing and certain other functions. You can type ahead as many as 255 characters—the maximum line length.

To correct mistakes, you press the BACKSPACE key. This erases the last character you typed. To rub out an entire line, hold down the CTRL key while you press X. This **control combination** is referred to from this point on in the manual as CTRL-X.

If you type anything, AmigaDOS waits until you have finished typing before displaying any other output. Because AmigaDOS waits for you to finish, you can type ahead without your input and output becoming intermixed. AmigaDOS recognizes that you have finished a line when you press the RETURN key. You can also tell AmigaDOS that you have finished with a line by cancelling it. To cancel a line, you can either press CTRL-X or press BACKSPACE until all the characters on the line have been erased. Once AmigaDOS is satisfied that you

have finished, it starts to display the output that it was holding back. If you wish to stop the output so that you can read it, simply type any character (pressing the space bar is the easiest), and the output stops. To restart output, press BACKSPACE, CTRL-X, or RETURN. Pressing RETURN causes AmigaDOS to try to execute the command line typed after the current program exits.

AmigaDOS recognizes CTRL-\ as an end-of-file indicator. In certain circumstances, you use this combination to terminate an input file. (For a circumstance when you would use CTRL-\, see Section 1.3.6.)

If you find that strange characters appear on the screen when you type anything on the keyboard, you have probably pressed CTRL-O by mistake. AmigaDOS recognizes this control combination as an instruction to the console device (CON:) to display the alternative character set. To undo this condition, you press CTRL-N. Any further characters should then appear as normal. On the other hand, you could press ESC-C to clear the screen and display normal text.

Note: Any input through the console device CON: ignores function keys and cursor keys. If you want to receive these keys, you should use RAW:. (For a description of RAW:, see Section 1.3.6, "Understanding Device Names," later in this chapter.)

Finally, AmigaDOS recognizes all commands and **arguments** typed in either upper or lower case. AmigaDOS displays a **filename** with the characters in the case used when it was created, but finds the file no matter what combination of cases you use to specify the filename.

1.3 Using the Filing System

This section describes the AmigaDOS filing system. In particular, it explains how to name, organize, and recall your files.

A file is the smallest named object used by AmigaDOS. The simplest identification of a file is by its filename, discussed below in Section 1.3.1. However, it may be necessary to identify a file more fully. Such an identification may include the device or volume name, and/or directory name(s) as well as the filename. These will be discussed in following sections.

1.3.1 Naming Files

AmigaDOS holds information on disks in a number of **files**, named so that you can identify and recall them. The filing system allows filenames to have up to thirty characters, where the characters may be any printing character except slash (/) and colon (:). This means that you can include space( ), equals (=), plus (+), and double quote ("), all special characters recognized by the CLI, within a filename. However, if you use these special characters, you must

enclose the entire filename with double quotes. To introduce a double quote character within a filename, you must type an asterisk (*) immediately before that character. In addition, to introduce an asterisk, you must type another asterisk. This means that a file named

A*B = C"

should be typed as follows:

"A**B = C*" "

in order for the CLI to accept it.

Note: This use of the asterisk is in contrast to many other operating systems where it is used as a universal **wild card.** An asterisk by itself in AmigaDOS represents the keyboard and the current window. For example,

```
COPY filename to *
```

copies the filename to the screen.

Avoid spaces before or after filenames because they may cause confusion.

1.3.2 Using Directories

The filing system also allows the use of **directories** as a way to group files together into logical units. For example, you may use two different directories to separate program source from program documentation, or to keep files belonging to one person distinct from those belonging to another.

Each file on a disk must belong to a directory. An empty disk contains one directory, called the **root directory**. If you create a file on an empty disk, then that file belongs to this root directory. However, directories may themselves contain further directories. Each directory may therefore contain files, or yet more directories, or a mixture of both. Any filename is unique only within the directory it belongs to, so that the file "fred" in the directory "bill" is a completely different file from the one called "fred" in the directory "mary".

This filing structure means that two people sharing a disk do not have to worry about accidentally overwriting files created by someone else, as long as they always create files in their own directories.

> **WARNING:** When you create a file with a filename that already exists, AmigaDOS deletes the previous contents of that file. No message to that effect appears on the screen.

You can also use this directory structure to organize information on the disk, keeping different sorts of files in different directories.

An example might help to clarify this. Consider a disk that contains two directories, called "bill" and "mary." The directory "bill" contains two files, called "text" and "letter". The directory "mary" contains a file called "data" and two directories called "letter" and "invoice". These sub-directories each contain a file called "jun18". Figure 1-A represents this structure as follows:

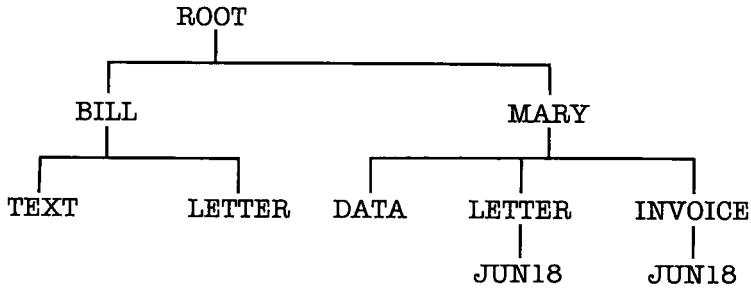


Figure 1-A: Using Directory Structure

Note: The directory "bill" has a file called "letter," while the directory "mary" contains a directory called "letter". However, there is no confusion here because both files are in different directories. There is no limit to the depth that you can "nest" directories.

To specify a file fully, you must include the directory that owns it, the directory owning that directory, and so on. To specify a file, you give the names of all the directories on the path to the desired file. To separate each directory name from the next directory or filename, you type a following slash (/). Thus, the full specification of the data files on the disk shown in Figure 1-A above is as follows:

```
bill/text
bill/letter
mary/data
mary/letter/jun18
mary/invoice/jun18
```

1.3.3 Setting the Current Directory

A full file description can get extremely cumbersome to type, so the filing system maintains the idea of a **current directory.** The filing system searches for files in this current directory. To specify the current directory, you use the CD (Current Directory) command. If you have set "mary" as your current directory, then the following names would be sufficient to specify the files in that directory:

```
data
letter/jun18
invoice/jun18
```

You can set any directory as the current directory. To specify any files within that directory, simply type the name of the file. To specify files within sub-directories, you need to type the names of the directories on the path from the current directory specified.

All the files on the disk are still available even though you've set up a current directory. To instruct AmigaDOS to search through the directories from the root directory, you type a colon (:) at the beginning of the file description. Thus, when your file description has the current directory set to "mary", you can also obtain the file "data" by typing the description ":mary/data". Using the current directory method simply saves typing, because all you have to do is specify the filename "data".

To obtain the other files on the disk, first type ":bill/text" and ":bill/letter" respectively. Another way might be to CD or type / before a filename. Slash does not mean "root" as in some systems, but refers to the directory above the current directory. AmigaDOS allows multiple slashes. Each slash refers to the level above. So a Unix (TM) ../ is a / in AmigaDOS. Similarly, an MS-DOS™ ..\ is a / in AmigaDOS. Thus, if the current directory is ":mary/letter", you may specify the file ":mary/invoice/jun18" as "/invoice/jun18". To refer to the files in ":bill", you could type:

```
CD :bill
```

or

```
CD //bill
```

Then you could specify any file in "bill" with a single filename. Of course, you could always use the // feature to refer directly to a specific file. For example,

```
TYPE //bill/letter
```

displays the file without your first setting "bill" as the current directory. To go straight to the root level, always type a colon (:) followed by a directory name. If you use slashes, you must know the exact number of levels back desired.

1.3.4 Setting the Current Device

Finally, you may have many disk drives available. Each disk device has a name, in the form DFn (for example, DF1), where the "n" refers to the number of the device. (Currently, AmigaDOS accepts the device names DF0 to DF3.)

Each individual disk is also associated with a unique name, known as a volume name (see below for more details).

In addition, the logical device SYS: is assigned to the disk you started the system up from. You can use this name in place of a disk device name (like DF0:).

The current directory is also associated with a **current drive**, the drive where you may find the directory. As you know, prefacing a file description with a colon serves to identify the root directory of the current drive. However, to give the root directory of a specific drive, you precede the colon with the drive name. Thus, you have yet another way of specifying the file "data" in directory "mary", that is "DF1:mary/data". This assumes that you have inserted the disk into drive DF1. So, to reference a file on the drive DF0 called "project-report" in directory "peter", you would type "DF0:peter/project-report", no matter which directory you had set as the current one.

Note: When you refer to a disk drive or any other device, on its own or with a directory name, you should always type the colon, for example, DF1:.

Figure 1-B illustrates the structure of a file description. Figure 1-C gives some examples of valid file descriptions.

Left of the:	**Right of the:**	**Right of a/**
Device name	Directory name	Subdirectory name
or	or	or
Volume name	Filename	Filename

Figure 1-B: The Structure of a File Description

```
SYS:commands
DF0:bill
DF1:mary/letter
DF2:mary/letter/jun18
DOC:report/section1/figures
FONTS:silly-font
C:cls
```

Figure 1-C: Examples of File Descriptions

To gain access to a file on a particular disk, you can type its unique name, which is known as the disk's **volume name**, instead of the device name. For instance, if the file is on the disk "MCC", you can specify the same file by typing the name "MCC:peter/project-report". You can use the volume name to refer to a disk regardless of the drive it is in. You assign a volume name to a disk when you format it (for further details, see "FORMAT" in Chapter 2, "Commands," later in this manual).

A device name, unlike a volume name, is not really part of the name. For example, AmigaDOS can read a file you created on DF0: from another drive, such as DF1:, if you place the disk in that drive, assuming of course that the drives are interchangeable. That is, if you create a file called "bill" on a disk in drive DF0:, the file is known as "DF0:bill". If you then move the disk to drive DF1:, AmigaDOS can still read the file, which is then known as "DF1:bill".

1.3.5 Attaching a Filenote

Although a filename can give some information about its contents, it is often necessary to look in the file itself to find out more. AmigaDOS provides a simple solution to this problem. You can use the command called FILENOTE to attach an associated comment. You can make up a comment of up to 80 characters (you must enclose comments containing spaces in double quotes). Anything can be put in a file comment: the day of the file's creation, whether or not a bug has been fixed, the version number of a program, and anything else that may help to identify it.

You must associate a comment with a particular file—not all files have them. To attach comments, you use the FILENOTE command. If you create a new file, it will not have a comment. Even if the new file is a copy of a file that has a comment, the comment is not copied to the new file. However, any comment attached to a file which is overwritten is retained. To write a program to copy a file and its comment, you'll have to do some extra work to copy the comment. For details, see Chapter 2 of the *AmigaDOS Developer's Manual.*

When you rename a file, the comment associated with it doesn't change. The RENAME command only changes the name of a file. The file's contents and comment remain the same regardless of the name change. For more details, see LIST and FILENOTE in Chapter 2 of this manual.

1.3.6 Understanding Device Names

Devices have names so that you can refer to them by name. Disk names such as DF0: are examples of **device names.** Note that you may refer to device names, like filenames, using either upper or lower case. For disks, you follow the device name by a filename because AmigaDOS supports files on these devices. Furthermore, the filename can include directories because AmigaDOS also supports directories.

You can also create files in memory with the device called RAM:. RAM: implements a filing system in memory that supports any of the normal filing system commands.

Note: RAM: requires the library l/ram-handler to be on the disk.

Once the device RAM: exists, you can, for instance, create a directory to copy all the commands into memory. To do this, type the following commands:

```
MAKEDIR ram:c
COPY sys:c TO ram:c
ASSIGN C: RAM:C
```

You could then look at the output with DIR RAM:. It would include the directory "c" (DIR lists this as c(dir).) This would make loading commands very quick but would leave little room in memory for anything else. Any files in the RAM: device are lost when you reset the machine.

AmigaDOS also provides a number of other devices that you can use instead of a reference to a disk file. The following paragraphs describe these devices including NIL:, SER:, PAR:, PRT:, CON:, and RAW:. In particular, the device NIL: is a dummy device. AmigaDOS simply throws away output written to NIL:. While reading from NIL:, AmigaDOS gives an immediate "end-of-file" indication. For example, you would type the following

```
EDIT abc TO nil:
```

to use the editor to browse through a file, while AmigaDOS throws away the edited output.

You use the device called SER: to refer to any device connected to the serial line (often a printer). Thus, you would type the following command sequence:

```
COPY xyz TO ser:
```

to instruct AmigaDOS to send the contents of the file "xyz" down the serial line. Note that the serial device only copies in multiples of 400 bytes at a time. Copying with SER: can therefore appear granular.

The device PAR: refers to the parallel port in the same way.

AmigaDOS also provides the device PRT: (for PRinTer). PRT: is the printer you chose in the "preferences" program. In this program, you can define your printer to be connected through either the serial or parallel port. Thus, the command sequence

```
COPY xyz TO PRT:
```

prints the file "xyz," no matter how the printer is connected.

PRT: translates every linefeed character in a file to carriage return plus linefeed. Some printers, however, require files without translation. To send a file with the linefeeds as just linefeeds, you use PRT:RAW instead of PRT:.

AmigaDOS supports multiple windows. To make a new window, you can specify the device CON:. The format for CON: is as follows:

```
CON:x/y/width/height/[title]
```

where "x" and "y" are coordinates, "width" and "height" are integers describing the width and height of the new window, and "title", which is optional, is a string. The title appears on the window's title bar. You must include all the slashes (/), including the last one. Your title can include up to thirty characters (including spaces). If the title has spaces, you must enclose the whole description ii double quotes (") as shown in the following example:

```
"CON:20/10/300/100/my window"
```

There is another window device called RAW:, but it is of little use to the general user. (See Chapter 2 of the *AmigaDOS Developer's Manual* in this book for further details.) You can use RAW: to create a raw window device similar to CON:. However, unlike CON:, RAW: does no character translation and does not allow you to change the contents of a line. That is to say, RAW: accepts input and returns output in exactly the same form that it was originally typed. This means characters are sent to a program immediately without letting you erase anything with the BACKSPACE key. You usually use RAW: from a program where you might want to do input and output without character translation.

> **WARNING:** RAW: is intended for the advanced user. Do not use RAW: experimentally.

There is one special name, which is * (asterisk). You use this to refer to the current window, both for input or for output. You can use the COPY command to copy from one file to another. Using *, you can copy from the current window to another window, for example,

```
COPY * TO CON:20/20/350/150/
```

from the current window to the current window, for example,

```
COPY * TO *
```

or from a file to the current window, for example,

```
COPY bill/letter TO *
```

AmigaDOS finishes copying when it comes to the end of the file. To tell AmigaDOS to stop copying from *, you must give the CTRL-\ combination. Note that * is NOT the universal wild card.

1.3.7 Using Directory Conventions and Logical Devices

In addition to the aforementioned physical devices, AmigaDOS supports a variety of useful **logical devices.** AmigaDOS uses these devices to find the files that your programs require from time to time. (So that your programs can refer to a standard device name regardless of where the file actually is.) All of these "logical devices" may be reassigned by you to reference any directory.

The logical devices described in this section are as follows:

Name	Description	Directory
SYS:	System disk root directory	:
C:	Commands directory	:C
L:	Library directory	:L
s:	Sequence Library	:S
LIBS:	Library for Open Library calls	:LIBS
DEVS:	Device for Open Device calls	:DEVS
FONTS:	Loadable fonts for Open Fonts	:FONTS
	Temporary workspace	:T

Figure 1-D: Logical Devices

Logical device name: SYS:
Typical directory name: My.Boot.Disk:
"SYS" represents the SYStem disk root directory. When you first start up the Amiga system, AmigaDOS assigns SYS: to the root directory name of the disk in DF0:. If, for instance, the disk in drive DF0: has the volume name My.Boot.Disk, then AmigaDOS assigns SYS: to My.Boot.DISK:. After this assignment, any programs that refer to SYS: use that disk's root directory.

Logical device name: C:
Typical directory name: My.Boot.Disk:c
'C' represents the Commands directory. When you type a command to the CLI (DIR <cr>, for example), AmigaDOS first searches for that command in your current directory. If the system cannot find the command in the current directory, it then looks for "C:DIR". So that, if you have assigned "C:" to another directory (for example, "Boot_disk:c"), AmigaDOS reads and executes from "Boot_disk:c/DIR".

Logical device name: L:
Typical directory name: My.Boot.Disk:1
"L" represents the Library directory. This directory keeps the overlays for large commands and nonresident parts of the operating system. For instance, the disk based run-time libraries (Ram-Handler, Port-Handler, Disk-Validator, and so forth) are kept here. AmigaDOS requires this directory to operate.

Logical device name: S:
Typical directory name: My.Boot.Disk:s
"S" represents the Sequence library. Sequence files contain command sequences that the EXECUTE command searches for and uses. EXECUTE first looks for the sequence (or batch) file in your current directory. If EXECUTE cannot find it there, it looks in the directory that you have assigned S: to.

Logical device name: LIBS:
Typical directory name: My.Boot.Disk:LIBS
Open Library function calls look here for the library if it is not already loaded in memory.

Logical device name: DEVS:
Typical directory name: My.Boot.Disk:DEVS
Open Device calls look here for the device if it is not already loaded in memory.

Logical device name: FONTS:
Typical directory name: My.Boot.Disk:FONTS
Open Fonts look here for your loadable fonts if they are not already loaded in memory.

Note: In addition to the above assignable directories, many programs open files in the ":T" directory. As you recall, you find file (or directory) names predicated with a ":" in the root directory. Therefore ":T" is the directory T, within the root, on the current disk. You use this directory to store temporary files. Programs such as editors place their temporary work files, or backup copies of the last file edited, in this directory. If you run out of space on a disk, this is one of the first places you should look for files that are no longer needed.

When the system is first booted, AmigaDOS initially assigns C: to the :C directory. This means that if you boot with a disk that you had formatted by issuing the command:

```
FORMAT DRIVE DF0: NAME "My.Boot.Disk"
```

SYS: is assigned to "My.Boot.Disk". The "logical device" C: is assigned to the C directory on the same disk (that is, My.Boot.Disk:c). Likewise, the following assignments are made

```
C:      My.Boot.Disk:c
L:      My.Boot.Disk:l
S:      My.Boot.Disk:s
LIBS:   My.Boot.Disk:libs
DEVS:   My.Boot.Disk:devs
FONTS:  My.Boot.Disk:fonts
```

If a directory is not present, the corresponding logical device is assigned to the root directory.

If you are so lucky as to have a hard disk (called DH0:) and you want to use the system files on it, you must issue the following commands to the system:

```
ASSIGN SYS:      DH0:
ASSIGN C:        DH0:C
ASSIGN L:        DH0:L
ASSIGN S:        DH0:S
ASSIGN LIBS:     DH0:LIBS
ASSIGN DEVS:     DH0:DEVS
ASSIGN FONTS:    DH0:FONTS
```

Please keep in mind that assignments are global to all CLI processes. Changing an assignment within one window changes it for all windows.

If you want to use your own special font library, type

```
ASSIGN FONTS: "Special font disk:myfonts"
```

If you want your commands to load faster (and you have memory "to burn"), type

```
makedir ram:c
copy sys:c ram:c all
assign c: ram:c
```

This copies all of the normal AmigaDOS commands to the RAM disk and reassigns the commands directory so that the system finds them there.

1.4 Using AmigaDOS Commands

An AmigaDOS command consists of the command name and its arguments, if any. To execute an AmigaDOS command, you type the command name and its arguments after the CLI prompt.

When you type a command name, the command runs as part of the Command Line Interface (CLI). You can type other command names ahead, but AmigaDOS does not execute them until the current command has finished. When a command has finished, the current CLI prompt appears. In this case, the command is running interactively.

The CLI prompt is initially n>, where n is the number of the CLI process. However, it can be changed to something else with the PROMPT command. (For further details on the PROMPT command, see Chapter 2 of this manual.)

WARNING: If you run a command interactively and it fails, AmigaDOS continues to execute the next command you typed anyway. Therefore, it can be dangerous to type many commands ahead. For example, if you type

```
COPY a TO b
DELETE a
```

and the COPY command fails (perhaps because the disk is full), then DELETE executes and you lose your file.

1.4.1 Running Commands in the Background

You can instruct AmigaDOS to run a command, or commands, in the background. To do this, you use the RUN command. This creates a new CLI as a separate process of lower priority. In this case, AmigaDOS executes subsequent command lines at the same time as those that have been RUN. For example, you can examine the contents of your directory at the same time as sending a copy of your text file to the printer. To do this, type

```
RUN TYPE text_file to PRT:
LIST
```

RUN creates a new CLI and carries out your printing while you list your directory files on your original CLI window.

You can ask AmigaDOS to carry out several commands using RUN. RUN takes each command and carries it out in the given order. The line containing commands after RUN is called a command line. To terminate the command line, press RETURN. To extend your command line over several lines, type a plus sign (+) before pressing RETURN on every line except the last. For example,

```
RUN JOIN text_file1 text_file2 AS text_file +
SORT text_file TO sorted_text +
TYPE sorted_text to PRT:
```

1.4.2 Executing Command Files

You can also use the EXECUTE command to execute command lines in a file instead of typing them in directly. The CLI reads the sequence of commands from the file until it finds an error or the end of the file. If it finds an error,

AmigaDOS does not execute subsequent commands on the RUN line or in the file used by EXECUTE, unless you have used the FAILAT command. See Chapter 2 of this manual for details on the FAILAT command. The CLI only gives prompts after executing commands that have run interactively.

1.4.3 Directing Command Input and Output

AmigaDOS provides a way for you to redirect standard input and output. You use the > and < symbols as commands. When you type a command, AmigaDOS usually displays the output from that command on the screen. To tell AmigaDOS to send the output to a file, you can use the > command. To tell AmigaDOS to accept the input to a program from a specified file rather than from the keyboard, you use the < command. The < and > commands act like traffic cops who direct the flow of information. For example, to direct the output from the DATE command and write it to the file named "text_file", you would type the following command line:

```
DATE > text_file
```

See Chapter 2 of the *User's Manual* for a full specification of the < and > symbols.

1.4.4 Interrupting AmigaDOS

AmigaDOS allows you to indicate four levels of attention interrupt with CTRL-C, CTRL-D, CTRL-E, and CTRL-F. To stop the current command from whatever it was doing, press CTRL-C. In some cases, such as EDIT, pressing CTRL-C instructs the command to stop what it was doing and then to return to reading more EDIT commands. To tell the CLI to stop a command sequence initiated by the EXECUTE command as soon as the current command being executed finishes, press CTRL-D. CTRL-E and CTRL-F are only used by certain commands in special cases. See the *Developer's Manual* in this book for details.

Note: It is the programmer's responsibility to detect and respond to these interruption flags. AmigaDOS will not kill a program by itself.

1.4.5 Understanding Command Formats

This section explains the standard format or argument template used by most AmigaDOS commands to specify their arguments. Chapter 2 of this manual includes this argument template in the documentation of each of the commands. The template provides you with a great deal of flexibility in the order and form of the syntax of your commands.

The argument template specifies a list of **keywords** that you may use as

synonyms, so that you type the alternatives after the keyword, and separate them with an =

For example,

```
ABC,WWW,XYZ=ZZZ
```

specifies keywords, ABC, WWW, and XYZ. The user may use keyword ZZZ as an alternative to the keyword XYZ.

These keywords specify the number and form of the arguments that the program expects. The arguments may be optional or required. If you give the arguments, you may specify them in one of two ways:

By position In this case, you provide the arguments in the same order as the keyword list indicates.

By keyword In this case, the order does not matter, and you precede each argument with the relevant keyword.

For example, if the command MYCOMMAND read from one file and wrote to another, the argument template would be:

```
FROM,TO
```

You could use the command specifying the arguments by position:

```
MYCOMMAND input-file output-file
```

or using the keywords:

```
MYCOMMAND FROM input-file TO output-file
MYCOMMAND TO output-file FROM input-file
```

You could also combine the positional and keyword argument specifications, for example, with the following:

```
MYCOMMAND input-file TO output-file
```

where you give the FROM argument by position, and the TO argument by keyword. Note that the following form is incorrect:

```
MYCOMMAND output-file FROM input-file
```

because the command assumes that 'output-file' is the first positional argument (that is, the FROM file).

If the argument is not a single word (that is, surrounded or "delimited" by spaces), then you must enclose it with quotation marks ("). If the argument has the same value as one of the keywords, you must also enclose it with quotation marks. For example, the following:

```
MYCOMMAND "file name" TO "destination"
```

supplies the text "file name" as the FROM argument, and the file name "destination" as the TO argument.

The keywords in these argument lists have certain qualifiers associated with them. These qualifiers are represented by a slash (/) and a specific letter. The meanings of the qualifiers are as follows:

/A The argument is required and may not be omitted.
/K The argument must be given with the keyword and may not be used positionally.
/S The keyword is a switch (that is, a toggle) and takes no argument.

The qualifiers A and K may be combined, so that the template

```
DRIVE/A/K
```

means that you must give the argument and keyword DRIVE.

In some cases, no keywords may be given. For example, the command DELETE simply takes a number of files for AmigaDOS to delete. In this case, you simply omit the keyword value, but the commas normally used to separate the keywords remain in the template. Thus, the template for DELETE, that can take up to ten filenames, is

```
,,,,,,,,,
```

Finally, consider the command TYPE. The argument template is

```
FROM/A,TO,OPT/K
```

which means that you may give the first argument by position or by keyword, but that first argument is required. The second argument (TO) is optional, and you may omit the keyword. The OPT argument is optional, but if it is given, you must provide the keyword. So, the following are all valid forms of the TYPE command:

```
TYPE filename
TYPE FROM filename
TYPE filename TO output-file
TYPE filename output-file
TYPE TO output-file FROM filename OPT n
TYPE filename OPT n
TYPE filename OPT n TO output-file
```

Although this manual lists all the arguments expected by the commands, you can display the argument template by simply typing the name of the command, followed by a space and a question mark (?).

If the arguments you specify do not match the template, most commands simply display the message "Bad args" or "Bad arguments" and stop. You must retype the command name and argument. To display on the screen help on what arguments the command expected, you can always type a question mark (?).

1.5 Restart Validation Process

When you first insert a disk for updating, AmigaDOS creates a process at low priority. This validates the entire structure on the disk. Until the restart process has completed this job, you cannot create files on the disk. It is possible, however, to read files.

When the restart process completes, AmigaDOS checks to see if you have set the system date and time. To set the date and time, you use the DATE command. If you do not specify the system date, AmigaDOS sets the system date to the date and time of the most recently created file on the inserted disk. This ensures that newer versions of files have more recent dates, even though the actual time and date will be incorrect.

If you ask for the date and the time before the validation is complete, AmigaDOS displays the date and time as unset. You can then either wait for the validation to complete or use DATE to enter the correct date and time. Validation should happen at once; otherwise, it should never take longer than one minute.

1.6 Commonly Used Commands: An Example Session

This manual describes the various AmigaDOS commands. The Command Line Interpreter (CLI) reads AmigaDOS commands typed into a CLI window and translates them into actions performed by the computer. In this sense the CLI

is similar to more "traditional" computer interfaces: you type in commands and the interface displays text in return.

Because the Workbench interface is sufficient and friendly for most users, the Workbench diskettes are shipped with the CLI interface "disabled". To use the commands in this manual you must "enable" the CLI interface. This puts a new icon, labeled "CLI" on your Workbench. When you have selected and opened this icon, a CLI window becomes available, and you can use it to issue text commands directly to AmigaDOS.

How to Enable the Command Line Interface

Boot your computer using the Kickstart diskette and a writable copy of your Workbench diskette. Open the Workbench diskette icon. Open the "Preferences" tool. Near the left-hand side of the screen, about two-thirds of the way down you will notice "CLI" with a button for "ON" and a button "OFF". Select the "ON" button. Select "Save" (lower right part of the Preferences screen) to leave Preferences.

How to Make a New CLI Window

To use the CLI commands, you open a CLI window. Open the "System" drawer. The CLI icon (a cube containing "1>") should now be visible. Open it.

Using the CLI

To use the CLI interface, select the CLI window and type the desired CLI commands (described within this manual). The CLI window(s) may be sized and moved just like many others. To close the CLI window, type "ENDCLI".

Workbench and CLI: Their Relationships and Differences

Type "DIR" to display a list of files (and directories) in the current disk directory. This is a list of files that makes up your Workbench. You may notice that there are many more files in this directory than there are icons on the Workbench. The reason for this is that Workbench will only display file "X" if it has an associated "X.info" file. In fact the ".info" (pronounced "dot info") file contains all of the icon display information.

For example, the diskcopy program has two files associated with it. The file "Diskcopy" contains the program and "Diskcopy.info" contains the Workbench information about it. In the case of painting data files like "mount.pic", the file "mount.pic.info" contains icon information and the name of the program (default) that should process it (GraphiCraft). In this case, when the

user "opens" the data file (mount.pic.info), Workbench runs the program and passes the data file name (mount.pic) to it.

AmigaDOS sub-directories correspond to Workbench drawers. Random access block devices such as disks (DF0:) correspond to the diskette icons you have seen.

Not all programs or commands can be run under both Workbench and the CLI environment. None of the CLI commands described in Chapter 2 of the *AmigaDOS User's Manual* can be run from Workbench. For example, there are two separate Diskcopy commands. The one in the :c/ directory works with AmigaDOS (CLI). The one in the system directory (drawer) works with Workbench.

An Introduction to Some of the AmigaDOS Commands

Although all of the commands that are available through the CLI are explained in detail in the reference part of the *AmigaDOS User's Manual,* we have found that most users will use very few of the advanced options. Therefore we have provided a summary here showing various commands in their most common form.

The commands summarized below (along with the actual AmigaDOS command name) ask AmigaDOS to do such commands as

- Copy a diskette (DISKCOPY)
- Format a new diskette (FORMAT)
- Make a formatted diskette bootable;
  create a CLI disk (INSTALL)
- Relabel a diskette (RELABEL)
- Look at the directory of a diskette (DIR)
- Get information about files (LIST)
- Prevent a file from accidental deletion (PROTECT)
- Get Information about a file system (INFO)
- Change a current directory (CD)
- Set the date and time (DATE)
- Redirect the output of a command (>)
- Type a text file to the screen (TYPE)
- Rename a file (RENAME)
- Delete a file (DELETE)
- Create a new directory (MAKEDIR)
- Copy files on a dual-drive system (COPY)
- Copy files on a single-drive system (COPY)
- Find files on a diskette (DIR OPT A)
- Do something automatically at boot time (using Startup-Sequence)

- Tell AmigaDOS where to look for certain things (ASSIGN)
- Open a new CLI window (NEWCLI)
- Close an existing CLI window (ENDCLI)

All of the command sequences below assume that you have started your system with a CLI disk rather than a Workbench disk, or that you have turned on the CLI using the preferences tool and have entered the CLI by that path. The sequence for turning on the CLI is provided earlier in this manual.

For a New User

For a new user, we suggest that you read and try each of these items in sequence. Each command that is shown below leaves a test disk in a known state so that the command that immediately follows will work exactly as shown. Later, when you are more familiar with the system, the paragraph titles shown below will serve to refresh your memory.

How to Begin

Before you begin this section, be sure you have two blank, double-sided diskettes, and either your Workbench disk or your CLI disk. Before you begin, write-protect your master diskette, and write-enable the blank diskettes. Most of the commands given below assume that you have a single-drive system; however, for convenience of those with dual-drive systems, the dual-drive version of the command is occasionally given.

Commands that instruct AmigaDOS to execute are shown in the following sections, indented from the left margin. After typing each command, press the RETURN key to return control to AmigaDOS. Although the commands are all shown in capital letters, this is simply to distinguish them from the rest of the text. AmigaDOS will accept the commands in lower case as well as upper case.

In the sections that follow, the notations "df0:" and "drive 0" refer to the disk drive that is built into the Amiga. The notation "df1:" refers to the first external 3½-inch disk drive.

You will occasionally see a semicolon on a command line that you are told to type. What follows the semicolon is treated as a comment by AmigaDOS. Since AmigaDOS ignores the rest of the line, you don't need to type the comment along with the command. It is for your information only.

For most commands, you can get a very limited form of help by typing the command name, followed by a question mark (?) and pressing RETURN. It shows you the "template" of a command, containing the sequence of parameters it expects and the keywords it recognizes.

Copying a Disk

You can use this sequence to back up your system master disk or any other disk.

For a 1 disk system

```
DISKCOPY FROM df0: TO df0:
```

For a 2 disk system

```
DISKCOPY FROM df0: TO df1:
```

Follow the instructions as they appear. For a single drive system, you'll be instructed to insert the master (FROM) disk. Then, as the copying progresses, AmigaDOS asks you to insert the copy (TO) disk, swapping master and copy in and out until all of the diskette has been duplicated. For a two disk system, you'll be instructed to put the master diskette into drive df0: (the built-in drive) and the copy diskette onto which to copy into df1: (the first external drive).

Remove your master diskette (either Workbench or CLI disk) and put your master diskette in a safe place. Leave the copy write-enabled so that you can store information on it. Insert the copy you have just made into the built-in drive and reboot your system from the copy. (See Introduction To Amiga for the reboot process).

After the reboot, reenter the CLI mode again. If you boot with a CLI disk, the reboot enters the CLI automatically. If you are using a Workbench disk, you must open the CLI icon in the system drawer of the Workbench.

Formatting a Disk

To try this command, your Workbench or CLI diskette copy should be in drive 0, and you should have a blank diskette available.

Sometimes rather than simply copy a disk, you'll want to prepare a data disk for your system. Then later you can copy selected files to this data disk. Format your second blank disk by using the FORMAT command:

```
FORMAT DRIVE df0: NAME "AnyName"
```

Follow the instructions. You can format diskettes in either drive 0 (df0:, built in to your Amiga) or an external drive.

After the format is completed, wait for the disk activity light to go off and

remove the freshly formatted diskette. Reinsert your Workbench or CLI diskette. The formatted diskette can now be used to hold data files. It is not bootable, however.

Making a Disk Bootable

To try this command, your Workbench or CLI diskette copy should be in drive 0, and you should have your freshly formatted disk available.

There are several different ways to create a CLI diskette. Two of these ways are shown below.

A bootable disk is one that you can use to start up your Amiga following the Kickstart process. You can change a formatted disk into a CLI disk by typing the command:

```
INSTALL ?
```

Note: to use this command on a single drive system, you MUST use the question mark! Otherwise AmigaDOS will try to do the install on the disk currently in drive 0.

AmigaDOS responds:

```
DRIVE/A
```

Remove your Workbench diskette copy and insert the formatted disk. Then type:

```
df0:
```

and press RETURN. AmigaDOS copies boot sectors to the diskette. Now, if you wait until the disk activity light goes out, you can then perform a full reset (CTRL-Amiga-Amiga). When the system reboots, you will go directly into the CLI rather than into the Workbench.

Your formatted diskette now contains a CLI and nothing else. This means that although you see the interpreter, it can't perform any of the commands shown in this section. A CLI needs several files before its commands can be performed. All of the command files are located in the C directory of your master diskette.

The second way to produce a CLI disk gives you a more useful disk in that it leaves the CLI command directories intact. Here is a step-by-step process to change a writable copy of a Workbench diskette into a CLI diskette:

1. Copy your Workbench diskette.
2. Open the CLI as described above.
3. Click the selection button on the CLI window and type the command:

```
RENAME FROM s/startup-sequence TO s/NO-startup-sequence
```

Now if you wait for the disk activity light to go off and perform a full reset, your Workbench diskette copy will have become a CLI. To restore the Workbench, perform the rename again, but with the name sequence reversed. You see, if AmigaDOS can't find a file with the exact name "startup-sequence" in the "s" directory, it will enter command mode and wait for you to type a command.

Relabeling a Disk

Before you try this command, your Workbench or CLI diskette copy should be in drive 0.

If, after either copying or formatting a diskette, you are not satisfied with the volume name you have given it, you can change the name of the volume by using the RELABEL command:

```
relabel AnyName: DifferentName
```

In this example, we have referred to the diskette we just formatted by its volume name. You will be asked to insert volume AnyName into any disk drive so that RELABEL can relabel it.

After this command completes, remove the diskette and reinsert your Workbench or CLI diskette. The diskette you removed now has the new name.

Looking at the Directory

Before you try this command, your Workbench or CLI diskette copy should be in drive 0.

You look at the contents of a diskette with the command:

```
DIR or DIR df0:
```

This form lists the contents of your current directory. You can list the contents of a different directory by specifying the pathname for that directory. For example, the command:

```
DIR df0:c or DIR c
```

lists the contents of the c(dir) on drive df0. Directories are equivalent to the drawers you see when the Workbench screen is visible.

You can look at the directory of a different disk unit, if you have one, by specifying its name. For example:

```
DIR df1:
```

lists the contents of a diskette inserted in drive 1 (the first external drive if you have one attached).

You can even look at the directory of a diskette that isn't currently in the drive by specifying its volume name. For example, the contents of that freshly formatted diskette whose name we changed can be displayed by the command:

```
DIR DifferentName:
```

AmigaDOS will ask you to insert diskette DifferentName into the drive so that DIR can read it and report the contents of the directory. Don't do it yet, however, because there are no files present for DIR to read. We'll add some files later.

Using the LIST Command

To try this command, your Workbench or CLI diskette copy should be in drive 0.

The DIR command tells you the names of files that are in your directory. The LIST command provides additional information about those files. Type the command:

```
LIST or LIST df0:
```

AmigaDOS provides information about all files in the current directory, including how large each file is, whether it may or may not be deleted, whether it is a file or a directory, and the date and time of its creation.

If you specify the name of a directory with LIST, it lists information about the files within that directory:

```
LIST c
```

The "rwed" are called protection flags, for read, write, execute, and delete. When each flag is set, using the PROTECT command, a file is supposed to be readable, writable, executable, or deleteable. As of the current release, AmigaDOS only pays attention to the delete-flag. If the "d" doesn't show up in the "rwed" column for a filename, AmigaDOS won't delete that file during a DELETE command.

Using the Protect Command

To try this command, your Workbench or CLI diskette copy should be in drive 0.

This command protects (or unprotects) a file from being deleted accidentally. Try the command:

```
DATE > myfile
PROTECT myfile
LIST myfile
```

You will see that all of the protect-flags have been set to "——". Now if you try:

```
DELETE myfile
```

AmigaDOS responds:

```
"Not Deleted - file is protected from deletion"
```

To reenable deletion of the file:

```
PROTECT myfile d or PROTECT myfile rwed
```

Getting Information About the File System

Your Workbench or CLI diskette copy should still be in drive 0. Type the command:

```
INFO
```

It tells you how much space is used and how much is free on your diskettes, whether they are read-only or read-write, and the name of the volume. You can make more space on the diskette by deleting files. You can change the name of the volume by using the RELABEL command.

If you want to get information about a disk that isn't in your single-drive at the moment, issue the command as:

```
INFO ?
```

AmigaDOS responds:

```
none:
```

AmigaDOS has loaded the INFO command from your CLI disk and shows you the template for the command. The response "none:" says that you don't have to type anything other than a RETURN key to have it perform the command. Remove your CLI disk and insert the disk on which you want INFO to operate. Wait for the disk activity light to go on and off. Then press RETURN. AmigaDOS gives you INFO about this other disk. This works for DIR as well as INFO.

Changing Your Current Directory

Until now, we have only stayed at the "root" or topmost hierarchical level of the diskette directory. You will find more information about the directory tree structure in section 1.3 of this manual. To see the level at which you are currently positioned in your directory tree, you use the command:

```
CD
```

To change to a different current directory, you tell the system which directory is to become the current one. For example, when you did a "dir" command on df0: the CLI diskette you saw an entry c(dir). If you want to make this directory the current one, you issue the command:

```
CD C or CD df0:c
```

Now when you issue the command DIR, it shows the contents of this level of the filing system. The command CD (alone) shows you the name of your current directory. You go up to the root directory (the top level) by specifying:

```
CD:
```

on the current volume (if you refer to your diskettes by volume name) or

```
CD df0:
```

on the built-in drive.

Setting the Date and Time

You can set the AmigaDOS clock by using the DATE command:

```
DATE 12:00:00 12-oct-85
```

Now the system clock counts up from this date and time.

Redirecting the Output of a Command

Before you try this command, your Workbench or CLI diskette should be in drive 0.

Normally the output of all commands goes to the monitor screen. You can change where the system puts the output by using the redirect command ">". The forward arrow symbol means send the output toward this output file name. Here's an example:

```
DATE > datefile
```

Execute the command so that you can use the datefile described below. This command creates (or overwrites) a file named "datefile" in your current directory.

Or, just to have something on that formatted diskette named DifferentName, type the following:

```
DATE > DifferentName:datefile
```

AmigaDOS prompts you to insert the volume with that name. After the disk activity light goes out, remove DifferentName and reinsert your CLI or Workbench diskette. Now issue the command:

```
DIR DifferentName:
```

Again you are prompted to insert DifferentName into any drive. AmigaDOS lists the directory of this diskette, which now contains a file named datefile.

Replace your CLI or Workbench diskette in the drive.

Typing a Textfile to the Screen

You can see the contents of a textfile by using the TYPE command:

```
TYPE datefile
```

This command will display whatever you have in the specified file. If you wish to stop the output momentarily to read something on the screen, press the space bar. To restart it press the BACKSP key. If you wish to end the TYPE command, hold down the CTRL key, and press the C key.

If you wish to verify that another diskette also has the datefile contents on it, you can perform the command:

```
TYPE DifferentName:datefile
```

Changing the Name of a File

Before you try this command, your Workbench or CLI diskette copy should be in drive 0.

You can change the name of a file by using the RENAME command:

```
RENAME FROM datefile TO newname
```

or

```
RENAME datefile newname
```

Now use TYPE to verify that the new name refers to the same contents.

```
TYPE newname
```

Notice that the alternate form of the command doesn't require that you use the FROM and TO. Most of the AmigaDOS commands have an alternate form, abbreviated from that shown in this preface section. The longer form has been used primarily to introduce you to what the command does. Be sure to examine the summary pages to familiarize yourself with the alternate command forms that are available.

Deleting Files

To try this command, your Workbench or CLI diskette should be in drive 0.

You may be working on several versions of a program or textfile, and eventually wish to delete versions of that file that you don't need anymore. The DELETE command lets you erase files and releases the disk space to AmigaDOS for reuse.

Note: If you DELETE files, it is not possible to retrieve them. Be certain that you really do wish to delete them.

Here is a sample command sequence, that creates a file using the redirection command, types it to verify that it is really there, then deletes it.

```
DIR > directorystuff
TYPE directorystuff
DELETE directorystuff
TYPE directorystuff
```

To the final command in the above sequence, AmigaDOS responds:

```
Can't Open directorystuff
```

indicating that the file can't be found, because you deleted it.

Copying Files

Before you enter this command, your Workbench or CLI diskette should be in drive 0.

On a dual-drive system, copying files is easy:

```
COPY FROM df0:sourcepath TO df1:destinationpath
```

or

```
COPY df0:sourcepath df1:destinationpath
```

On a single-drive system, copying files is a little more complex. You must copy certain system files from your system diskette into the system memory. This is also called using the RAM: device, often known as a ramdisk. Copy the file(s) to the ramdisk, change your directory to the ramdisk, then copy from the ramdisk onto the destination diskette. Here is a sample sequence.

Be sure your Workbench or CLI diskette is in the internal disk drive. Issue the commands:

```
COPY df0:c/cd RAM:
COPY df0:c/copy RAM:
CD RAM:
```

Insert the source data diskette into the drive. (For this example, copy something from the Workbench or CLI diskette, which is already in the drive). Type:

```
COPY df0:c/execute ram:execute
                or
COPY df0:c/execute execute
                or
COPY df0:c/execute ram:
```

Remove the source diskette, and insert the destination diskette into the drive. Type:

```
COPY ram:execute df0:execute
      or
COPY execute df0:execute (If you did the CD RAM: this form works.)
```

Remove the destination diskette and insert your CLI or Workbench diskette again. Type:

```
CD df0:
```

and you are back where you started. The only other command you may want to perform is:

```
DELETE RAM:cd RAM:copy RAM:execute
```

which releases the ramdisk memory to the system for other uses.

Creating a New Directory

You can create a new directory (newdrawer) within the current directory by using the MAKEDIR command:

```
MAKEDIR newdrawer
```

Now if you issue the DIR command, you will see that there is an entry for:

```
newdrawer (dir)
```

You can also use the RENAME command to move a file from one directory (drawer) to another on the same diskette:

```
MAKEDIR newdrawer
RENAME FROM newname TO newdrawer/newname
```

moves the file from the current directory into the newdrawer you have created. To check that it has really been moved, issue the command:

```
DIR
```

Then type:

```
DIR newdrawer
```

AmigaDOS looks in the newdrawer, and shows you that the file named "newname" is there.

Is My File Somewhere on This Disk?

Before you enter this command, your Workbench or CLI diskette copy should be in drive 0.

Sometimes you wish to see everything on the diskette, instead of only

one directory at a time. You can use the DIR command with one of its options:

```
DIR OPT A
```

which lists all directories and subdirectories on the diskette. Keep in mind the <space><BACKSP> combination to pause and restart the listing.

To get a closer look at the disk's contents, you might redirect the output to a file:

```
DIR > mydiskdir OPT A
```

Notice that the redirect-the-output command character and filename MUST come before the list of options for the DIR command.

Now, if you wish, you can TYPE the file mydiskdir and press the space bar to pause the listing. Use the RETURN key to resume the listing. Or, you can use ED to view the file, as follows:

```
ED mydiskdir
```

Use the cursor keys to move up and down in the file.
Use the key combination ESC then T <RETURN> to move to the top of the file.
Such a combination can be referred to as "ESC-T", meaning ESC followed by T.
Use the key combination ESC-B <RETURN> to move to the bottom of the file.
Use the key combination ESC-M then a number <RETURN> to move to a specific line number within the file.
Use the key combination ESC-Q <RETURN> to QUIT without changing the file or
Use ESC-X <RETURN> to write any changes to your file back into the original file name.

Chapter 3 of the *AmigaDOS User's Manual* has more detailed information on using ED.

Doing Something Automatically at Boot Time

There is a file in the "s" subdirectory on your Workbench or CLI diskette called Startup Sequence. This is an execute file. It contains a sequence of CLI commands that AmigaDOS performs whenever you reboot the system. The last two commands in your Workbench diskette Startup Sequence are LoadWb

(load the Workbench program) and ENDCLI which basically leaves the Workbench program in control. You can make up your own Startup Sequence file using ED or EDIT to create a custom version of an execute command sequence. The EXECUTE command summary and tutorial section in the *AmigaDOS User's Manual* has details about various commands that you can have in this file. Note that Startup Sequence can also be used to auto-run a program.

> **WARNING:** Take care to modify only a copy of your diskette ——— never modify the master diskette ——— if you decide to change the Startup Sequence.

Assigning the Diskette on Which AmigaDOS Looks for Things

Before you enter this command, your Workbench or CLI diskette copy should be in drive 0.

Occasionally, you might wish to change to a different diskette and then continue your work. For example, you may have booted the system using a Workbench diskette, then wish to change to a CLI diskette. If the CLI diskette has a directory on it that contains the executable commands you want to perform, (for example, a c(dir)), you can change to that diskette by using the ASSIGN command.

If you don't use ASSIGN, you will have to swap diskettes to get commands done. Here is an example that doesn't use ASSIGN. The intent is to change diskettes and begin using "mydisk:" as the main diskette. Any unneeded files have already been deleted so as to provide workspace.

```
CD mydisk:
```

AmigaDOS responds "insert mydisk into any drive". Insert it, then type:

```
DIR
```

AmigaDOS prompts "insert Workbench [or whatever the boot diskette name was] in any drive". It knows, from boot time, that the DIR command is in the boot diskette, c directory. AmigaDOS reads the DIR command, then asks "insert mydisk in any drive". Any other AmigaDOS command also results in the need for a diskette swap. To avoid this, use the ASSIGN command as follows:

```
ASSIGN c: mydisk:c
```

AmigaDOS asks "insert mydisk into any drive". From now on, all commands to AmigaDOS will be sought from the command (c) directory of this other diskette and AmigaDOS won't ask for the original diskette back for simple commands.

Once you've done this, you'll probably want to type:

```
CD mydisk:
```

There are other things that AmigaDOS can assign. If you issue the command

```
ASSIGN LIST
```

you will see the other things as well. If you run a program that requires a serial device (modem, printer) or a parallel device (printer), AmigaDOS looks in the directory currently assigned to DEVS: to locate the device. If all of the system directories are on this new main diskette, you can avoid having AmigaDOS ask you to reinsert the original diskette by providing an execute file on your diskettes that reassigns all devices to that diskette. The contents of this execute file for a diskette named "mydisk" are as follows:

```
ASSIGN SYS: mydisk:
ASSIGN S: mydisk:s
ASSIGN DEVS: mydisk:devs
ASSIGN L: mydisk:l
ASSIGN FONTS: mydisk:fonts
ASSIGN LIBS: mydisk:libs
```

To create this execute file, use the command:

```
COPY FROM * TO reassign
```

Then type the above ASSIGN lines. After you've typed the last line, enter the key combination CTRL-\ which ends the file. The "*" stands for the keyboard and current CLI window, so this method of creating a file is one possible alternative to using ED or EDIT.

Creating a New CLI

AmigaDOS is a multi-tasking system. You can have multiple windows open at the same time, each with its own current directory and executing separate commands. You create a new CLI by using the command NEWCLI:

```
NEWCLI
```

This opens a separate window, with a prompt that identifies the current process. For example, if the first window has a prompt:

```
1>
```

then the new CLI might have a prompt:

```
2>
```

You can move the new window around, make it bigger, make it smaller and so on. To issue commands to the new CLI, click within its window. Now anything you type goes into the window where you clicked the selection button most recently. Try the following:

1. Click in window 1, then type:

```
DIR df0:c
```

2. Quickly click in window 2, and type:

```
INFO
```

Both CLIs will work at the same time to fulfill your requests. This demonstrates the multi-tasking capabilities of the Amiga. Notice that you aren't limited to only two CLIs, you can, if there is memory available, open as many as 20 CLIs.

Closing a CLI

You finish with a CLI and close its window with the command ENDCLI. Click the selection button of the mouse in the window for the CLI you wish to close, and type:

```
ENDCLI
```

That's all there is to it.

Closing Comments

The above series of command descriptions introduces you to the kinds of things you can do with AmigaDOS commands from the CLI. There are several commands that haven't been covered in the above session at all. In addition,

most of the commands described above have other "templates" (ways you can enter the commands) and options that haven't been demonstrated.

Chapter 2 of the *AmigaDOS User's Manual* contains a reference section that shows the templates for each of the commands in AmigaDOS. You can look at the description for each command to find more information. Once you are familiar with the commands, and the forms in which you can use them, the quick reference listing at the end of the chapter will be useful to remind you of the commands that are available.

1.7 Conventions Used

In Chapter 2 of this manual, in the "Format" description for the AmigaDOS commands, you will find the following notations used:

<name>	Indicates a parameter name that you should fill in for this command. Example: EXECUTE <commandfile> where the name of the command file is a required parameter.
[ ]	Square brackets are used to indicate that an item is optional. It needn't be provided for the command to function but, if provided, conveys additional information to AmigaDOS about how to perform the command. Example: QUIT [<code>]
|	A vertical bar tells you that you can select one or another of the alternatives that are separated by the vertical bar for a command. Example: DIR [OPT A | I | AI] The example indicates that you can choose A, I or AI for the specification.
<name>*	Indicates one-or-more occurrences of a parameter name; if you supply more than one such parameter, individual parameters must be separated by at least one blank space.

For AmigaDOS CLI commands, unless some form of punctuation, such as a comma or a plus-sign is actually included in the command Format line, you must always separate the parameters with blank spaces. Don't confuse the Format information with the "Template" for the command. The command template is explained in section 1.4.5 of the *AmigaDOS User's Manual.*

Chapter 2

AmigaDOS Commands

This chapter is divided into two parts: the first part describes the user commands available on the Amiga; the second describes the developer commands. The user commands fall into several categories: file utilities, CLI control, command sequence control, and system and storage management. Part I provides alphabetized command descriptions that give the format, template, purpose, and specification of each command as well as an example of its use. Part 2 has the same organization.

The chapter starts with a list of unfamiliar terminology. At the end of the chapter there is a quick Contents reference card that lists all the commands by function.

2.1 AmigaDOS User's Commands

Unfamiliar Terminology

In this manual you could find some terms that you have not seen before. The list below includes some common terms that are confusing if you are unfamiliar with them.

Boot — startup. It comes from the expression "pulling yourself up by your bootstraps."

Default — initial setting or, in other words, what happens if you do nothing. So that, in this manual, "default" is used to mean "in absence of something else".

Device name — part of a name that precedes the colon (:), for example, CON:, DFO:, PRT:, and so forth.

File handle an internal AmigaDOS value that represents an open file or device.

Logical device a name you can give to a directory with ASSIGN that you can then use as a device name.

Object code binary output from an assembler or compiler, and binary input to a linker.

Reboot restart.

Stream an open file or device that is associated with a file handle. For example, the input stream could be from a file and the output stream could be to the console device.

System disk a disk containing the Workbench and commands.

Volume name a name you give to a physical disk.

Note: Command format is explained in section 1.7; command template is explained in section 1.4.5.

;

Format: [<command>];[<comment>]
Template: "command";"comment"
Purpose: To add comments to command lines.
Specification:
The CLI ignores everything after the semicolon (;).

Examples:

```
;This line is only a comment
```

ignores the part of the line containing "This line is only a comment."

```
copy <file> to prt: ; print the file
```

copies the file to the printer, but ignores the comment "print the file."

See also: EXECUTE

><

Format: <command>[>outputfilename][inputfilename][<commandargs*]
Template: "command">"TO"<"FROM" "args"
Purpose: To direct command input and output.

Specification:
You use the symbols > and < to direct the output and input of a command. The direction of the point of the angle bracket indicates the direction of information flow. You can use these symbols to change where any command reads input or writes output. The output from a command usually goes to the current window. However, if you type a > symbol after a command and before a filename, the command writes the output to that file instead. Similarly, if you type the < symbol before a filename, the command reads from that file instead of from the keyboard.

You do not have to specify both the TO and FROM directions and files. The existence and number of "args" depends on the command you used. Redirection only happens for the command you specified. AmigaDOS reverts to the initial or "default" input and output (that is, the keyboard and current window) afterward. Notice that redirection must *precede* the arguments.

Examples:

```
DATE > diary_dates
```

writes the output of the DATE command (that is, today's date and time) to the file "diary_dates".

```
my_program < my_input
```

tells my_program to accept input from my_input instead of from the keyboard.

```
LIST > temp
SORT temp TO *
```

produces a sorted list of files and displays them on the screen.

The following sequence:

```
ECHO > 2nd.date 02-jan-78
DATE < 2nd.date ?
DELETE 2nd.date
```

creates a file called 2nd.date that contains the text "02-jan-78<linefeed>". Next it uses this file as input to the command DATE. Note that the "?" is necessary for DATE to accept input from the standard input, rather than the command line. Finally, as you no longer need the file, the DELETE command deletes 2nd.date.

ASSIGN

Format: ASSIGN [[<name>]<dir>][LIST]
Template: ASSIGN "NAME,DIR,LIST/S"
Purpose: To assign a logical device name to a filing system directory.
Specification:
NAME is the logical device name given to the directory specified by DIR.

If you just give the NAME, AmigaDOS deletes the logical device name given (that is, it removes the assignment).

ASSIGN without any parameters or the switch LIST diplays a listing of all current assignments.

When you use ASSIGN, you must ensure that there is a disk inserted in the drive. This is important because ASSIGN makes an assignment to a disk volume and not to a drive.

Note that the effect of ASSIGN is lost when you restart or "reboot" your computer.

Examples:

```
ASSIGN sources: :new/work
```

Sets up the logical device name "sources" to the directory ":new/work". Then to gain access to files in ":new/work", you can use the logical device name "sources", as in

```
TYPE sources:xyz
```

which displays the file ":new/work/xyz".

```
ASSIGN LIST
```

lists the current logical device names in use.

BREAK

Format: BREAK <task>[ALL][C][D][E][F]
Template: BREAK "TASK/A,ALL/S,C/S,D/S,E/S,F/S"
Purpose: To set attention flags in the given process.
Specification:
BREAK sets the specified attention flags in the process. C sets the CTRL-C flag, D sets the CTRL-D flag, and so on. ALL sets all the flags from CTRL-C through

CTRL-F. By default, AmigaDOS only sets the CTRL-C flag. The action of BREAK is identical to selecting the relevant process by moving the mouse to the window, clicking the Selection Button, and pressing the required control key combination.

Examples:

```
BREAK 7
```

sets the CTRL-C attention flag of process 7. This is identical to selecting process 7 and pressing CTRL-C.

```
BREAK 5 D
```

sets the CTRL-D attention flag of process 5.

```
BREAK 3 D E
```

sets both CTRL-D and CTRL-E.

CD

Format: CD[<dir>]
Template: CD "DIR"
Purpose: To set or change a current directory or drive.
Specification:
CD with no parameters displays the name of the current directory. In the format list above, <dir> indicates a new current directory (that is, one in which unqualified filenames are looked up). If the directory you specify is not on the current drive, then CD also changes the current drive.

To change the current directory to the directory that owns the current one (if one exists), type CD followed by a single slash (/). Thus CD / moves the current directory one level up in the hierarchy unless the current directory is a root directory (that is, the top level in the filing system). Multiple slashes are allowed; each slash refers to an additional level above.

Examples:

```
CD df1:work
```

sets the current directory to "work" on disk "df1", and sets the current drive to "df1".

```
CD SYS:COM/BASIC
CD /
```

sets the current directory to "SYS:COM".

COPY

Format: COPY [[FROM]<name>][TO<name>][ALL][QUIET]
Template: COPY "FROM,TO/A,ALL/S,QUIET/S"
Purpose: To copy a file or directory from one place to another.
Specification:
COPY places a copy of the file or directory in the file or directory specified as TO. The previous contents of TO, if any, are lost.

If you specify a directory name as FROM, COPY copies all the files in the FROM directory to the TO directory. If you do not specify the FROM directory, AmigaDOS uses the current directory. The TO directory must exist for COPY to work; it is not created by COPY.

If you specify ALL, COPY also copies the files in any subdirectories. In this case, it automatically creates subdirectories in the TO directory, as required. The name of the current file being copied is displayed on the screen as it happens unless you give the QUIET switch.

You can also specify the source directory as a pattern. In this case, AmigaDOS copies any files that match the pattern. See the command LIST for a full description of patterns. You may specify directory levels as well as patterns.

Examples:

```
COPY file1 TO :work/file2
```

copies 'file1' in the current directory to "file2" in the directory ":work".

```
COPY TO df1:backup
```

copies all the files in the current directory to "df1:backup". It does not copy any subdirectories, and df1: backup must already exist.

```
COPY df0: to df1: ALL QUIET
```

makes a logical copy of disk"df0" on disk"df1" without any reflection of filenames.

```
COPY test-#? to df1:xyz
```

copies all files in the current directory that start "test-" to the directory xyz on the disk "df1", assuming that "xyz" already exists. (For an explanation of patterns, such as "#?", see the command LIST in this chapter.)

```
COPY test_file to PRT:
```

copies the file "test_file" to your printer.

```
COPY * TO CON:10/10/200/100/
```

Click the window that you typed the copy command into. This "reactivates" it so that console input is taken from there. Every time you type a line it will be displayed in the new window. Press CTRL-\ when you are done and the new window will close.

```
COPY DF0:?/#? TO DF1: ALL
```

copies every file in any one character subdirectory of DF0: to the root directory of DF1:.

See also: JOIN

DATE

Format: DATE [<date>][<time>][TO|VER<name>]
Template: DATE "DATE,TIME,TO=VER/K"
Purpose: To display or set the system date or time.
Specification:

DATE with no parameter displays the currently set system date and time. This includes the day of the week. Time is displayed using a 24-hour clock.

DATE <date> sets the date. The form of <date> is DD-MMM-YY. If the date is already set, you can reset it by specifying a day name (this sets the date forward to that day) or by specifying 'tomorrow' or 'yesterday'.

DATE <time> sets the time. The form of <time> is HH:MM (for Hours and Minutes). You should use leading zeros when necessary. Note that, if you use a colon (:), AmigaDOS recognizes that you have specified the time rather than the date. That is to say, you can set both the date and the time, or either date or time in any order because DATE only refers to the time when you use the form HH:MM.

If you do not set the date, the restart disk validation process sets the system date to the date of the most recently created file. See Chapter 1 for details on the restart validation process.

To specify the destination of the verification, you use the equivalent keywords TO and VER. The destination is the terminal unless you specify otherwise.

Note: If you type DATE before the restart validation has completed, the time is displayed as unset. To set the time, you can either use DATE or just wait until the validation process is finished.

Examples:

DATE

displays the current date.

DATE 06-Sep-82

sets the date to the 6th of September 1982. The time is not reset.

DATE tomorrow

resets the date to one day ahead.

DATE TO fred

sends the current date to the file "fred".

DATE 10:50

sets the current time to ten 'til eleven.

DATE 23:00

sets the current time to 11:00 P.M.

DATE 01-JAN-02

sets the date to January 1st, 2002. (The earliest date you can set is 01-JAN-78.)

DELETE

Format: DELETE <name>[<name>*][ALL][Q|QUIET]
Template: DELETE ",,,,,,,,,ALL/S,Q=QUIET/S"
Purpose: To delete up to ten files or directories.
Specification:

DELETE attempts to delete each file you specify. If it cannot delete a file, the screen displays a message, and AmigaDOS attempts to delete the next file in the list. You may not delete a directory if it contains any files.

You can also use a pattern to specify the filename. See the description of the command LIST for full details of patterns. The pattern may specify directory levels as well as filenames. In this case, all files that match the pattern are deleted.

If you specify ALL with a directory name, DELETE will delete that directory and all subdirectories and files within that directory and its subdirectories.

Unless you specify the switch QUIET (or use the alternative, Q), the name of the file being deleted appears on the screen as it happens.

Examples:

```
DELETE old-file
```

deletes the file "old-file".

```
DELETE work/prog1 work/prog2 work
```

deletes the files "prog1" and "prog2" in the directory "work", and then deletes the directory "work".

```
DELETE t#?/#?(1|2)
```

deletes all the files that end in "1" or "2" in directories that start with "t". (For an explanation of patterns, such as "#?", see the command LIST later in this chapter.)

```
DELETE DF1:#? ALL
```

deletes all the files on DF1:.

See also: DIR (1-DEL option)

DIR

Format: DIR[<name>][OPT A|I|AI]
Template: DIR "DIR,OPT/K"
Purpose: To provide a display of the files in a directory in sorted order. DIR can also include the files in subdirectories, and you can use DIR in interactive mode.
Specification:
DIR alone shows the files in the current directory. DIR followed by a directory provides the files in that directory. The form of the display is first any

subdirectories, followed by a sorted list of the files in two columns. If you want to know if a file exists type LIST filename.

Typing DIR filename, where filename is a file which exists results in the Amiga responding with: "filename is not a directory."

To pass options to DIR, use the OPT keyword. Use the A option to include any subdirectories below the specified one in the list. Each sublist of files is indented.

To list only the directory names use the D option.

The I option specifies that DIR is to run in interactive mode. In this case, the files and directories are displayed with a question mark following each name. Press RETURN to display the next name in the list. To quit the program, type Q. To go back to the previous directory level or to stop (if at the level of the initial directory), type B.

If the name displayed is that of a directory, type E to enter that directory and display the files and subdirectories. Use E and B to select different levels. Typing the command DEL (that is, typing the three letters D E L, not pressing the DEL key) can be used to delete a directory, but this only works if the directory is empty.

If the name is that of a file, typing DEL deletes the file, or typing T Types (that is, displays) the file on the screen. In the last case, press CTRL-C to stop it "typing" and return to interactive mode.

To find the possible responses to an interactive request, type?.

Examples:

```
DIR
```

provides a list of files in current directory.

```
DIR df0: OPT a
```

lists the entire directory structure of the disk "df0".

DISKCOPY

Format: DISKCOPY [FROM]<disk>TO<disk>[NAME <name>]
Template: DISKCOPY "FROM/A,TO/A/K,NAME/K"
Purpose: To copy the contents of one 3-½ inch floppy disk to another.
Specifications:
DISKCOPY makes a copy of the entire contents of the disk you specified as FROM, overwriting the previous contents of the entire disk you specified as

TO. DISKCOPY also formats a new disk as it copies. You normally use the command to produce backup floppy disks.

Once you have given the command, AmigaDOS prompts you to insert the correct disks. At this point, you insert the correct source and destination disks.

You can use the command to copy any 3-½ inch AmigaDOS disk to another, but the source and destination disks must be identical in size and structure. To copy information between different sized disks, you use COPY.

You can also use the command to copy a floppy disk using a single floppy drive. If you specify the source and destination as the same device, then the program reads in as much of the source disk into memory as possible. It then prompts you to place the destination disk in the drive and then copies the information from memory onto the destination disk. This sequence is repeated as many times as required.

If you do not specify a new name for your disk, DISKCOPY creates a new disk with the same name as the old one. However, AmigaDOS can tell the difference between two disks with the same name because every disk is associated with the date and time of its creation. DISKCOPY gives the new disk the current system date as its creation date and time.

Note: To copy part of a disk, you can use COPY to RAM:.

Examples:

```
DISKCOPY FROM df0: TO df1:
```

makes a backup copy of the disk "df0" onto disk "df1".

```
DISKCOPY FROM df0: To df0:
```

makes a backup copy of the disk in drive "df0" using only a single drive.

See also: COPY

ECHO

Format: ECHO <string>
Template: ECHO " "
Purpose: To display the argument given.
Specification:
ECHO writes the single argument to the current output stream (which can be a file or a device). This is normally only useful within a command sequence or as part of a RUN command. If you give the argument incorrectly, an error is displayed.

Examples:

```
RUN COPY :work/prog to df1:work ALL QUIET +
ECHO "Copy finished"
```

creates a new CLI to copy the specified directory as a background process. When it has finished, the screen displays

Copy finished

If the following Execute file exists

```
ECHO "Starting 'MYCOPY' Execute file"
COPY DF1:ABC TO RAM:ABC
COPY DF1:XYZ TO RAM:XYZ
ECHO "Remove the diskette in DF1:"
ECHO "Insert the new diskette in DF1:"
WAIT 10 SECS
COPY RAM:ABC TO DF1:ABC
COPY RAM:XYZ TO DF1:ABC
ECHO "Done"
```

then

```
EXECUTE MYCOPY
```

copies 2 files to RAM disk and back.

ED

Format: ED[FROM]<name>[SIZE<n>]
Template: ED "FROM/A,SIZE"
Purpose: To edit text files.
Specification:
ED is a screen editor. You can use ED as an alternative to the line editor EDIT. The file you specify as FROM is read into memory, then ED accepts your editing instructions. If FROM filename does not exist, AmigaDOS creates a new file.

Because the file is read into memory, there is a limit to the size of file you can edit with ED. Unless you specify otherwise, workspace size is 40,000 bytes. This workspace size is usually sufficient for most files. However, to alter the workspace, you specify a suitable value after the SIZE keyword.

There is a full specification of ED in Chapter 3.

Examples:

```
ED work/prog
```

edits the file "work/prog", assuming it exists; otherwise, ED creates the file.

```
ED huge-file SIZE 50000
```

edits a very large file "huge-file", using a workspace of 50,000 bytes.

EDIT

Format: EDIT [FROM]<name>[[TO]<name>][WITH<name>][VER<name>] [OPT<option>]
Template: EDIT "FROM/A,TO,WITH/K,VER/K,OPT/K"
Purpose: To edit text files.
Specification:
EDIT is a line editor (that is, it edits a sequential file line by line). If you specify TO, EDIT copies from file FROM to file TO. Once you have completed the editing, the file TO contains the edited result, and the file FROM is unchanged. If you do not specify TO, then EDIT writes the edited text to a temporary file. If you give the EDIT commands Q or W, then EDIT renames this temporary file FROM, having first saved the old version of FROM in the file ":t/edit-backup". If you give the EDIT command STOP, then EDIT makes no change to the file FROM.

EDIT reads commands from the current input stream, or from a WITH file if it is specified.

EDIT sends editor messages and verification output to the file you specify with VER. If you omit VER, the terminal is used instead.

OPT specifies options: Pn sets the maximum number of previous lines to n; Wn sets the maximum line width. The initial setting is P4OW120.

Note: You cannot use the < and > symbols to redirect input and output when you call EDIT.

See Chapter 4 for a full specification of EDIT.

Examples:

```
EDIT work/prog
```

edits the file "work/prog". When editing is complete, EDIT saves the old version of "work/prog" in ":t/edit-backup".

```
EDIT work/prog TO work/newprog
```

edits the file "work/prog", placing the edited result in the file "work/newprog".

```
EDIT work/prog WITH edits/0 VER nil:
```

edits the file "work/prog" with the edit commands stored in the file "edits/0". Verification output from EDIT is sent to the dummy device "nil:".

ENDCLI

Format: ENDCLI
Template: ENDCLI
Purpose: To end an interactive CLI process.
Specification:
AmigaDOS only allows ENDCLI as an interactive command. ENDCLI removes the CLI currently selected by the mouse.

You shouldn't use ENDCLI except on a CLI created by the NEWCLI command. If the initial CLI (process 1) is ended, and no other has been set up by the NEWCLI command, then the effect is to terminate the AmigaDOS session.

Note that there are no arguments to the ENDCLI command, and no check for invalid arguments.

Note: Do not experiment with ENDCLI before you've used NEWCLI. Using ENDCLI on the initial CLI always pulls the rug out from under you by terminating that CLI. If you started the CLI from the Workbench, then there is no problem as you are returned to the Workbench. If you started AmigaDOS with just the CLI running, then ending the last CLI gives you no way of creating a new one.

Examples:
The following sequence:

```
NEWCLI
LIST
ENDCLI
```

opens a new window, lists the directory, and closes the window again.

EXECUTE

Format: EXECUTE <commandfile>[<arg>*]
Template: EXECUTE "command-file","args"
Purpose: To execute a file of commands with argument substitution.
Specification:

You normally use EXECUTE to save typing. The command file contains commands executed by the Command Line Interface. AmigaDOS executes these commands one at a time, just as though you had typed them at the keyboard. If the execution creates a new CLI window, the results may not be identical to typing at the keyboard.

You can also use EXECUTE to perform parameter (that is, value) substitution, where you can give certain names as parameters. Before the command file is executed, AmigaDOS checks the parameter names with those you've given after the EXECUTE command. If any match, AmigaDOS uses the values you specified instead of the parameter name. Parameters may have values specified that AmigaDOS uses if you do not explicitly set the parameter. If you have not specified a parameter, and if there is no default, then the value of the parameter is empty and nothing is substituted for it.

To use parameter substitution, you give directives to the EXECUTE command. To indicate these, you start a line with a special character, which is initially a period or "dot" (.). The directives are as follows:

.KEY		Argument template, used to specify the format of the arguments, may be abbreviated to .K
.DOT	ch	Change dot character (initially".") to ch
.BRA	ch	Change bra character (initially "<") to ch
.KET	ch	Change ket character (initially ">") to ch
.DOLLAR	ch	Change default-char (initially "$") to ch, may be abbreviated to .DOL
.DEF	keyword value	Give default to parameter
.<space>		Comment line
.<newline>		Blank comment line

Before execution, AmigaDOS scans the contents of the file for any items enclosed by BRA and KET characters ("<" and ">"). Such items may consist of a keyword or a keyword and a default value for AmigaDOS to use if you have left the keyword unset. (To separate the keyword and the default, if there is one, you type a dollar sign "$"). Thus, AmigaDOS replaces <ANIMAL> with

the value you associated with the keyword ANIMAL, while it replaces <ANIMAL$WOMBAT> with the value of ANIMAL if it has one, and otherwise it defaults to WOMBAT.

A file can only use the dot commands if the first line has a dot command on it. The CLI looks at the first line. If it starts with a dot command, for example, a comment (.<space>txt) then the CLI scans the file looking for parameter substitution and builds a temporary file in the :T directory. If the file doesn't start with a dot command, then it is assumed that there are NO dot commands in the file, which also means no parameter substitution is performed. For the no-dot case, the CLI starts executing the file directly without having to copy it to :T. Note that you can still embed comments in an execute file by using the CLI's comment character, the semicolon (;). If you don't need parameter substitution and dot commands, don't use them. They save you extra accesses to the disk for the temporary file.

AmigaDOS provides a number of commands that are only useful in command sequence files. These include IF, SKIP, LAB, and QUIT. These can be nested in a command file.

Note that you can also nest EXECUTE files. That is, you can have a command file that contains EXECUTE commands.

To stop the execution of a command file, you press CTRL-D. If you are nesting command files, that is, if one command file calls another, you can stop the entire set of EXECUTE commands by pressing CTRL-C. CTRL-D only stops the current command file from executing.

Examples:
Assume the file "list" contains the following:

```
.k filename/a
run copy <filename> to prt: +
echo "Printing of <filename> done"
```

Then the following command

```
EXECUTE list test/prg
```

acts as though you had typed the following commands at the keyboard.

```
RUN copy test/prg to prt: +
ECHO "Printing of test/prg done"
```

Another example, "display", uses more of the features described above:

```
.key name/a
IF EXISTS <name>
TYPE <name> OPT n  (If the file given is on the current directory, type it
                    with line numbers)
ELSE
ECHO "<name> is not on this directory"
ENDIF


RUN EXECUTE display work/prg2
```

should display the file work/prg2 with line numbers on the terminal if it exists on the current directory. If the file is not there, the screen displays the following message:

work/prg2 is not on this directory.

See also: ;,IF,SKIP,FAILAT,LAB,ECHO,RUN,QUIT

Additional Examples for the EXECUTE Command:

Example #1
Parameter Substitution by Keyword Name and/or Position

The .KEY (or .K) statement supplies both keyword names and positions in command files. It tells EXECUTE how many parameters to expect and how to interpret them. In other words, .KEY serves as a "template" for the parameter values you specify. Only one .KEY statement is allowed per command file. If present, it should be the first command line in the file.

When you enter a command line, AmigaDOS resolves parameter substitutions for the keywords in two ways: by specification of the keyword in front of the parameter, and by the relative positions of the parameters in the line. Keyword name substitution takes precedence.

Assume that the execute file named DEMO1 contains the following .KEY statement:

```
.KEY flash,pan
```

tells AmigaDOS to expect two parameter substitutions, <flash> and <pan>. (The angle brackets indicate the keyword value to be substituted at execution time.)

Suppose you enter the following command line:

```
EXECUTE DEMO1 pan somename flash othername
```

The value "othername" is assigned to <flash>, and the value "somename" is assigned to <pan>.

You can omit the keyword names if the parameter substitutions are in the order given in the .KEY statement. For example, the following statement is equivalent to the preceding one:

```
EXECUTE DEMO1 othername somename
```

This is because the values correspond to the keyword order specified in the .KEY statement.

You can also mix the two methods of parameter substitution. Suppose you have a .KEY statement with several parameters, as follows:

```
.KEY word1, word2, word3, word4
```

The execute file processor removes parameter names from the input line to fill the meanings of any keyword values it finds. Then, with any remaining input, it fills the leftover keyword positions according to the position of the input value.

For example:

```
EXECUTE DEMO2 word3 ccc word1 aaa bbb ddd
```

The processor assigns ccc to <word3>, aaa to <word1>, and has two parameters left over. Scanning from left to right in the .KEY statement, it finds that <word2> is still unassigned. Thus, <word2> gets the next input word, bbb. Finally, <word4> hasn't been assigned either, so it gets the last input word, ddd.

You can indicate special conditions for parameter substitution, as follows:

```
.KEY name1/a, name2/a, name3, name4/k
```

The "/a" indicates that a value must be supplied to fill the parameters for name1 and name2. Values for name3 and name4 are optional, though the "/k" indicates that <name4> (if supplied) must be preceded by the explicit keyword "name4." For example:

```
EXECUTE DEMO3 fee fie foe name4 fum
```

If the user does not supply a required parameter (such as name1 or name2 in the preceding example), EXECUTE issues an error message.

As an example of the use of the /k option, suppose you have created an execute file named COMPILE and it lets you optionally specify a filename to which a printout of the compilation is to be directed. Your .key statement might read:

```
.key compilewhat/a,printfile/k
```

If a user enters a line such as:

```
EXECUTE COMPILE myfile PRINTFILE myprint
```

the execute file says the keyword PRINTFILE is optional and need not be supplied, but if used, there must be a value entered along with it. Thus the above line is correct, since myprint is specified as the target output file.

Example #2
Assigning Default Parameters and Different Bracket Characters

```
.KEY word1
```

The .DEF directive establishes a default value for a keyword if the user does not specify a value on the command line. To detect an unsupplied parameter value, you can compare it to "" (two double-quotes in a row). You must perform this comparison before executing any .DEF statement in the execute file.

You can assign defaults in either of two ways. The first way requires that you specify the default every time you reference a parameter, using the "$" operator.

For example, in the following statement:

```
ECHO "<word1$defword1> is the default for Word1."
```

"defword1" is the default specified for word1 and is printed when the above statement executes. The second way is to define a default once. For example, with the following assignment:

```
.DEF word1 "defword1"
```

you can execute the following statement:

```
ECHO "<word1> is the default for Word1."
```

The output of both of the above ECHO statements will be:

```
defword1 is the default for Word1.
```

Note that a second use of .DEF for a given parameter has no effect:

```
.DEF word1 "——— New default———"
ECHO "<word1> is STILL the default for Word1."
```

(The first assignment, "defword1" will be substituted for word1 at execution time.)

Assigning Different Bracket Characters

Wherever EXECUTE finds enclosing angle brackets, it looks within them to see if it can substitute a parameter. An unsupplied parameter with no default becomes a "null" string.

Suppose you want to use a string that contains the angle bracket characters, < and >. You can use the directives .BRA and .KET to define substitutes for the bracket characters. For example,

```
ECHO "This line does NOT print <angle> brackets."
.BRA {
.KET }
ECHO "This line DOES print <angle> brackets."
ECHO "The default for word1 is {word1}."
```

The first ECHO statement causes the processor to look for the parameter substitution for "angle," since that's the current meaning of the angle bracket characters. Since "angle" wasn't included in the .KEY statement, the processor substitutes the null string for it. Then, after the .BRA and .KET directives redefine the bracket characters, the second ECHO statement prints the characters:

```
This line DOES print <angle> brackets.
```

The third ECHO statement illustrates that the braces ({ and }) now function to enclose keywords for the purpose of parameter substitution.

Example #3
File Copy Simulation Showing Command File Structures

The IF statement lets you perform tests and cause different actions based on the results of those tests. Among the possible tests are testing strings for equality and testing to see if a file exists. You can use an ELSE statement with

an IF to specify what should be done in case the IF condition is not true. The ELSE statement, if used, is considered a part of the IF statement block. An ENDIF terminates an IF statement block.

The example programs below also use a SKIP statement. The SKIP statement lets you skip FORWARD ONLY within your execute file to a label defined by a LAB statement.

The IF . . . ENDIF structure is illustrated by the following short example. It is generally a good idea to test for keywords that might be omitted, or might be entered as null ("") in quotes, as shown below:

```
IF "<word1>" EQ "usage"
  SKIP USAGE
ENDIF
IF "<word2>" EQ ""
  SKIP USAGE
ENDIF
```

Enclosing your parameter substitution words in double quotes within IF statements prevents EXECUTE from reporting an error if the keyword is omitted.

If you omit the double quotes and the value is not supplied, the result can be a line that reads:

```
IF EQ "usage"
```

This produces an error, because the two operators IF and EQ are adjacent. Using double quotes around the keyword replacement indicators results in a line that reads:

```
IF "" EQ "usage"
```

which is legal.

You can use NOT in an IF statement to reverse the meaning of the test you perform. For example:

```
IF NOT exists <from>
```

There can be nothing on the IF line other than the test condition. For example, the following is incorrect:

```
IF <something> EQ true SKIP DONE
```

The correct form of the above statement is as follows:

```
IF <something> EQ "true"
  SKIP DONE
ENDIF
```

As the example above shows, the string constant tested for need not be enclosed in double quotes; in the preceding example, either "TRUE" or TRUE is acceptable.

As shown in the sample command file below, IF statements can be nested so that commands can be executed based on multiple true statements. Note that EXECUTE lets you indent to make the nesting of IF statements more readable.

The following sample command file simulates a file copying utility that illustrates certain useful structures in a command file: IF . . . [ELSE] . . . ENDIF, LAB, and SKIP.

```
.KEY from, to                          ;(Assign parameter list)
IF "<from>" eq ""                      ;(Check for a FROM file)
being supplied.
  SKIP usage                           ;(No file, show user how to)
use.
ENDIF
IF "<to>" eq ""                        ;(Check for a TO file)
being supplied.
  SKIP usage                           ;(No file, show user how to use)
ENDIF

IF NOT exists <from>                   ;(Check if FROM file doesn't exist)
  ECHO "The from file you selected     ;(<from>) could not be found."
  ECHO "Please use the DIR or LIST     command and try again."
  SKIP DONE                            ;(Note: We can SKIP out of an IF.)
ENDIF

IF exists <to>                         ;(Check if TO file exists.)
  IF "<o>" EQ "O"                      ;(Did the user supply "O")
on the line?
      copy from <from> to <to>
      ECHO "Replaced file named <to> with a copy of file named <from>"
      ECHO "Request fulfilled."
  ELSE
      ECHO "Command will overwrite an existing file ONLY if"
      ECHO "the O parameter is specified."
      ECHO "Request Denied"
      SKIP usage                       ;(Explain how to use this file)
    ENDIF
```

```
ELSE
      ECHO "copy from <from> to <to>."
ENDIF
SKIP DONE

LAB usage
ECHO "cp: usage...."
ECHO "The following copy forms are supported:"
ECHO " x cp FROM sourcefile TO destinationfile"
ECHO " x cp FROM sourcefile destinationfile"
ECHO " x cp sourcefile TO destinationfile"
ECHO " x cp sourcefile destinationfile"
ECHO " x cp TO destinationfile FROM sourcefile"
ECHO " x cp sourcefile destinationfile O"
ECHO " x cp FROM sourcefile TO destinationfile O"
ECHO " x cp O FROM sourcefile TO destinationfile"
ECHO "where: x is short for EXECUTE; cp is the name of"
ECHO "this command file, and "O" means 'overwrite existing file'."

LAB DONE
```

Example #4
Sample Looping Batch File

The SKIP command allows only forward jumps. To create a loop structure within a command file, use EXECUTE iteratively. That is, use the EXECUTE command within the file itself to send execution backwards to a label. The following executable example illustrates looping.

This file displays five messages:

```
"This message prints once at the beginning. (parm1, parm2)"
"Loop number I."
"Loop number II."
"Loop number III."
"This message prints once at the end. (parm1, parm2)"
```

```
.KEY parm1,parm2, loopcnt, looplabel
FAILAT 20
IF NOT "<looplabel>" EQ ""          ;(Called with label?)
    SKIP <looplabel>                ;(Yes, then loop.)
ENDIF

ECHO "This message prints once   ;(Start of loop)
  at the beginning. (<parm1>, <parm2>)"
```

```
LAB 1st-loop
IF "<loopcnt>" EQ "III"              ;(Are we done looping?)
SKIP loopend-<looplabel>             ;(Yes, unwind.)
ENDIF
ECHO "Loop number <loopcnt>I."  ;(Go "backwards" in this file.)
EXECUTE. loop.sample "<parm1>" "<parm2>" <loopcnt>I 1st-loop

LAB loopend-<looplabel>
IF NOT "<loopcnt>" EQ ""
  SKIP EXIT
ENDIF
                                     ;(End of loop)
ECHO "This message prints once at the end. [<parm1>,<parm2>)"

LAB EXIT
```

FAILAT

Format: FAILAT <n>
Template: FAILAT "RCLIM"
Purpose: To instruct a command sequence to fail if a program returns an error code greater than or equal to this number.
Specification:
Commands indicate that they have failed in some way by setting a return code. A nonzero return code indicates that the command has found an error of some sort. A return code greater than or equal to a certain limit (the fail limit) terminates a sequence of noninteractive commands (that is, the commands that you specify after RUN or in an EXECUTE file). The return code indicates how serious the error was, and is normally 5, 10, or 20.

You may use the FAILAT command to alter this fail level from its initial value of 10. If you increase the level, you indicate that certain classes of error should not be regarded as fatal, and that execution of subsequent commands may proceed after an error. The argument should be a positive number. The fail level is reset to the initial value of 10 on exit from the command sequence.

You must use FAILAT before commands such as IF to test to see if a command has failed; otherwise, the command sequence terminates before executing the IF command.

If you omit the argument, the current value of the fail level is displayed.

Examples:

```
FAILAT 25
```

The command sequence only terminates before the end if a command stops with a return code greater than or equal to (>=)25.

See also: IF, EXECUTE,RUN,QUIT

FAULT

Format: FAULT[<n>*]
Template: FAULT ",,,,,,,,"
Purpose: To display the messages corresponding to the fault codes you supply.
Specification:
AmigaDOS looks up the numbers and displays the corresponding messages. Up to ten messages may be displayed.

Examples:

```
FAULT 222
```

displays the message for fault 222.

```
FAULT 221 103 121 218
```

displays the messages for faults 221, 103, 121, and 218.

FILENOTE

Format: FILENOTE[FILE]<file>COMMENT<string>
Template: FILENOTE "FILE/A,COMMENT/K"
Purpose: To attach a comment or a note to a file.
Specification:
FILENOTE assigns a comment to a specified file.

The keyword COMMENT introduces an optional comment of up to 80 characters. A comment may be more than one word (that is, contain spaces between characters). In this case, you must enclose the comment within double quotes (").

A comment is associated with a particular file. When you examine the file with the command LIST, the comment appears on the line below:

```
prog        30 rwed Today 11:07:33
  : version 3.2 - 23-mar-85
```

When you create a new file, it does not normally have a comment. If you overwrite an existing file that has a comment, then the comment is retained

even though the contents of the file have changed. The command COPY copies a file. If a file with a comment is copied, the new file does not have the comment from the original attached to it although the destination file may have a comment which is retained.

Examples:

```
FILENOTE prog2 COMMENT "Ver 3.3 26-mar-85"
```

attaches the comment "Ver 3.3 26-mar-85" to program 2.

See also: LIST

FORMAT

Format: FORMAT DRIVE <drivename> NAME <string>
Template: FORMAT"DRIVE/A/K,NAME/A/K"
Purpose: To format and initialize a new 3½-inch floppy disk.
Specification:
The program formats a new floppy disk in the method required for AmigaDOS. Once the disk is formatted, it is initialized and assigned the name you specify. Notice that you must give both the DRIVE and NAME keywords. The only valid options that you can give after the DRIVE keyword are DF0:, DF1:, DF2:, or DF3:. You can type any string after NAME, but if you use spaces, you must enclose the whole string in double quotes (").

> **WARNING:** FORMAT formats and initializes a disk as an empty disk. If you use a disk that is not empty, you'll lose the previous contents of the disk.

The name assigned should be unique. It may be one to thirty characters in length and composed of one or more words separated by spaces. If the name is more than one word, you should enclose it in double quotes.

Note: It is not necessary to format a disk if you are about to DISKCOPY to it.

Examples:

```
FORMAT DRIVE df0: NAME "Work disk"
```

formats and initializes the disk in drive "df0" with the name "Work disk".

See also: DISKCOPY,INSTALL,RELABEL

IF

Format: IF[NOT][WARN][ERROR][FAIL][<str>EQ<str>][EXISTS <name>]
Template: IF "NOT/S,WARN/S,ERROR/S,FAIL/S,EQ/K,EXISTS/K"
Purpose: To allow conditionals within command sequences.
Specification:
You can only use this command in an EXECUTE command file. If one or more of the specified conditions is satisfied, IF carries out all the following commands until it finds a corresponding ENDIF or ELSE command; otherwise, if the conditions are not satisfied, it carries out whatever follows a corresponding ELSE command. (ENDIF and ELSE are only useful in command sequences containing IF.) ENDIF terminates an IF command; ELSE provides an alternative if the IF conditions fail. Note that the conditions and commands in IF and ELSE commands can span more than one line before their corresponding ENDIFs.

The following table shows some of the ways you can use the IF, ELSE, and ENDIF commands:

```
IF <condition>
<command>
ENDIF
```

```
IF <condition>
<command>
ELSE
<command>
ENDIF
```

```
IF <condition>
<command>
IF <condition>
<command>
ENDIF
ENDIF
```

Note that ELSE is optional and that nested IFs jump to the nearest ENDIF.

ERROR is only available if you set FAILAT to greater than 10. Similarly, FAIL is only available if you set FAILAT to greater than 20.

Keyword	**Function**
NOT	reverses the result.
WARN	satisfied if previous return code >=5.
ERROR	" >=10.
FAIL	" >=20.
<a>EQ<b>	satisfied if the text of a and b is identical (disregarding case).
EXISTS <file>	satisfied if the file exists.

You can use IF EQ to detect an unset parameter in a command file by using the form:

```
IF <a> EQ ""
```

Examples:

```
IF EXISTS work/prog
TYPE work/prog
ELSE
ECHO "file not found"
ENDIF
```

If the file "work/prog" exists, then AmigaDOS displays it. Otherwise, AmigaDOS displays the message "file not found" and executes the next command in the command sequence.

```
IF ERROR
SKIP errlab
ENDIF
```

If the previous command stopped with a return code >= 10, then AmigaDOS skips the command sequence until you define a label "errlab" with the LAB command.

```
IF ERROR
IF EXISTS fred
ECHO "The file 'fred' exists, but an error occurred anyway."
ENDIF
ENDIF
```

See also: FAILAT,SKIP,LAB,EXECUTE,QUIT

INFO

Format: INFO
Template: INFO
Purpose: To give information about the filing system.
Specification:
The command displays a line of information about each disk unit. This includes the maximum size of the disk, the current used and free space, the number of soft disk errors that have occurred, and the status of the disk.

Examples:

```
INFO
```

```
Unit   Size    Used   Free   Full  Errs  Status      Name
DF1:   880K       2   1756     0%  0     Read/Write  Test-6
DF0:   880K    1081    677    61%  0     Read/Write  AmigaDOS CLI
```

```
Volumes available:
Test-6 [Mounted]
AmigaDOS CLI [Mounted]
```

INSTALL

Format: INSTALL[DRIVE]<drive>
Template: INSTALL "DRIVE/A"
Purpose: To make a formatted disk bootable.
Specification:
The purpose of the INSTALL command is to make a disk bootable (that is, you can use INSTALL to make a disk that starts up your Amiga). To do this, you simply type the name of the drive where you have inserted the disk that you want to become the boot (startup) disk. There are four possible drive names: DF0:,DF1:,DF2:, and DF3:.

Examples:

```
INSTALL df0:
```

makes the disk in drive "df0:" a bootable disk.

JOIN

Format: JOIN <name> <name>[<name>*]AS<name>
Template: JOIN ",,,,,,,,,,,,,,,AS/A/K"
Purpose: To concatenate up to 15 files to form a new file.
Specification:
AmigaDOS copies the specified files in the order you give into the new file. Note that the new file cannot have the same name as any of the input files.

Examples:

```
JOIN part1 part2 AS textfile
```

joins the two files together, placing the result in "textfile". The two original files remain unchanged, while "textfile" contains a copy of "part1" and a copy of "part2".

LAB

Format: LAB <string>
Template: LAB <text>
Purpose: To implement labels in command sequence files.
Specification:
The command ignores any parameters you give. Use LAB to define a label "text" that is looked for by the command SKIP.

Examples:

```
LAB errlab
```

defines the label "errlab" to which SKIP may jump.

See also: SKIP,IF,EXECUTE

LIST

Format: LIST[[DIR]<dir>][P|PAT <pat>][KEYS][DATES][NODATES][TO <name>][S<str>][SINCE <date>][UPTO <date>][QUICK]
Template: LIST "DIR, P = PAT/K, KEYS/S, DATES/S, NODATES/S, TO/K, S/K, SINCE/K,UPTO/K,QUICK/S"
Purpose: To examine and list specified information about a directory or file.
Specification:
If you do not specify a name (the parameter DIR), LIST displays the contents of the current directory. The first parameter LIST accepts is DIR. You have three options. DIR may be a filename, in which case LIST displays the file information for that one file. Secondly, DIR may be a directory name. In this case LIST displays file information for files (and other directories) within the specified directory. Lastly, if you omit the DIR parameter, LIST displays information about files and directories within the current directory (for further details on the current directory, see the CD command).

Note: LIST, unlike DIR, does NOT sort the directory before displaying it.

If no other options are specified, LIST displays:

```
file_name      size protection date time
:comment
```

These fields are defined as follows:

file_name: Name of file or directory.

size: The size of the file in bytes. If there is nothing in the file, this field will state "empty". For directories this entry states "dir".

protection: This specifies the access available for this file; rwed indicates Read, Write, Execute, and Delete.

date and time: The file creation date and time.

comment: This is the comment placed on the file using the FILENOTE command. Note that it is preceded with a colon (:).

Options available:

TO This specifies the file (or device) to output the file listing to. If omitted, the output goes to the current CLI window.

KEYS Displays the block number of each file header or directory.

DATES Displays dates in the form DD-MMM-YY (the default unless you use QUICK).

NODATES Does not display date and time information.

SINCE <date> Displays only files last updated on or after <date>. <date> can be in the form DD-MMM-YY or a day name in the last week (for example, MONDAY) or TODAY or YESTERDAY.

UPTO <date> Displays only files last updated on or before <date>.

P<pat> Searches for files whose names match <pat>.

S<str> Searches for filenames containing substring <str>.

QUICK Just displays the names of files and directories (like the DIR command).

You can specify the range of filenames displayed in two ways. The simplest way is to use the S keyword, which restricts the listing to those files containing the specified substring. To specify a more complicated search expression, use the P or PAT keyword. This is followed by a pattern that matches as described below.

A pattern consists of a number of special characters with special meanings, and any other characters that match them.

The special characters are: '()?%#|

In order to remove the special effect of these characters, preface them with '. Thus '? matches ? and '' matches '.

?	Matches any single character.
%	Matches the null string.
#<p>	Matches zero or more occurrences of the pattern <p>.
<p1><p2>	Matches a sequence of pattern <p1> followed by <p2>.
<p1>|<p2>	Matches if either pattern <p1> or pattern <p2> match.
()	Groups patterns together.

Thus:

LIST PAT A#BC	Matches AC ABC ABBC, and so forth.
LIST PAT A#(B|C)D	Matches AD ABD ABCD, and so forth.
LIST PAT A?B	Matches AAB ABB ACB, and so forth.
LIST PAT A#?B	Matches AB AXXB AZXQB, and so forth.
LIST PAT'?#?'#	Matches ?# ?AB# ??##, and so forth
LIST PAT A(B|%)#C	Matches A ABC ACCC, and so forth.
LIST PAT #(AB)	Matches AB ABAB ABABAB, and so forth.

Examples:

```
LIST
```

displays information about all the files and directories contained in the current directory. For example,

```
File_1
File 2
File.3
:comment      (notice that File.3 has a comment)
File004
```

```
LIST work S new
```

displays information about files in the directory "work" whose names contain the text "new". Note that LIST S produces the response: "Args no good for key" because there is an "S" directory. LIST "s" will list this directory's contents.

```
LIST work P new#?(x|y)
```

examines the directory "work", and displays information about all files that start with the letters "new" and that end with either "x" or "y".

```
LIST QUICK TO outfile
```

sends just the names, one on each line, to the file "outfile". You can then edit the file and insert the command TYPE at the beginning of each line. Then type:

```
EXECUTE outfile
```

to display the files.

See also: DATE,DIR,FILENOTE,PROTECT

MAKEDIR

Format: MAKEDIR<dir>
Template: MAKEDIR "/A"
Purpose: To make a new directory.
Specification:
MAKEDIR creates a directory with the name you specify. The command only creates one directory at a time, so any directories on the path must already exist. The command fails if the directory or a file of the same name already exists in the directory above it in the hierarchy.

Examples:

```
MAKEDIR tests
```

creates a directory "tests" in the current directory.

```
MAKEDIR df1:xyz
```

creates a directory "xyz" in the root directory of disk "df1".

```
MAKEDIR df1:xyz/abc
```

creates a directory "abc" in the parent directory "xyz" on disk "df1". However, "xyz" must exist for this command to work.

See also: DELETE

NEWCLI

Format: NEWCLI[<window>]
Template: NEWCLI "WINDOW"
Purpose: To create a window associated with a new interactive CLI process.

Specification:
AmigaDOS creates a new CLI window. The new window becomes the currently selected process. The new window has the same set directory and prompt string as the one where NEWCLI is executed. Each CLI window is independent, allowing separate input, output, and program execution.

To connect the keyboard to your new CLI, move the mouse to point the cursor at the new window, and press the left mouse button (that is, the Selection Button). You can point at any position on the window when selecting a new CLI.

When you give NEWCLI with no argument, AmigaDOS creates a window of standard size and position. To change the size of the window, move the mouse to point the cursor at the bottom right corner (sizing Gadget), and press the Selection Button. You can then change the window size. To change the position of the window, move the mouse to the Drag Bar, press the left mouse button and move the mouse to where you want the window.

To customize a CLI window, you can give an exact position and size or even a new title on the title bar. The "window" syntax to do this is as follows:

```
CON:x/y/width/height/title
```

where "CON:" denotes a console window, "x" and "y" are the coordinates describing the window's position, "width" and "height" are the size of the window, and "title" is the string you want on the title bar. You need not specify a title string as it is optional, but you must give the final slash (/). All dimensions are in screen pixels.

Examples:

```
NEWCLI
```

creates a new CLI process and makes it the current CLI.

```
NEWCLI CON:10/30/300/100/myCLI
```

creates a new CLI at the position 10,30, of size 300 × 100 pixels, with the title "myCLI".

```
NEWCLI "CON:20/15/300/100/my own CLI"
```

Double quotes allow the title to have spaces. For further information on the console device, CON:, see Section 1.3.6, Understanding Device Names.

Note: Unlike a background process created with the RUN command, a NEWCLI process hangs around after you have created it.

See also: ENDCLI, RUN

PROMPT

Format: PROMPT<prompt>
Template: PROMPT "PROMPT"
Purpose: To change the prompt in the current CLI.
Specification:
If you do not give a parameter, then AmigoDOS resets the prompt to the standard string (">"). Otherwise, the prompt is set to the string you supply. AmigaDOS also accepts one special character combination (%N). This is demonstrated in the example below.

Examples:

```
PROMPT
```

resets the current prompt to ">".

```
PROMPT "%N> "
```

resets the current prompt to "n>", where n is the current process number. AmigaDOS interprets the special character combination %N as the process number.

PROTECT

Format: PROTECT[FILE]<filename>[FLAGS<status>]
Template: PROTECT"FILE,FLAGS/K"
Purpose: To set a file's protection status.
Specification:
PROTECT takes a file and sets its protection status.

The keyword FLAGS takes four options: read (r), write (w), delete (d), and execute (e). To specify these options you type an r, w, d, or e after the name of the file. If you omit an option, PROTECT assumes that you do not require it. For instance, if you give all the options except d, PROTECT ensures that you cannot delete the file. Read, write, and delete can refer to any kind of file. AmigaDOS only pays attention to the delete (d) flag in the current release. Users and user programs, however, can set and test these flags if they wish.

Examples:

```
PROTECT prog1 FLAGS r
```

sets the protection status of program 1 as read only.

```
PROTECT prog2 rwd
```

sets the protection of program 2 as read/write/delete.

See also: LIST

QUIT

Format: QUIT[<returncode>]
Template: QUIT "RC"
Purpose: To exit from a command sequence with a given error code.
Specification:
QUIT reads through the command file and then stops with a return code. The default return code is zero.

Examples:

```
QUIT
```

exits the current command sequence.

```
FAILAT 30
IF ERROR
QUIT 20
ENDIF
```

If the last command was in error, this terminates the command sequence with return code 20.

For more on command sequences, see the specification for the EXECUTE command earlier in this chapter.

See also: EXECUTE,IF,LAB,SKIP

RELABEL

Format: RELABEL[DRIVE]<drive>[NAME]<name>
Template: RELABEL "DRIVE/A,NAME/A"
Purpose: To change the volume name of a disk.

Specification:
RELABEL changes the volume name of a disk to the <name> you specify. Volume names are set initially when you format a disk.

Examples:

```
RELABEL df1: "My other disk"
```

See also: FORMAT

RENAME

Format: RENAME[FROM]<name>[TO|AS]<name>
Template: RENAME "FROM/A,TO=AS/A"
Purpose: To rename a file or directory.
Specification:
RENAME renames the FROM file with the specified TO name. FROM and TO must be filenames on the same disk. The FROM name may refer to a file or to a directory. If the filename refers to a directory, RENAME leaves the contents of the directory unchanged (that is, the directories and files within that directory keep the same contents and names).

Only the name of the directory is changed when you use RENAME. If you rename a directory, or if you use RENAME to give a file another directory name (for example, rename :bill/letter as :mary/letter), AmigaDOS changes the position of the directory, or file, in the filing system hierarchy. Using RENAME is like changing the title of a file and then moving it to another section or drawer in the filing cabinet. Some other systems describe the action as "moving" a file or directory.

The RENAME command will not execute if the only change is the "case" of one or more letters. For example,

```
RENAME fox to Fox
```

does not work.

Note: If you already have a file with exactly the same name as the TO file, RENAME won't work. This should stop you from overwriting your files by accident.

Examples:

```
RENAME work/prog1 AS :arthur/example
```

renames the file "work/prog1" as the file "arthur/example". The root directory must contain "arthur" for this command to work.

RUN

Format: RUN <command>
Template: RUN command+ command. . . . .
Purpose: To execute commands as background processes.
Specification:
RUN creates a noninteractive Command Line Interface (CLI) process and gives it the rest of the command line as input. The background CLI executes the commands and then deletes itself.

The new CLI has the same set directories and command stack size as the CLI where you called RUN.

To separate commands, type a plug sign (+) and press RETURN. RUN interprets the next line after a + (RETURN) as a continuation of the same command line. Thus, you can make up a single command line of several physical lines that each end with a plus sign.

RUN displays the process number of the newly created process.

Examples:

```
RUN COPY :t/0 PRT:+
DELETE :t/0+
ECHO "Printing finished"
```

prints the file ":t/0" by copying it to the line printer device, deletes it, then displays the given message.

```
RUN EXECUTE comseq
```

executes in the background all the commands in the file "comseq".

SEARCH

Format: SEARCH[FROM]<name>|<pat>[SEARCH]<string> [ALL]
Template: SEARCH "FROM,SEARCH/A,ALL/S"
Purpose: To look for a text string you specify in all the files in a directory.
Specification:
SEARCH looks through all the files in the specified directory, and any files in subdirectories if you specify ALL. SEARCH displays any line that contains the

text you specified as SEARCH. It also displays the name of the file currently being searched.

You can also replace the directory FROM with a pattern. (See the command LIST for a full description of patterns.) If you use a pattern, SEARCH only looks through files that match the specified pattern. The name may also contain directories specified as a pattern.

AmigaDOS looks for either upper or lower case of the search string. Note that you must place quotation marks around any text containing a space.

As usual, to abandon the command, press CTRL-C, the attention flag. To abandon the search of the current file and continue on to the next file, if any, press CTRL-D.

Examples:

```
SEARCH SEARCH vflag
```

searches through the files in the current directory looking for the text "vflag".

```
SEARCH df0: "Happy day" ALL
```

looks for files containing the text "Happy day" on the entire disk "df0:".

```
SEARCH test-#? vflag
```

looks for the text "vflag" in all files in the current directory starting with "test-".

SKIP

Format: SKIP <label>
Template: SKIP "LABEL"
Purpose: To perform a jump in a command sequence.
Specification:

You use SKIP in conjunction with LAB. (See LAB for details.) SKIP reads through the command file looking for a label you defined with LAB, without executing any commands.

You can use SKIP either with or without a label; without one, it finds the next unnamed LAB command. With one, it attempts to find a LAB defining a label, as specified. LAB must be the first item on a line of the file. If SKIP does not find the label you specified, the sequence terminates and AmigaDOS displays the following message:

```
label "<label>" not found by Skip
```

SKIP only jumps forward in the command sequence.

Examples:

```
SKIP
```

skips to the next LAB command without a name following it.

```
IF ERROR

SKIP errlab

ENDIF
```

If the last command stopped with a return code >=20, this searches for the label "errlab" later in the command file.

```
FAILAT 100
ASSEM text
IF ERROR
SKIP ERROR
ENDIF
LINK
SKIP DONE
LAB ERROR
ECHO "Error doing Assem"
LAB DONE
ECHO "Next command please"
```

See also: EXECUTE,LAB,IF,FAILAT,QUIT

SORT

Format: SORT[FROM]<name>[[TO]<name>][COLSTART<n>]
Template: SORT "FROM/A,TO/A,COLSTART/K"
Purpose: To sort simple files.
Specification:
This command is a very simple sort package. You can use SORT to sort files although it isn't fast for large files, and it cannot sort files that don't fit into memory.

You specify the source as FROM, and the sorted result goes to the file TO. SORT assumes that FROM is a normal text file where each line is separated

with a carriage return. Each line in the file is sorted into increasing alphabetic order without distinguishing between upper and lower cases.

To alter this in a very limited way, use the COLSTART keyword to specify the first column where the comparison is to take place. SORT then compares the characters on the line from the specified starting position to the end; if the lines still match after this, then the remaining columns from the first to just before the column specified as COLSTART are included in the comparison.

Note: The initial stack size (that is, 4000 bytes) is only suitable for small files of less than 200 lines or so. If you want to sort larger files, you must use the STACK command to increase the stack size; how much you should increase the size is part skill and part guesswork.

> **WARNING:** The Amiga will crash if STACK is too small. If you are not sure, it is better to overestimate the amount you need.

Examples:

```
SORT text TO sorted-text
```

sorts each line of information in "text" alphabetically and places the result in "sorted-text".

```
SORT index TO sorted-index COLSTART 4
```

sorts the file "index", where each record contains the page number in the first three columns and the index entry on the rest of the line, and puts the output in "sorted-index" sorted by the index entry, and matching index entries sorted by page number.

See also: ><, STACK

STACK

Format: STACK[<n>]
Template: STACK "SIZE"
Purpose: To display or set the stack size for commands.
Specification:
When you run a program, it uses a certain amount of stack space. In most cases, the initial stack size, 4000 bytes, is sufficient, but you can alter it using the STACK command. To do this, you type STACK followed by the new stack

value. You specify the value of the stack size in bytes. STACK alone displays the currently set stack size.

The only command that you would normally need to alter the stack size for is the SORT command. Recursive commands such as DIR need an increased stack if you use them on a directory structure more than about six levels deep.

> **WARNING:** The only indication that you have run out of stack is that the Amiga crashes! If you are not sure, it is better to overestimate the amount you need.

Examples:

```
STACK
```

displays the current stack size.

```
STACK 8000
```

sets the stack to 8000 bytes.

See also: RUN, SORT

STATUS

Format: STATUS[<process>][FULL][TCB][SEGS][CLI|ALL]
Template: STATUS "PROCESS, FULL/S,TCB/S,SEGS/S,CLI=ALL/S"
Purpose: To display information about the currently existing CLI processes.
Specification:
STATUS alone lists the numbers of the CLI processes and the program running in each.

PROCESS specifies a process number and only gives information about that process. Otherwise, information is displayed about all processes.

FULL = SEGS + TCB + CLI

SEGS displays the names of the sections on the segment list of each process.

TCB displays information about the priority, stacksize, and global vector size of each process.

For further details on stack and global vector size, see the AmigaDOS Technical Reference in this book.

CLI identifies Command Line Interface processes and displays the section name(s) of the currently loaded command (if any).

Examples:

```
STATUS
```

displays brief information about all processes.

```
STATUS 4 FULL
```

displays full information about process 4.

TYPE

Format: TYPE[FROM]<name>[[TO]<name>][OPT N|H]
Template: TYPE "FROM/A,TO,OPT/K"
Purpose: To type a text file or to type a file out as hexadecimal numbers.
Specification:
TO indicates the output file that you specify; if you omit this, output is to the current output stream, which means, in most cases, that the output goes to the current window.

Tabs that you have given in the file are expanded. However, tabs are not treated as special by TYPE; the console driver processes them. To interrupt output, press CTRL-C. To suspend output, press the space bar or type any other character. To resume output, press RETURN or CTRL-X.

OPT specifies an option to TYPE. The first option to TYPE is "n", which includes line numbers in the output.

The second option you can give TYPE is "h." Use the "h" option to write out each word of the FROM file as a hex number.

Examples:

```
TYPE work/prog
```

displays the file "work/prog".

```
TYPE work/prog OPT n
```

displays the file "work/prog" with line numbers.

```
TYPE obj/prog OPT h
```

displays the code stored in "obj/prog" in hexadecimal.

WAIT

Format: WAIT <n>[SEC|SECS][MIN|MINS][UNTIL <time>]
Template: WAIT ",SEC=SECS/S,MIN=MINS/S,UNTIL/K"
Purpose: To wait for the specified time.
Specification:
You can use WAIT in command sequences or after RUN to wait for a certain period, or to wait until a certain time of day. Unless you specify otherwise, the waiting time is one second.

The parameter should be a number, specifying the number of seconds (or minutes, if MINS is given) to wait.

Use the keyword UNTIL to wait until a specific time of day, given in the format HH:MM.

Examples:

```
WAIT
```

waits 1 second.

```
WAIT 10 MINS
```

waits 10 minutes.

```
WAIT UNTIL 21:15
```

waits until quarter past nine at night.

WHY

Format: WHY
Template: WHY
Purpose: To explain why the previous command failed.
Specification:
Usually when a command fails the screen displays a brief message that something went wrong. This typically includes the name of the file (if that was the problem), but does not go into any more detail. For example, the command

```
COPY fred TO *
```

might fail and display the message

Can't open fred

This could happen for a number of reasons—for example, "fred" might already be a directory, or there might not be enough space on the disk to open the file, or it might be a read-only disk. COPY makes no distinction between these cases, because usually the user knows what is wrong. However, immediately after you come across a command that has failed, you can type WHY and press RETURN to display a much fuller message, describing in detail what went wrong.

Examples:

```
TYPE DFO:
can't open DFO:
WHY
Last command failed because object not of required type
```

WHY gives you a hint about why your command failed. TYPE DFO: failed because AmigaDOS won't let you type a device.

See also: FAULT

2.2 AmigaDOS Developer's Commands

ALINK

Format: ALINK[[FROM|ROOT]<filename>[,<filename>*|+<filename*]] [TO <name>][WITH <name>][LIBRARY|LIB <name>] [MAP <map>][XREF <name>][WIDTH <n>]

Template: ALINK "FROM=ROOT,TO/K,WITH/K,VER/K,LIBRARY=LIB/K, MAP/K,XREF/K,WIDTH/K"

Purpose: To link together sections of code into an executable file.

Specification:

ALINK instructs AmigaDOS to link files together. It also handles automatic library references and builds overlay files. The output from ALINK is a file loaded by the loader and run under the overlay supervisor, if required.

For details and a full specification of the ALINK command, see Chapter 4 of the *AmigaDOS Developer's Manual* in this book.

Examples:

```
ALINK a+b+c TO output
```

links the files "a","b", and "c", producing an output file "output".

ASSEM

Format: ASSEM[PROG|FROM]<prog>[-O <code>][-V <ver>][-L <listing>] [-E] [-C|OPT <opt>][-I <dirlist>]
Template: ASSEM"PROG=FROM/A,-O/K,-V/K,-L/K,-H/K,-E/K,-C=OPT/K,-I/K"
Purpose: To assemble a program in MC68000 assembly language.
Specification:
ASSEM assembles programs in MC68000 assembly language. See Chapter 3 of the *AmigaDOS Developer's Manual* in this book for details.

PROG	is the source file.
-O	is the object file (that is, binary output from the assembler).
-V	is the file for messages. (Unless you specify -V, messages go to the terminal.)
-L	is the listing file.
-C	specifies options to the assembler.
-H	is a header file which can be read as if inserted at the front of the source (like INCLUDE in the source itself).
-I	sets up a list of directories to be searched for included files.
-E	is the file that receives the "equates" directive (EQU) assignments from your source. You use -E to generate a header file containing these directives.

The options you can specify with OPT or -C are as follows:

S	produce a symbol table dump as part of the object file.
X	produce a cross-reference file.
W<size>	set workspace to <size>.

Examples:

```
ASSEM prog.asm TO prog.obj
```

assembles the source program in "prog.asm", placing the result in the file "prog.obj". It writes any error messages to the terminal, but does not produce any assembly listing.

```
ASSEM prog.asm TO prog.obj -h slib -l prog-list
```

assembles the same program to the same output, but includes the file "slib" in the assembly, and places an assembly listing in the file "prog-list".

```
ASSEM foo.asm -o foo.obj opt w8000
```

assembles a *very* small program.

DOWNLOAD

Template: DOWNLOAD "FROM/A,TO/A"
Purpose: To download programs to the Amiga.
Specification:
The command DOWNLOAD downloads programs written on another computer (for example, a Sun) to the Amiga.

To use DOWNLOAD, you must have a BillBoard. Then, to download your linked load file from the Sun to the Amiga, you type on the Sun:

```
binload -p &
```

(this only needs to be done once), then type on the Amiga:

```
download <sun filename> <amiga filename>
```

(Before you boot your Sun, you must make sure that both the BillBoard and Amiga are already on and powered up, otherwise they won't be recognized by the Sun.) The <sun filename> by convention should end with .ld. Once you've done this, to run the program, you type the <amiga filename>.

Note that the command "binload" is not an AmigaDOS command. You use binload on a Sun to load files in binary for downloading to your Amiga.

Note that DOWNLOAD always accesses files on the Sun relative to the directory in which binload was started. If you cannot remember the directory in which binload was started, you must specify the full name. To stop binload on the Sun, you can do a "ps" and then a "kill" on its PID. Note that the soft reset of the computer tells binload to write a message to its standard output (the default being the window where it started). If DOWNLOAD hangs, press CTRL-C to kill it.

Chapter 1 of the *AmigaDOS Developer's Manual* in this book describes in detail how to download programs from an IBM PC to Amiga, from the Sun to the Amiga, and even gives some hints on how to download from unsupported computers.

Examples:

```
binload -p &
download test.ld test.
```

or

```
download /usr/fred/DOS/test.ld test
```

then type the following:

```
test
```

These commands download the specified Sun filenames to the Amiga filenames.

READ

Template: READ "TO/A,SERIAL/S"
Purpose: READ reads data from the parallel port or serial line and stores it in a file.
Specification:
The command READ listens to the parallel port and expects a stream of hexadecimal characters. If you press the SERIAL switch, READ listens, instead, to the serial line. Each hex pair is stored as a byte in memory. READ recognizes Q as the hex stream terminator. READ also recognizes the ASCII digits 0–9 and the capital letters A through F. READ ignores spaces, new lines, and tabs. You must send an ASCII hex digit for every nibble, and you must have an even number of nibbles. When the stream is complete, READ writes the bytes from memory to the disk file you specified.

Note: You can use this command to transfer binary or text files.

> **WARNING 1:** Be careful when READing to the same file twice. READ overwrites the original contents the second time.
>
> **WARNING 2:** You may lose characters if you use high baud rates with the serial connection.

Examples:

```
READ TO df0:new
```

READs to the file "df0:new" from the parallel port.

```
READ new SERIAL
```

READs to the file "new" from the serial line.

2.3 *AmigaDOS Commands Quick Reference Card*

User's Commands

File Utilities

;	comment character.
<>	direct command input and output respectively.
COPY	copies one file to another or copies all the files from one directory to another.
DELETE	deletes up to 10 files or directories.
DIR	shows filenames in a directory.
ED	enters a screen editor for text files.
EDIT	enters a line by line editor.
FILENOTE	attaches a note with a maximum of 80 characters to a specified file.
JOIN	concatenates up to 15 files to form a new file.
LIST	examines and displays detailed information about a file or directory.
MAKEDIR	creates a directory with a specified name.
PROTECT	sets a file's protection status.
RENAME	renames a file or directory.
SEARCH	looks for a specified text string in all the files of a directory.
SORT	sorts simple files.
TYPE	types a file to the screen that you can optionally specify as text or hex.

CLI Control

BREAK	sets attention flags in a given process.
CD	sets a current directory and/or drive.
ENDCLI	ends an interactive CLI process.
NEWCLI	creates a new interactive CLI process.
PROMPT	changes the prompt in the current CLI.
RUN	executes commands as background processes.
STACK	displays or sets the stack size for commands.

STATUS displays information about the CLI processes currently in existence.
WHY explains why a previous command failed.

Command Sequence Control

ECHO displays the message specified in a command argument.
EXECUTE executes a file of commands.
FAILAT fails a command sequence if a program returns an error code greater than or equal to this number.
IF tests specified actions within a command sequence.
LAB defines a label (see SKIP).
QUIT exits from a command sequence with a given error code.
SKIP jumps forward to LAB in a command sequence (see LAB).
WAIT waits for, or until, a specified time.

System and Storage Management

ASSIGN assigns a logical device name to a filing system directory.
DATE displays or sets the system date and time.
DISKCOPY copies the contents of one entire floppy disk to another.
FAULT displays messages corresponding to supplied fault or error codes.
FORMAT formats and initializes a new 3½-inch floppy disk.
INFO gives information about the filing system.
INSTALL makes a formatted disk bootable.
RELABEL changes the volume name of a disk.

Developer's Commands

Development System

ALINK links sections of code into a file for execution (see JOIN).
ASSEM assembles MC68000 language.
DOWNLOAD downloads programs to the Amiga.
READ reads information from the parallel port or serial line and stores it in a file.

Chapter 3

ED—The Screen Editor

This chapter describes how to use the screen editor ED. You can use this program to alter or create text files.

3.1 Introducing ED

You can use the editor ED to create a new file or to alter an existing one. You display text on the screen, and you can scroll it vertically or horizontally, as required.

ED accepts the following template:

```
ED "FROM/A,SIZE/K"
```

For example, to call ED, you type

```
ED fred
```

ED makes an attempt to open the file you have specified as "fred" (that is, the FROM file), and if this succeeds, then ED reads the file into memory and displays the first lines on the screen. Otherwise, ED provides a blank screen, ready for the addition of new information. To alter the text buffer that ED uses to hold the file, you specify a suitable value after the SIZE keyword, for example,

```
ED fred SIZE 45000
```

The initial size is based on the size of the file you edit, with a minimum of 40,000 bytes.

Note: You cannot edit every kind of file with ED. For example, ED does not accept source files containing binary code. To edit files such as these, you should use the editor EDIT.

> **WARNING:** ED always appends a linefeed even if the file does not end with one.

When ED is running, the bottom line of the screen is a message area and command line. Error messages appear here and remain until you give another ED command.

ED commands fall into two categories:

- immediate commands
- extended commands

You use immediate commands in **immediate mode;** you use extended commands in **extended mode.** ED is already in immediate mode when you start editing. To enter extended mode, you press the ESC key. Then, after ED has executed the command line, it returns automatically to immediate mode.

In immediate mode, ED executes commands right away. You specify an immediate command with a single key or control key combination. To indicate a control key combination, you press and hold down the CTRL key while you type the given letter, so that CTRL-M, for example, means hold down CTRL while you type M.

In extended mode, anything you type appears on the command line. ED does not execute commands until you finish the command line. You may type a number of extended commands on a single command line. You may also group any commands together and even get ED to repeat them automatically. Most immediate commands have a matching extended version.

ED attempts to keep the screen up to date. However, if you enter a further command while it is attempting to redraw the display, ED executes the command at once and updates the display when there is time. The current line is always displayed first and is always up to date.

3.2 Immediate Commands

This section describes the type of commands that ED executes immediately. Immediate commands deal with the following:

- cursor control
- text insertion
- text deletion
- text scrolling
- repetition of commands

3.2.1 Cursor Control

To move the cursor one position in any direction, you press the appropriate cursor control key. If the cursor is on the right hand edge of the screen, ED scrolls the text to the left to make the rest of the text visible. ED scrolls vertically a line at a time and horizontally ten characters at a time. You cannot move the cursor off the top or bottom of the file, or off the left hand edge of the text.

CTRL-], that is, CTRL and the square closing bracket "]" takes the cursor to the right hand edge of the current line unless the cursor is already there. When the cursor is already at the right hand edge, CTRL-] moves it back to the left hand edge of the line. The text is scrolled horizontally, if required. In a similar fashion, CTRL-E places the cursor at the start of the first line on the screen unless the cursor is already there. If the cursor is already there, CTRL-E places it at the end of the last line on the screen.

CTRL-T takes the cursor to the start of the next word. CTRL-R takes the cursor to the space following the previous word. In these two cases, the text is scrolled vertically or horizontally, as required.

The TAB key moves the cursor to the next tab position, which is a multiple of the tab setting (initially 3). It does NOT insert TAB characters into the file.

3.2.2 Inserting Text

While in immediate mode, ED is also in INSERT mode so any ordinary characters you type will be inserted at the current cursor position. ED has no type-over mode. To replace a word or line, you must delete the desired

contents and insert new information in its place. Any letter that you type in immediate mode appears at the **current cursor position** unless the line is too long (there is a maximum of 255 characters in a line). If you try to make a line longer than the maximum limit, ED refuses to add another character and displays the following message:

Line too long

However, on shorter lines, ED moves any characters to the right of the cursor to make room for the new text. If the line exceeds the size of the screen, the left hand end of the line disappears from view. Then ED redisplays the end of the line by scrolling the text horizontally. If you move the cursor beyond the end of the line, for example, with the TAB or cursor control keys, ED inserts spaces between the end of the line and any new character you insert.

To split the current line at the cursor and generate a new line, press RETURN. If the cursor is at the end of a line, ED creates a new blank line after the current one. Alternatively, you press CTRL-A to generate a blank line after the current one, with no split of the current line taking place. In either case, the cursor appears on the new line at the position indicated by the left margin (initially, column one).

To ensure that ED gives a carriage return automatically at a certain position on the screen, you can set up a right margin. Once you have done this, whenever you type a line that exceeds that margin, ED ends the line before the last word and moves the word and the cursor down onto a new line. This is called "word wrap." (Note that if you have a line with no spaces, ED won't know where to break the "word" and the automatic margin cannot work properly.) In detail, if you type a character and the cursor is at the end of the line and at the right margin position, then ED automatically generates a new line. Unless the character you typed was a space, ED moves down the half completed word at the end of the line to the newly generated line. However, if you insert some text when the cursor is NOT at the end of a line (that is, with text already to the right of the cursor), then setting a right margin does not work. Initially, the right margin is set up at column 79. You can turn off, or "disable", the right margin with the EX command. (For further details on setting margins, see Section 3.3.1, "Program Control".)

If you type some text in the wrong case (for example, in lower case instead of upper case), you can correct it with CTRL-F, To do this, you move the cursor to point at the letter you want to change and then press CTRL-F. If the letter is in lower case, CTRL-F flips the letter into upper case. On the other hand, if the letter is in upper case, CTRL-F flips it into lower case. However, if the cursor points at something that is not a letter (for example, a space or symbol), CTRL-F does nothing to it.

CTRL-F not only flips letters' cases but it also moves the cursor one place to

the right (and it moves the cursor even if there is no case to flip). So that, after you have changed the case of a letter with CTRL-F, the cursor moves right to point at the next character. If the next character is a letter, you can press CTRL-F again to change its case; you can then repeat the command until you have changed all the letters on the line. (Note that if you continue to press CTRL-F after the last letter on the line, the cursor keeps moving right even though there is nothing left to change.) For example, if you had the line

```
The Walrus and the Carpenter were walking hand in hand
```

and you kept CTRL-F pressed down, the line would become

```
the walrus and the carpenter were walking hand in hand
```

On the other hand, the following line:

```
IF <file> <= x
```

becomes

```
if <FILE> <= X
```

where the letters change case and the symbols remain the same.

3.2.3 Deleting Text

The BACKSPACE key deletes the character to the left of the cursor and moves the cursor one position left unless it is at the beginning of a line. ED scrolls the text, if required. The DEL key deletes the character at the current cursor position without moving the cursor. As with any deletion, characters remaining on the line shift back, and text that was invisible beyond the right hand edge of the screen becomes visible.

The action of CTRL-O depends on the character at the cursor. If this character is a space, then CTRL-O deletes all spaces up to the next nonspace character on the line. Otherwise, it deletes characters from the cursor, and moves text left, until a space occurs.

CTRL-Y deletes all characters from the cursor to the end of the line.

CTRL-B deletes the entire current line. You may use extended commands to delete blocks of text.

3.2.4 Scrolling

Besides vertically scrolling one line at a time by moving the cursor to the edge of the screen, you can vertically scroll the text 12 lines at a time with the control keys CTRL-U and CTRL-D.

CTRL-D moves the cursor to previous lines, while scrolling the text down; CTRL-U scrolls the text up and moves the cursor to lines further on in the file.

CTRL-V refreshes the entire screen, which is useful if another program besides the editor alters the screen. However, in typical use, messages from other processes appear in the window behind the editor window.

3.2.5 Repeating Commands

The editor remembers any extended command line you type. To execute this set of extended commands again at any time, press CTRL-G. In this way, you can set up a search command as an extended command. If the first occurrence of a string is not the one you need, press CTRL-G to repeat the search. You can set up and execute complex sets of editing commands many times.

Note: When you give an extended command as a command group with a repetition count, ED repeats the commands in the group that number of times each time you press CTRL-G. See the next section for more details on extended commands.

3.3 Extended Commands

This section describes the commands available to you in extended mode. These commands cover:

- program control
- block control
- movement
- searching text
- exchanging text
- altering text
- inserting text

To enter extended command mode, press the ESC key. Subsequent input then appears on the command line at the bottom of the screen. You can correct mistakes with BACKSPACE in the normal way. To terminate the command line, press either ESC or RETURN. If you press ESC, the editor remains in extended mode after executing the command line. On the other hand, if you press RETURN, it reverts to immediate mode. To leave the command line

empty, just press RETURN after pressing ESC to go back to immediate mode. In this case, ED returns to immediate command mode.

Extended commands consist of one or two letters, with upper and lower case considered the same. You can give multiple commands on the same command line by separating them with a semicolon. Commands are sometimes followed by an argument, such as a number or a string. A string is a sequence of letters introduced and terminated by a delimiter, which is any character except letters, numbers, space, semicolon, or brackets. Thus, valid strings might be:

```
/happy/
!23 feet!
:Hello!: "½"
```

Most immediate commands have a corresponding extended version. See the Table of Extended Commands at the end of this chapter for a complete list.

3.3.1 Program Control

This section provides a specification of the program control commands X (eXit), Q (Quit), SA (SAve), U (Undo), SH (SHow), ST (Set Tab), SL and SR (Set Left and Set Right), and EX (EXtend).

To instruct the editor to exit, you use the command X. After you have given the exit command, ED writes out the text it is holding in memory to the output, or destination file and then terminates. If you look at this file, you can see that all the changes you made are there.

ED also writes a temporary backup to :T/ED-BACKUP. This backup file remains until you exit from ED again, at which time, ED overwrites the file with a new backup.

To get out of the editor without keeping any changes, you use the Q command. When you use Q, ED terminates immediately without writing to the buffer and discards any changes you have made. Because of this, if you have altered the contents of the file, ED asks you to confirm that you really want to quit.

A further command lets you to take a "snapshot" copy of the file without coming out of ED. This is the SA command. SA saves the text to a named file or, in the absence of a named file, to the current file. For example,

```
SA !:doc/savedtext!
```

or

```
SA
```

SA is particularly useful in geographical areas subject to power failure or surge.

Hint: SA followed by Q is equivalent to the X command.

If you make any alterations between the SA and the Q commands, the following message appears:

Edits will be lost—type Y to confirm:

If you have made no alterations, ED quits immediately with the contents of your source file unchanged. SA is also useful because it allows you to specify a filename other than the current one. It is therefore possible to make copies at different stages and place them in different files or directories.

To undo the last change, you use the U command. The editor makes a copy of the line the cursor is on, and then it modifies this copy whenever you add or delete characters. ED puts the changed copy back into the file when you move the cursor off the current line (either by cursor control, or by deleting or inserting a line). ED also replaces the copy when it performs any scrolling either vertically or horizontally. The U command discards the changed copy and uses the old version of the current line instead.

> **WARNING:** ED does not undo a line deletion. Once you have moved from the current line, the U command cannot fix the mess you have got yourself into.

To show the current state of the editor, you use the SH command. The screen displays information such as the value of tab stops, current margins, block marks, and the name of the file being edited.

Tabs are initially set at every three columns. To change the current setting of tabs, you use the ST command followed by a number "n", which sets tabs at every "n" column.

To set the left margin and right margin, you use the SL and SR commands, again followed by a number indicating the column position. The left margin should not be set beyond the width of the screen.

To extend margins, you use the EX command. Once you have given EX, ED takes no account of the right margin on the current line. Once you move the cursor from the current line, ED turns the margins on again.

3.3.2 Block Control

To move, insert, or delete text, you use the block control commands described in this section.

You can identify a block of text with the BS (Block Start) and BE (Block End)

commands. To do this, move the cursor to anywhere on the first line that you want in the block and give the BS command. Then, move the cursor to the last line that you want in the block, using the cursor control commands or a search command, and give the BE command to mark the end of the block.

Note: Once you have defined a block with BS and BE, if you make ANY change to the text, the start and end of the block become undefined once more. The only exception to this is if you use IB (Insert Block).

To identify one line as the current block, move to the line you want, press ESC, and type:

```
BS;BE
```

The current line then becomes the current block.

Note: You cannot start or finish a block in the middle of a line. To do this, you must first split the line by pressing RETURN.

Once you have identified a block, you can move a copy of it into another part of the file with the IB (Insert Block) command. When you give the IB command, ED inserts a copy of the block immediately after the current line. You can insert more than one copy of the block, as it remains defined until you change the text, or delete the block.

To delete a block, you use the DB (Delete Block) command. DB deletes the block of text you defined with the BS and BE commands. However, when you have deleted the block, the block start and end values become undefined. This means that you CANNOT delete a block and then insert a copy of it (DB followed by IB); however, you can insert a copy of the block and then delete the block (IB followed by DB).

You can also use block marks to remember a place in a file. The SB (Show Block) command resets the screen window on the file so that the first line in the block is at the top of the screen.

To write a block to another file, you use the WB command (Write Block). This command takes a string that represents a filename. For example,

```
WB !:doc/example!
```

writes the contents of the block to the file "example" in the directory ":doc". Remember: if you use the filename-divider slash (/) to separate directories and files, you should not use slash as a delimiter. ED then creates a file with the name that you specified, possibly destroying a previous file with that name, and finally writes the buffer to it.

To insert a file into the current file, you use the IF command (Insert File). ED reads into memory the file with the name you gave as the argument string to IF, at the point immediately following the current line. For example,

```
IF !:doc/example!
```

inserts the file :doc/example into the current file beginning immediately after the current line.

3.3.3 *Moving the Current Cursor Position*

The command T moves the cursor to the top of the file, so that the first line in the file is the first line on the screen. The B command moves the cursor to the bottom of the file, so that the last line in the file is the bottom line on the screen.

The commands N and P move the cursor to the start of the next line and previous line, respectively. The commands CL and CR move the cursor one place to the left or one place to the right, while CE places the cursor at the end of the current line, and CS places it at the start.

The command M moves the cursor to a specific line. To move, you type M followed by the line number of the line you want as the new current line. For example,

```
M 503
```

moves the cursor to the five hundred and third line in the file. The M command is a quick way of reaching a known position in your file. You can, for instance, move to the correct line in your file by giving a repeat count to the N command, but it is much slower.

3.3.4 *Searching and Exchanging*

Alternatively you can move the screen window to a particular context with the command F (Find) followed by a string that represents the text to be located. The search starts at one place beyond the current cursor position and continues forward through the file. If the string is found, the cursor appears at the start of the located string. The string must be in quotes (or other delimiters "/", ".", "!", and so on). In order for a match to occur the strings must be of the same case, unless the UC command is used (see below).

To search backward through the text, you use the command BF (Backward Find) in the same way as F. BF finds the last occurrence of the string before the current cursor position. (That is, BF looks for the string to the left of the cursor and then through all the lines back to the beginning of the file.) To find the earliest occurrence, you use T (Top-of-file) followed by F. To find the last occurrence, you use B (Bottom-of-file) followed by BF.

The E (Exchange) command takes two strings separated with delimiter characters and exchanges the first string for the last. So, for example,

```
E /wombat/zebra/
```

would change the next occurrence of the text "wombat" to "zebra". The editor starts searching for the first string at the current cursor position and continues through the file. After the exchange is completed, the cursor moves to the end of the exchanged text.

You can specify empty strings by typing two delimiters with nothing between them. If the first, or "search", string is empty, the editor inserts the second string at the current cursor position. If the second string is empty, the next occurrence of the search string is exchanged for nothing (that is, the search string is deleted).

Note: ED ignores margin settings while you are exchanging text.

The EQ command (Exchange and Query) is a variant on the E command. When you use EQ, ED asks you whether you want the exchange to take place. This is useful when you want the exchange to take place in some circumstances, but not in others. For example, after typing

```
EQ /wombat/zebra/
```

the following message

Exchange?

appears on the command line. If you respond with an N, then the cursor moves past the search string; otherwise, if you type Y, the change takes place as normal. You usually only give EQ in repeated groups.

The search and exchange commands usually make a distinction between upper and lower case while making the search. To tell all subsequent searches not to make any distinction between upper and lower case, you use the UC command. Once you have given UC, the search string "wombat" matches "Wombat", "WOMBAT", "WoMbAt", and so on. To have ED distinguish between upper and lower case again, you use LC.

3.3.5 Altering Text

You cannot use the E command to insert a new line into the text. You use the I and A commands instead. Follow the I command (Insert before) with a string that you want to make into a new line. ED inserts this new line before the current line. For example,

```
I /Insert this BEFORE the current line/
```

inserts the string "Insert this BEFORE the current line" as a new, separate line Before the line containing the cursor. You use the A command (insert After) in the same way except that ED inserts the new line after the current line. That is,

```
A /Insert this AFTER the current line/
```

inserts the string "Insert this AFTER the current line" as a new line After the line containing the cursor.

To split the current line at the cursor position, you use the S command. S in extended mode is just like pressing RETURN in immediate mode (see Section 3.2.2 for further details on splitting lines).

The J command joins the next line to the end of the current one.

The D command deletes the current line in the same way as CTRL-B in immediate mode. The DC command deletes the character above the cursor in the same way as DEL.

3.3.6 Repeating Commands

To repeat any command a certain number of times, precede it with the desired number. For example,

```
4 E /slithy/brillig/
```

changes the next four occurrences of "slithy" to "brillig". ED verifies the screen after each command. You use the RP (Repeat) command to repeat a command until ED returns an error, such as reaching the end of the file. For example,

```
T; RP E /slithy/brillig/
```

changes all occurrences of "slithy" to "brillig". Notice that you need the T command to ensure that ALL occurrences of "slithy" are changed, otherwise only those after the current position are changed.

To execute command groups repeatedly, you can group the commands together in parentheses. You can also nest command groups within command groups. For example,

```
RP ( F /bandersnatch/; 3 A/ / )
```

inserts three blank lines (copies of the null string) after every line containing "bandersnatch". Notice that this command line only works from the cursor to the end of the file. To apply the command to every line in the file, you should first move to the top of the file.

Note that some commands are possible, but silly. For example,

```
RP SR 60
```

sets the right margin to 60 *ad infinitum*. However, to interrupt any sequence of extended commands, and particularly repeated ones, you type any character while the commands are taking place. If an error occurs, ED abandons the command sequence.

Quick Reference Card

Special Key Mappings

Command	**Action**
BACKSPACE	Delete character to left of cursor
DEL	Delete character at cursor
ESC	Enter extended command mode
RETURN	Split line at cursor and create a new line
TAB	Move cursor right to next tab position (does NOT insert a TAB character)
<up-arrow>	Move cursor up
<down-arrow>	Move cursor down
<left-arrow>	Move cursor left
<right-arrow>	Move cursor right

Immediate Commands

Command	**Action**
CTRL-A	Insert line
CTRL-B	Delete line
CTRL-D	Scroll text down
CTRL-E	Move to top or bottom of screen
CTRL-F	Flip case
CTRL-G	Repeat last extended command line
CTRL-H	Delete character left of cursor (BACKSPACE)
CTRL-I	Move cursor right to next tab position
CTRL-M	RETURN
CTRL-O	Delete word or spaces
CTRL-R	Cursor to end of previous word
CTRL-T	Cursor to start of next word
CTRL-U	Scroll text up
CTRL-V	Verify screen
CTRL-Y	Delete to end of line

CTRL-[	Escape (enter extended mode)
CTRL-]	Cursor to end or start of line

Extended Commands

This is a full list of extended commands including those that are merely extended versions of immediate commands. In the list, /s/ indicates a string, /s/t/ indicates two exchange strings, and "n" indicates a number.

Command	**Action**
A /s/	Insert line after current line
B	Move to bottom of file
BE	Block end at cursor
BF /s/	Backward find
BS	Block start at cursor
CE	Move cursor to end of line
CL	Move cursor one position left
CR	Move cursor one position right
CS	Move cursor to start of line
D	Delete current line
DB	Delete block
DC	Delete character at cursor
E /s/t/	Exchange "s" into "t"
EQ /s/t/	Exchange but query first
EX	Extend right margin
F /s/	Find string "s"
I /s/	Insert line before current
IB	Insert copy of block
IF /s/	Insert file "s"
J	Join current line with next
LC	Distinguish between upper and lower case in searches
M n	Move to line number "n"
N	Move to start of next line
P	Move to start of previous line
Q	Quit without saving text
RP	Repeat until error
S	Split line at cursor
SA	Save text to file
SB	Show block on screen
SH	Show information
SL n	Set left margin
SR n	Set right margin

Command	Action
ST n	Set tab distance
T	Move to top of file
U	Undo changes on current line
UC	Equate U/C and l/c in searches
WB /s/	Write block to file "s"
X	Exit, writing text into memory

Chapter 4

EDIT—The Line Editor

This chapter describes in detail how to use the line editor EDIT. The first part introduces the reader to the editor. The second part gives a complete specification of EDIT. There is a quick reference card containing all the EDIT commands at the end of the chapter.

4.1 Introducing EDIT

EDIT is a text editor that processes sequential files line by line under the control of editing commands. EDIT moves through the input, or source file, passing each line (after any possible alterations) to a sequential output file, the destination file. An EDIT run, therefore, makes a copy of the source file that contains any changes that you requested with the editing commands.

Although EDIT usually processes the source file in a forward sequential manner, it has the capability to move backward a limited number of lines. This is possible because EDIT doesn't write the lines that have been passed to the destination file immediately, but holds them instead in an output queue. The size of this queue depends on the amount of memory available. If you want to hold more information in memory, you can select the EDIT option, OPT, described in the next section, to increase the amount.

You can make more than one pass through the text.

The EDIT commands let you

a) change parts of the source,
b) output parts of the source to other destinations, and
c) insert material from other sources.

4.1.1 Calling EDIT

This section describes the format of the arguments you can give every time you call the EDIT command. EDIT expects the following arguments:

```
FROM/A,TO,WITH/K,VER/K,OPT/K
```

The command template described in Chapter 1 is a method of defining the syntax for each command. AmigaDOS accepts command arguments according to the format described in the command template. For example, some arguments are optional, some must appear with a keyword, and others do not need keywords because they appear only in a specific position. Arguments with a following /A (like FROM) must appear, but you do not have to type the keyword. Arguments with just a following /K (such as WITH, VER, and OPT) are optional, but you must type the keyword to specify them. Arguments without a following / (TO, for example), are optional. AmigaDOS recognizes arguments without a following slash (/) by their position alone. If you forget the syntax for EDIT, type:

```
EDIT ?
```

and AmigaDOS displays the full template on the screen. (For more details on using commands, see Chapters 1 and 2 of this manual.)

Using another method of description, the command syntax for EDIT is as follows:

```
[FROM] <file> [[TO] <file>] [WITH <file>] [VER <file>] [OPT Pn|Wn|Pn
Wn]
```

The argument FROM represents the source file that you want to edit. The argument must appear, but the keyword itself is optional. (That is, AmigaDOS accepts the FROM file by its position.) It does not require you to type the keyword FROM as well.

The TO file represents the destination file. This is the file where EDIT sends the output including the editing changes. If you omit the TO argument, EDIT uses a temporary file that it renames as the FROM file when editing is complete. If you give the EDIT command STOP, this renaming does not take place, and the original FROM file is untouched.

The WITH keyword represents the file containing the editing commands. If you omit the WITH argument, EDIT reads from the terminal.

The VER keyword represents the file where EDIT sends error messages and line verifications. If you omit the VER argument, EDIT uses the terminal.

You can use the OPT keyword to specify options to EDIT. Valid options are P<n>, which sets the number of previous lines available to the integer <n>, and W<n>, which sets the maximum line length handled to <n> characters. Unless you specify otherwise, AmigaDOS sets the options P40W120.

You can use OPT to increase, or decrease, the size of available memory. EDIT uses P*W (that is, the number of previous lines multiplied by the line width) to determine the available memory. To change the memory size, adjust the P and W numbers. P50 allocates more memory than usual; P30 allocates less memory than usual.

Here are some examples of how you can call EDIT:

```
EDIT program1 TO program1_new WITH edit_commands

EDIT program1 OPT P50W240

EDIT program1 VER ver_file
```

Note: Unlike ED, you cannot use EDIT to create a new file. If you attempt to create a new file, AmigaDOS returns an error because it cannot find the new file in the current directory.

4.1.2 Using EDIT Commands

This section introduces some of the basic EDIT commands omitting many of the advanced features. A complete description of the command syntax and of all commands appears in Section 4.2, "A Complete Specification of EDIT."

4.1.2.1 The Current Line

As EDIT reads lines from the source and writes them to the destination, the line that it has "in its hand" at any time is called the current line. EDIT makes all the textual changes to the current line. EDIT always inserts new lines before the current line. When you first enter EDIT, the current line is the first line of the source.

4.1.2.2 Line Numbers

EDIT assigns each line in the source a unique line number. This line number is not part of the information stored in the file, but EDIT computes it by counting the lines as they are read. When you're using EDIT, you can refer to a specific line by using its line number. A line that has been read retains its original line number all the time it is in main memory, even when you delete lines before or after it, or insert some extra lines. The line numbers remain unchanged until you rewind the file, or until you renumber the lines with the = command. EDIT assigns the line numbers each time you enter the file. The line numbers, therefore, may not be the same when you re-enter.

4.1.2.3 Selecting a Current Line

To select a current line in EDIT, you can use one of three methods:

a) counting lines,
b) specifying the context, or
c) specifying the line number.

These three methods are described below.

By Line Counting

The N and P commands allow you to move to the next or previous lines. If you give a number before the N or P command, you can move that number of lines forward or backward. To move forward to the next line, type:

```
N
```

For any EDIT command, you can type either upper or lower case letters.
To move four lines forward, type:

```
4N
```

to make the fourth line from the current line your new current line.
To move back to a line above the current line, type:

```
P
```

The P command also takes a number. For example, type:

```
4P
```

This makes the fourth line above the current line your new current line. It is only possible to go back to previous lines that EDIT has not yet written to the output. EDIT usually lets you go back 40 lines. To be able to move back more than this, you specify more previous lines with the P option when you enter EDIT (see Section 4.1.1 earlier in this chapter for further details on the P option).

Moving to a Specific Line Number

The M command allows you to select a new current line by specifying its line number. You type the M command and the desired line number. For example, the command M45 tells EDIT to Move to line 45. If you are beyond line 45, this command moves back to it provided it is still in main memory.

You can combine the specific line number and line counting commands. For example,

```
M12; 3N
```

To separate consecutive commands on the same line, type; (a semicolon).

By Context

You use the F command (Find) to select a current line by context. For example,

```
F/Jabberwocky/
```

means to find the line containing "Jabberwocky". The search starts at the current line and moves forward through the source until the required line is found. If EDIT reaches the end of the source without finding a matching line, it displays the following message:

SOURCE EXHAUSTED

It is also possible to search backward by using the BF command (Backward Find). For example,

```
BF/gyre and gimble/
```

BF also starts with the current line, but EDIT moves backward until it finds the desired line. If EDIT reaches the head of the output queue without finding a matching line, it displays the following message:

NO MORE PREVIOUS LINES

Notice that in the examples above, the desired text (Jabberwocky and gyre and gimble) is enclosed in matching single slashes (/). This desired text is called a character string. The characters you use to indicate the beginning and end of the character string are called delimiter characters. In the examples above, / was used as the delimiter. A number of special characters such as : . , and * are available for use as delimiters; naturally, the string itself must not contain the delimiter character. EDIT ignores the spaces between the command name and the first delimiter, but considers spaces within the string as significant, since it matches the context exactly. For example,

```
F /tum tum tree/
```

does not find "tum-tum tree" or "tum tum tree".

If you use an F command with no argument, EDIT repeats the previous search. For example,

```
F/jubjub bird/; N; F
```

finds the second occurrence of a line containing "jubjub bird". The N command between the two F commands is necessary because an F command always starts by searching the current line. If you omitted N, the second F would find the same line as the first.

4.1.2.4 Qualifiers

The basic form of the F command described above finds a line that contains the given string anywhere in its length. To restrict the search to the beginning or the end of lines, you can place one of the letters B or E in front of the string. In this case, you must type one or more spaces after F. For example,

```
F B/slithy toves/
```

means Find the line Beginning with "slithy toves", while

```
F E/bandersnatch/
```

means Find the line Ending with "bandersnatch". As well as putting further conditions on the context required, the use of B or E speeds up the search, as EDIT only needs to consider part of each line.

B and E as used above are examples of **qualifiers**, and the whole argument is

called a **qualified string.** A number of other qualifiers are also available. For example,

```
F P/a-sitting on a gate/
```

means Find the next line containing Precisely the text "a-sitting on a gate". The required line must contain no other characters, either before or after the given string. That is to say, when you give this command, EDIT finds the next line containing:

a-sitting on a gate

However, EDIT does not find the line:

a-sitting on a gate.

To find an empty line (Precisely nothing), you can use an empty string with the P qualifier, for example,

```
F P//
```

You can give more than one qualifier in any order.

4.1.2.5 Making Changes to the Current Line

This section describes how to use the E, A, and B commands to alter the text on your current line.

Exchanging strings

The E command Exchanges one string of characters in the line for another. For example:

```
E/Wonderland/Looking Glass/
```

removes the string "Wonderland" from the current line, and replaces it with "Looking Glass". Note that you use a single central delimiter to separate the two strings. To delete parts of the line (exchange text for nothing), you can use a null second string, as follows:

```
E/monstrous crow//
```

To add new material to the line, you can use the A or B commands. The A command inserts its second string After the first occurrence of the first string

on the current line. Similarly, the B command insert its second string Before the first occurrence of the first string on the current line. For example, if the current line contained

If seven maids with seven mops

then the following command sequence:

A/seven/ty/; B L/seven/sixty-/

would turn it into:

If seventy maids with sixty-seven mops

If you had omitted the L qualifier from the B command above, the result would have been:

If sixty-seventy maids with seven mops

because the search for a string usually proceeds from left to right, and EDIT uses the first occurrence that it finds. You use the qualifier L to specify that the search should proceed Leftward. The L qualifier forces the command that it qualifies to act on the Last occurrence of its first argument.

If the first string in an A, B, or E command is empty, EDIT inserts the second string at the beginning or the end of the line. To further qualify the position of the second string, you use or omit the L or the E qualifiers.

If you give EDIT an A, B, or E command on a line that does not match the qualified string given as the first argument, the following message appears either on the screen or in a verification file that you specified when you entered EDIT.

NO MATCH

See Section 4.1.1, "Calling EDIT" for details on the verification file.

4.1.2.6 Deleting Whole Lines

This section describes how to remove lines of text from your file. To delete a range of lines, you can specify their line numbers in a D command. To use the D command, type D and the line number. If you type a space and a second number after D, EDIT removes all the lines from the first line number to the last. For example,

deletes lines 97 to 104 inclusive, leaving line 105 as the new current line. To delete the current line, type D without a qualifying number. For example,

```
F/plum cake/; D
```

deletes the line containing "plum cake", and the line following it becomes the new current line. You can combine a qualified search with a delete command, as follows:

```
F B/The/; 4D
```

This command sequence deletes four lines, the first of which is the line beginning with "The".

You can also type a period (.) or an asterisk (*) instead of line numbers. To refer to the current line, type a period. To refer to the end-of-file, type an asterisk. For example,

```
D. *
```

deletes the rest of the source including the current line.

4.1.2.7 Inserting New Lines

This section describes how to insert text into your file with EDIT. To insert one or more lines of new material BEFORE a given line, you use the I command. You can give the I command alone or with a line number, a period (.), or an asterisk (*). EDIT inserts text before the current line if you give I on its own, or follow it with a period (.). If you type an asterisk (*) after I, your text is inserted at the end of the file (that is, before the end-of-file line). Any text that you type is inserted before the line you specified.

To indicate the end of your insertion, press RETURN, type Z, and press RETURN again. For example,

```
I 468
The little fishes of the sea,
They sent an answer back to me.
Z
```

inserts the two lines of text before line 468.

If you omit the line number from the command, EDIT inserts the new material before the current line. For example,

```
F/corkscrew/; I
He said, "I'll go and wake them, if ..."
Z
```

This multiple command finds the line containing "corkscrew" (which then becomes the current line) and inserts the specified new line.

After an I command containing a line number, the current line is the line of that number; otherwise, the current line is unchanged.

To insert material at the end of the file, type I*.

To save you typing, EDIT provides the R (Replace) command, the exact equivalent of typing DI (D for Delete followed by I for Insert). For example,

```
R19 26
In winter when the fields are white
Z
```

deletes lines 19 to 26 inclusive, then inserts the new material before line 27, which becomes the current line.

4.1.2.8 Command Repetition

You can also use individual repeat counts as shown in the examples for N and D above with many EDIT commands. In addition, you can repeat a collection of commands by forming them into a command group using parentheses as follows:

```
6(F P//; D)
```

deletes the next six blank lines in the source. Command groups may not extend over more than one line of command input.

4.1.3 Leaving EDIT

To end an EDIT session, you use the command W (for Windup). EDIT "winds through" to the end of the source, copying it to the destination, and exits. Unless you specify a TO file, EDIT renames the temporary output file as the FROM filename.

EDIT can accept commands from a number of command sources. In the simplest case, EDIT accepts commands directly from the terminal (that is, from the keyboard); this is called the **primary command level.** EDIT can, however, accept commands from other sources, for example, **command files** or WITH files.

You can call command files from within EDIT, and further command files from within command files, with the C command, so that each nested command file becomes a separate command level. EDIT stops executing the commands in the command file when it comes to the end of the command file, or when it finds a Q. When EDIT receives a Q command in a command file, or it comes to the end of the file, it immediately stops executing commands from that file, and reverts to the previous command level. If EDIT finds a Q command in a nested command file, it returns to executing commands in the command file at the level above. If you stop editing at the primary command level, by typing Q, or if EDIT finds a Q in a WITH file, then EDIT winds up and exits in the same way as it does with W.

The command STOP terminates EDIT without any further processing. In particular, EDIT does not write out any pending lines of output still in memory so that the destination file is incomplete. If you only specify the FROM argument, EDIT does NOT overwrite the source file with the (incomplete) edited file. You should only use STOP if you do not need the output from the EDIT run.

EDIT writes a temporary backup to :T/ED-BACKUP when you exit with the W or Q commands. This backup file remains until you exit from EDIT with these commands again, whereupon EDIT overwrites the file with a new backup. If you use the STOP command, EDIT does not write to this file.

4.1.4 A Combined Example: Pulling It All Together

You can meet most simple editing requirements with the commands already described. This section presents an example that uses several commands. The text in italics following the editing commands in the example is a comment. You are not meant to type these comments; EDIT does not allow comments in command lines.

To make it easier for you to follow what is happening, we have included this file as "Edit_Sample" on your accompanying disk.

Take the following source text (with line numbers):

```
1 Tweedledee and Tweedledum
2 agreed to a battle,
3 For Tweedledum said Tweedledee
4 ad spoiled his nice new rattle.
5
6 As black as a tar barrel
7 Which frightened both the heroes so
8 They quite forgot their quorell
```

Execute these EDIT commands:

```
M1; E/dum/dee/; E/dee/dum/      <the order of the E commands matters!>
N; E/a/A/; B /a /have /         <now at line 2>
F B/ad/; B//H/                  <H at line start>
F P//; N; I                     <before line after blank one>
Just then flew down a monstrous
crow,
Z
M6; 2(A L//,/; N)               <commas at end of lines>
F/quore/; E/quorell/quarrel./
                                <F is, in fact, redundant>
W                               <Windup>
```

The following text (with new line numbers) is the result.

```
1 Tweedledum and Tweedledee
2 Agreed to have a battle,
3 For Tweedledum said Tweedledee
4 Had spoiled his nice new rattle.
5
6 Just then flew down a monstrous crow,
7 As black as a tar barrel,
8 Which frightened both the heroes so,
9 They quite forgot their quarrel.
```

Note: If you experiment with editing this source file, you'll find that you don't have to use the commands in the example above. For instance, on the second line, you could use the following command:

```
E/a/have a/
```

to produce the same result. In other words, E Exchanges "a" for "have a", and B places "have" Before "a" to produce "have a".

4.2 A Complete Specification of EDIT

After reading the first part of this chapter on the basic features of EDIT, you should be able to use the editor in a simple way. The rest of this chapter is a reference section that provides a full specification of all the features of EDIT. You may need to consult this section if you have any problems when editing or if you want to use EDIT in a more sophisticated way.

The features described in this section are as follows:

- Command syntax
- Control of Command, Input, and Output Files
- Processing EDIT
- Functional Groupings of EDIT Commands
- Line Windows
- String Operations on the Current Line
- Miscellaneous Current Line Commands
- Inspecting Parts of the Source: The Type Commands
- Control of Command, Input, and Output Files
- Loops
- Global Operations
- Displaying the Program State
- Terminating an EDIT Run
- Current Line Verification
- Miscellaneous Commands
- Abandoning Interactive Editing

4.2.1 Command Syntax

EDIT commands consist of a command name followed by zero or more arguments. One or more space characters may optionally appear between a command name and the first argument, between non-string arguments, and between commands. A space character is only necessary in these places to separate successive items otherwise treated as one (for example, two numbers).

EDIT understands that a command is finished in any of the following ways: when you press RETURN; when EDIT reaches the end of the command arguments; or when EDIT reads a semicolon (;), or closing parentheses ()), that you have typed.

You use parentheses to delimit command groups.

To separate commands that appear on the same line of input, you type a semicolon. This is only strictly necessary in cases of ambiguity where a command has a variable number of arguments. EDIT always tries to read the longest possible command.

Except where they appear as part of a character string, EDIT thinks of upper and lower case letters as the same.

4.2.1.1 Command Names

A command name is either a sequence of letters or a single special character (for example, #). An alphabetic command name ends with any nonletter; only the first four lettters of the name are significant. One or more spaces may appear between command names and their arguments; EDIT requires at least

one space when an argument starting with a letter follows an alphabetic name.

4.2.1.2 Arguments

The following sections describe the six different types of arguments you can use with EDIT commands:

- strings
- qualified strings
- search expressions
- numbers
- switch values
- command groups

4.2.1.3 Strings

A string is a sequence of up to 80 characters enclosed in delimiters. You may use an empty (null) string. (A null string is exactly what it sounds like: a nonstring, that is, delimiters enclosing nothing, for example, //.) The character that you decide to use to delimit a particular string may not appear in the string. The terminating delimiter may be omitted if it is immediately followed by the end of the command line.

The following characters are available for use as delimiters:

```
/ • + – , ? : *
```

that is, common English punctuation characters (except ;) and the four arithmetic operators.

Here are some examples of strings:

```
/A/
*Menai Bridge*
??
+String with final delimiter omitted
```

4.2.1.4 Multiple Strings

Commands that take two string arguments use the same delimiter for both and do not double it between the arguments. An example is the A command:

```
A /King/The Red /
```

For all such commands the second string specifies replacement text. If you omit the second string, EDIT uses the null string. If you do this with the A and

B command, then nothing happens because you have asked EDIT to insert nothing after or before the first string. However, if you omit the second string after an E command, EDIT deletes the first string.

4.2.1.5 Qualified Strings

Commands that search for contexts, either in the current line or scanning through the source, specify the context with qualified strings. A qualified string is a string preceded by zero or more qualifiers. The qualifiers are single letters. They may appear in any order. For example,

```
BU/Abc/
```

Spaces may not appear between the qualifiers. You may finish a list of qualifiers with any delimiter character. The available qualifiers are B (Beginning), E (End), L (Left or Last), P (Precisely), and U (Uppercase).

4.2.1.6 Search Expressions

Commands that search for a particular line in the source take a search expression as an argument. A search expression is a single qualified string. For example,

```
F B/Tweedle/
```

tells EDIT to look for a line beginning with the string "Tweedle".

4.2.1.7 Numbers

A number is a sequence of decimal digits. Line numbers are a special form of number and must always be greater than zero. Wherever a line number appears, the characters "." and "*" may appear instead. A period represents the current line, and an asterisk represents the last line at the end of the source file. For example,

```
M*
```

instructs EDIT to move to the end of the source file.

4.2.1.8 Switch Values

Commands that alter EDIT switches take a single character as an argument. The character must be either a + or –. For example, in

```
V–
```

the minus sign (–) indicates that EDIT should turn off the verification. If you

then type V+, EDIT turns the verification on again. In this case, you can consider + as "on" and – as "off".

4.2.1.9 Command Groups

To make a number of individual EDIT commands into a command group, you can enclose them in parentheses. For example, the following line:

```
(f/Walrus/;e/Walrus/Large Marine Mammal/)
```

finds the next occurrence of 'Walrus' and changes it to 'Large Marine Mammal'. Command groups, however, may not span more than one line of input. For instance, if you type a command group that is longer than one line, EDIT only accepts the commands up to the end of the first line. Then, because EDIT does not find a closing parenthesis at the end of that line, it displays the following error message:

Unmatched parenthesis

Note that it is only necessary to use parentheses when you intend to repeat a command group more than once.

4.2.1.10 Command Repetition

EDIT accepts many commands preceded by an unsigned decimal number to indicate repetition. For example

```
24N
```

If you give a value of zero, then EDIT executes the command indefinitely (or until end-of-file is reached). For example, if you type

```
0(e /dum/dee/;n)
```

EDIT exchanges every occurrence of "dum" for "dee" to the end of the file.

You can specify repeat counts for command groups in the same way as for individual commands:

```
12(F/handsome/; E/handsome/hansom/; 3N)
```

4.2.2 Processing EDIT

This section describes what happens when you run EDIT. It gives details about where input comes from and where the output goes, what should appear on your screen, and what should eventually appear in your file after you have run EDIT.

4.2.2.1 Prompts

When EDIT is being run interactively, that is, with both the command file connected to the keyboard and the verification file connected to a window, it displays a prompt when it is ready to read a new line of commands. Although, if the last command of the previous line caused verification output, EDIT does not return a prompt.

If you turn the verification switch V on, EDIT verifies the current line in place of a prompt in the following circumstances:

- if it has not already verified the current line,
- if you have made any changes to the line since it was last verified, or
- if you have changed the position of the operational window.

Otherwise, when EDIT does not verify the current line, it displays a colon character (:) to indicate that it is ready for a new line of commands. This colon is the usual EDIT prompt.

EDIT never gives prompts when you are inserting lines.

4.2.2.2 The Current Line

As EDIT reads lines from the source file and writes them to the destination file, the line that EDIT has in its hand at any time is called the current line. Every command that you type refers to the current line. EDIT inserts new lines before the current line. When you start editing with EDIT, the current line is the first line of the source.

4.2.2.3 Line Numbers

EDIT identifies each line in the source by a unique line number. This is not part of the information stored in the file. EDIT computes these numbers by counting the lines as it reads them. EDIT does not assign line numbers to any new lines that you insert into the source.

EDIT distinguishes between original and nonoriginal lines. Original lines are source lines that have not been split or inserted; nonoriginal lines are split lines and inserted lines. Commands that take line numbers as arguments may only refer to original lines. EDIT moves forward, or backward up to a set limit, according to whether the line number you type is greater or less than the

current line number. EDIT passes over or deletes (if appropriate) nonoriginal lines in searches for a given original line.

When you type a period (.) instead of a line number, EDIT always uses the current line whether original or nonoriginal. (For an example of its use, see Section 4.1.2.6, Deleting Whole Lines.)

You can renumber lines with the "=" command. This ensures that all lines following the current line are original. Type:

```
=15
```

to number the current line as 15, the next line 16, the next 17, and so on to the end of the file. This is how you allocate line numbers to nonoriginal lines. If you do not qualify the = command with a number, EDIT displays the message:

Number expected after =

4.2.2.4 Qualified Strings

To specify contexts for EDIT searches, you can use qualified strings. EDIT accepts the null string and always matches it at the initial search position, which is the beginning of the line except as specified below. In the absence of any qualifiers, EDIT may find the given string anywhere in a line. Qualifiers specify additional conditions for the context. EDIT recognizes five qualifiers B, E, L, P, and U as follows:

B

where the string must be at the Beginning of the line. This qualifier may not appear with E, L, or P.

E

where the string must be at the End of the line. This qualifier may not appear with B, L, or P. If E appears with the null string, it matches with the end of the line. (That is, look for nothing at the end of a line.)

L

where the search for the string is to take place Leftward from the end of the line instead of rightward from the beginning. If there is more than one occurrence of the string in a line, this qualifier makes sure that the Last one is found instead of the first. L may not appear with B, E, or P. If L appears with the null string, it matches with the end of the line. (That is, look leftward from the end of the line for an occurrence of nothing.)

P

where the line must match the string Precisely and must contain no other characters. P must not appear with B, E, or L. If P appears with a null string, it matches with an empty line.

U

where the string match is to take place whether or not upper or lower case is used. (That is, as though you translated both the string and the line into Uppercase letters before comparing them.) For example, when you specify U, the following string

```
/TWEEDledum/
```

should match a line containing

TweedleDUM

as well as any other combination in upper or lower case.

4.2.2.5 Output Processing

EDIT does not write lines read in a forward direction to the destination file immediately, but instead it adds them to an output queue in main memory. When EDIT has used up the memory available for such lines, it writes out the lines at the head of the queue as necessary. Until EDIT has actually written out a line to the destination file, you can move back and make it the current line again.

You can also send portions of the output to destination files other than TO. When you select an alternative destination file, EDIT writes out the queue of lines for the current output file.

4.2.2.6 End-of-File Handling

When EDIT reaches the end of a source file, a dummy end-of-file line becomes current. This end-of-file line has a line number one greater than the number of lines in the file. EDIT verifies the line by displaying the line number and an asterisk.

When the end-of-file line is current, commands to make changes to the current line, and commands to move forward, produce an error. Although, if you contain these kinds of commands within an infinitely repeating group, EDIT does not give an error on reaching the end-of-file line. The E (Exchange) command is an example of a command to make changes to the current line. The N (Next) command is an example of a command to move forward.

4.2.3 *Functional Groupings of EDIT Commands*

This section contains descriptions of all EDIT commands split up by function. A summary and an alphabetical list of commands appear later.

The following descriptions use slashes (/) to indicate delimiter characters (that is, characters that enclose strings). Command names appear in upper case; argument types appear in lower case as follows:

Notation	Description
a,b	line numbers (or.or*)
cg	command group
m,n	numbers
q	qualifier list (possibly empty)
se	search expression
s,t	strings of arbitrary characters
sw	switch value (+ or –)
/	string delimiter

Table 4.1: Notation for Command Descriptions

Note: Command descriptions that appear in the rest of this manual with the above notation show the SYNTAX of the command; they are not examples of what you actually type. Examples always appear as follows in

this typeface.

4.2.3.1 Selection of a Current Line

These commands have no function other than to select a new current line. EDIT adds lines that it has passed in a forward direction to the destination output queue (for further details on the output queue, see Section 4.1, "Introducing EDIT"). EDIT queues up lines that it has passed in a backward direction ready for subsequent reprocessing in a forward direction. M takes a line number, period, or asterisk. So, using the command notation described above, the correct syntax for M is as follows:

Ma

where Ma moves forward or backward to line "a" in the source. Only original lines can be accessed by line number.

M+

makes the last line actually read from the file current line. M+ moves through all the lines currently held in memory until the last one is reached.

M-

makes the last line on the output queue current. This is like saying to EDIT: "Move back as far as you can."

N

moves forward to the next line in the source. When the current line is the last line of the source, executing an N command does not create an error. EDIT increases the line number by adding one to it and creates a special end-of-file line. However, if you try to use an N command when you are already at the end of the source file, EDIT returns an error.

P

moves back to the previous line. You can move more than one line back by either repeating P, or giving a number before it. The number that you give should be equal to the number of lines you want to move back.

The syntax for the F (Find) command is

F se

So, F finds the line you specify with the search expression "se". The search starts at the current line and moves forward through the source. The search starts at the current line in order to cover the case where the current line has been reached as a side effect of previous commands—such as line deletion. An F command with no argument searches using the last executed search expression.

The syntax for the BF (Backward Find) command is

BF se

BF behaves like F except that it starts at the current line and moves backward until it finds a line that matches its search expression.

4.2.3.2 Line Insertion and Deletion

Commands may select a new current line as a side effect of their main function. Those followed by in-line insertion material must be the last command on a line. The insertion material is on successive lines terminated by Z on a line by itself. You can use the Z command to change the terminator. EDIT recognizes the terminator you give in either upper or lower case. For example, using the same notation,

```
Ia
   } <insertion material, as many
   } lines as necessary>
Z
```

inserts the insertion material before "a". Remember that "a" can be a specific line number, a period (representing the current line), or an asterisk (representing the last line of the source file). If you omit "a", EDIT inserts the material before the current line; otherwise, line "a" becomes the current line.

```
I/s/
```

inserts the contents of the file "s" (remember, "s" means any string) before the current line.

```
Ra b
                } <replacement material>
Z
Ra b/s/
```

The R command is equivalent to D followed by I. The second line number must be greater than or equal to the first. You may omit the second number if you want to replace just the one line (that is, if b = a). You may omit both numbers if you want to replace the current line. The line following line b becomes the new current line.

The syntax for the D (Delete) command is as follows:

```
Da b
```

So, D deletes all lines from a to b inclusive. You may omit the second line number if you want to delete just the one line (that is, if b = a). You may omit both numbers if you want to delete the current line. The line following line b becomes the new current line.

The syntax of the DF (Delete Find) command is

```
DF se
```

The command DF (Delete Find) tells EDIT to delete successive lines from the source until it finds a line matching the search expression. This line then becomes the new current line. A DF command with no argument searches (deleting as it goes) using the last search expression you typed.

4.2.4 Line Windows

EDIT usually acts on a complete current line. However, you can define parts of the line where EDIT can execute your subsequent commands. These parts of lines are called line windows. This section describes the commands you use to define a window.

4.2.4.1 The Operational Window

EDIT usually scans all the characters in a line when looking for a given string. However it is possible to specify a "line window", so that the scan for a character starts at the beginning of the window, and not at the start of the line. In all the descriptions of EDIT context commands, "the beginning of the line" always means "the beginning of the operational window."

Whenever EDIT verifies a current line, it indicates the position of the operational window by displaying a ">" character directly beneath the line. For example in the following:

```
26.
This is line 26 this is.
    >
```

the operational window contains the characters to the right of the pointer: "line 26 this is." EDIT omits the indicator if it is at the start of the line.

The left edge of the window is also called the **character pointer** in this context, and the following commands are available for moving it:

```
>
```

moves the pointer one character to the right.

```
<
```

moves the pointer one character to the left.

```
PR
```

Pointer Reset sets the pointer to the start of the line.

The syntax for the PA (Point After) command is

```
PA q/s/
```

Point After sets the pointer so that the first character in the window is the first character following the string s. For example,

```
PA L//
```

moves the pointer to the end of the line.

The syntax for the PB (Point Before) command is

```
PB q/s/
```

Point Before is the same as PA, but includes the string itself in the window.

4.2.4.2 Single Character Operations on the Current Line

The following two commands move the character pointer one place to the right after forcing the first letter into either upper or lower case. If the first character is not a letter, or is already in the required case, these commands are equivalent to >.

The command

```
$
```

forces lower case (Dollar for Down).

The command

```
%
```

forces upper case (Percent for uP).

The "_" (underscore) command changes the first character in the window into a space character, then moves the character pointer one place to the right.

The command

```
#
```

deletes the first character in the window. The remainder of the window moves one character to the left, leaving the character pointer pointing at the next character in the line. The command is exactly equivalent to

```
E/s//
```

where "s" is the first character in the window. To repeat the effect, you specify a number before the "#" command. If the value is "n", for example, then the repeated command is equivalent to the single command

```
E/s//
```

where "s" is the first character in the window. To repeat the effect, you specify a number before the "#" command. If the value is "n", for example, then the repeated command is equivalent to the single command

```
E/s//
```

where "s" is the first "n" character in the window or the whole of the contents of the window, whichever is the shorter. Consider the following example:

```
5#
```

deletes the next five characters in the window. If you type a number equal to or greater than the number of characters in the window, EDIT deletes the contents of the entire window. EDIT treats a sequence of "#" commands in the same way as a single, repeated "#" command. So, # # # # # is the same as typing a single #, pressing RETURN after each single #, five times.

You can use a combination of ">" "%" "$" "_" and "#" commands to edit a line character by character, the commands appearing under the characters they affect. The following text and commands illustrate this:

```
o Oysters,, Come ANDDWALK with     us
%>$$$$$$$#>>$$$$$$$$_$$$$$$$$$$###
```

The commands in the example above change the line to

```
O oysters, come and walk with us
```

leaving the character pointer immediately before the word "us".

4.2.5 String Operations on the Current Line

To specify which part of the current line to qualify, you can either alter the basic string or point to a variant, as described in the next two sections.

4.2.5.1 Basic String Operations

Three similar commands are available for altering parts of the current line. The A, B, and E commands insert their second (string) argument After, Before, or in Exchange for their first argument respectively. You may qualify the first argument. If the current line were

```
The Carpenter beseech
```

then the commands

```
E U/carpenter/Walrus/          <Exchange>
B/bese/did /                   <insert string before>
A L//;/                        <Insert string after>
```

would change the line to

The Walrus did beseech;

4.2.5.2 The Null String

You can use the null, or empty string (//) after any string command. If you use the null string as the second string in an E command, EDIT removes the first string from the line. Provided EDIT finds the first string, an A or B command with a null second string does nothing; otherwise, EDIT returns an error. A null first string in any of the three commands matches at the initial search position. The initial search position is the current character position (initially the beginning of the line) unless either of the E or L qualifiers is present, in which case the initial position is the end of the line. For example,

```
A//carpenter/
```

puts the text carpenter After nothing, that is, at the beginning of the line. Whereas

```
A L//carpenter
```

puts carpenter at the end of the line After the Last nothing.

4.2.5.3 Pointing Variant

The AP (insert After and Point), BP (insert Before and Point), and EP (Exchange and Point) commands take two strings as arguments and act exactly like A, B, and E. However, AP, BP, and EP have an additional feature: when the operation is complete, the character pointer is left pointing to the first character following both text strings. So, using the same command syntax notation,

```
AP/s/t/
```

is equivalent to

```
A/s/t/;PA/st/
```

while

BP/s/t/

is equivalent to

B/s/t/;PA/ts/

and

2EP U/tweadle/Tweedle/

would change

tweadledum and TWEADLEdee

into

Tweedledum and Tweedledee

leaving the character pointer just before dee.

4.2.5.4 Deleting Parts of the Current Line

You use the commands DTA (Delete Till After) and DTB (Delete Till Before) to delete from the beginning of the line (or character pointer) to a specified string. To delete from a given context until the end of the line, you use the commands DFA (Delete From After) and DFB (Delete From Before). If the current line were

All the King's horses and all the King's men

then the command

DTB L/King's/

would change it to

King's men

while

DTA/horses /

would change it to

and all the King's men

4.2.6 Miscellaneous Current Line Commands

This section includes some further commands that explain how to repeat commands involving strings, how to split the current line, and how to join lines together.

Whenever EDIT executes a string alteration command (for example, A, B, or E), it becomes the current string alteration command. To repeat the current string alteration command, you can type a single quote ('). The ' command has no arguments. It takes its arguments from the last A, B, or E command.

> **WARNING:** Unexpected effects occur if you use sequences such as
>
> E/castle/knight/; 4('; E/pawn/queen/)
>
> The second and subsequent executions of the ' command refer to a different command than the first. The above example would exchange castle and knight twice and exchange pawn and queen seven times instead of exchanging castle and knight once and then four times exchanging castle and knight and pawn and queen.

4.2.6.1 Splitting and Joining Lines

EDIT is primarily a line editor. Most EDIT editing commands do not operate over line boundaries, but this section describes commands for splitting a line into more than one line and for joining together two or more successive lines.

To split a line before a specified context, you use the SB command. The syntax for the SB command is

SB q/s/

SB takes an optional qualifier represented here by q, and a string /s/. SB Splits the current line Before the context you specify with the qualifier and string. EDIT sends the first part of the line to the output and makes the remainder into a new, nonoriginal current line.

To split a line after a specified context, you use the SA command. The syntax for SA is

SA q/s/

SA takes an optional qualifier and a string (q and /s/). SA Splits the current line After the context you specify with the qualifier and string.

To concatenate a line, you use the CL command. The syntax for CL is

CL/s/

CL takes an optional string that is represented here by /s/. CL or Concatenate Line forms a new current line by concatenating the current line, the string you specified, and the next line from the source, in that order. If the string is a null string, you may type the command CL without specifying a string.

For an example of splitting and joining lines, look at the text

Humpty Dumpty sat on a wall; Humpty
Dumpty had a
great fall.

The old verse appears disjointed; the lines need to be balanced. If you make the first line the current line, the commands

SA /; /; 2CL/ /

change the line into

Humpty Dumpty sat on a wall;

leaving

Humpty Dumpty had a great fall.

as the new current line.

4.2.7 Inspecting Parts of the Source: The Type Commands

The following commands all tell EDIT to advance through the source, sending the lines it passes to the verification file as well as to the normal output (where relevant). Because these commands are most frequently used interactively (that is, with verification to the screen), they are known as the "type" commands. They have this name because you can use them to "type" out the lines you specify on the screen. This does not however mean that you cannot use them to send output to a file. After EDIT has executed one of these commands, the last line it "typed" (that is, displayed) becomes the new current line.

The syntax for the T (Type) command is

Tn

Tn types "n" lines. If you omit "n", typing continues until the end of the source. However, you can always interrupt the typing with CTRL-C.

Note: Throughout this manual when you see a hyphen between two keys, you press them at the same time. So CTRL-C means to hold down the CTRL key while you type C.

When you use the T command, the first line EDIT types is the current line, so that, for example,

```
F /It's my own invention/; T6
```

types six lines starting with the one containing "It's my own invention". (Note that to find the correct line, you must type the "I" in "It's" in upper case.)

The command

```
TP
```

types the lines in the output queue. Thus, TP (Type Previous) is equivalent to EDIT executing M- followed by typing until it reaches the last line it actually read from the source.

The command

```
TN
```

types until EDIT has changed all the lines in the output queue. (For more information on the output queue, see Section 4.1, "Introducing EDIT.") So, a TN (Type Next) command types N lines, where N was the number specified as the P option. (To find out more about the P option, refer to Section 4.1.1, Calling EDIT). The advantage of the TN command is that everything visible during the typing operation is available in memory to P and BF commands.

The syntax for the TL (Type with Line numbers) command is as follows:

```
TLn
```

TLn types n lines as for T, but with line numbers added. Inserted and split lines do not have line numbers, EDIT displays a "+ + + +" instead. For example,

```
20 O oysters, come and walk with us
+ + + + and then we'll have some tea
```

The original line starting with "O oysters" has a line number. The nonoriginal line, inserted after line 20, starts with + + + +. (Remember that you can use the = command to renumber nonoriginal lines.)

4.2.8 Control of Command, Input, and Output Files

EDIT uses four types of files:

- command
- input
- output
- verification

Once you have entered EDIT, you cannot change the verification file with a command. (To find out more about the verification file, see Section 4.1.1, "Calling EDIT.") The following sections describe commands that can change the command, input, and output files that you set up when you enter EDIT.

4.2.8.1 Command Files

When you enter EDIT, it reads commands from the terminal or the file that you specify as WITH. To read commands from another file, you can use the C command. The syntax for the command is

```
C .s.
```

where the string "s" represents a filename. As AmigaDOS uses the slash symbol (/) to separate filenames, you should use periods (.), or some other symbol, to delimit the filename. A symbol found in a string should not be used as a delimiter. When EDIT has finished all the commands in the file (or you give a Q command), it closes the file and control reverts to the command following the C command. For example, the command

```
C .:T/XYZ.
```

reads and executes commands from the file :T/XYZ

4.2.8.2 Input Files

To insert the entire contents of a file at a specific point in the source, you use the I and R commands. These commands are described in Section 4.1.2.7 earlier in this chapter.

Section 4.1.1 described how to call EDIT. In that section, the source file was referred to as the FROM file. However, you can also associate the FROM file with other files, using the command FROM. The FROM command has the following form:

```
FROM .s.
```

where the string "s" is a filename. A FROM command with no argument reselects the original source file.

When EDIT executes a FROM command, the current line remains current; however, the next line comes from the new source.

EDIT does not close a source file when the file ceases to be current; you can read further lines from the source file by reselecting it later.

To close an output file that you opened in EDIT, and that subsequently you want to open for input (or the other way around), you must use the CF (Close File) command. The CF command has the following form:

```
CF .s.
```

where the string "s" represents a filename. When you end an EDIT session, EDIT closes automatically all the files you opened in EDIT.

Note: Any time you open a file, EDIT starts at the beginning of that file. If you close a file with CF, EDIT starts on the first line of that file if you reopen it, and not at the line it was on when you closed the file.

An example of the use of the FROM command to merge lines from two files follows:

Command	**Action**
M10	*Pass lines 1-9 from the FROM (source) file*
FROM .XYZ.	*Select new input, line 10 remains current*
M6	*Pass line 10 from FROM, lines 1-5 from XYZ*
FROM	*Reselect FROM*
M14	*Pass line 6 from XYZ, lines 11-13 from FROM*
FROM .XYZ.	*Reselect XYZ*
M*	*Pass line 14 from FROM, the rest of XYZ*
FROM	*Reselect FROM*
CF .XYZ.	*Close XYZ*
M*	*Pass the rest of FROM (lines 15 till end-of-file)*

4.2.8.3 Output Files

EDIT usually sends output to the file with filename TO. However, EDIT does not send the output immediately. It keeps a certain number of lines in a queue in main memory as long as possible. These lines are previous current lines or lines that EDIT has passed before reaching the present current line. The number of lines that EDIT can keep depends on the options you specified when you called EDIT. Because EDIT keeps these lines, it has the capability for moving backward in the source.

To associate the output queue with a file other than that with the filename TO, you can also use the TO command. The TO command has the form

TO .s.

where "s" is a filename.

When EDIT executes a TO command, it writes out the existing queue of output lines if the output file is switched.

EDIT does not close an output file when it is no longer current. By re-selecting the file, you can add further lines to it. The following example shows how you can split up the source between the main destination TO and an alternate destination XYZ.

Command	**Action**
M11	*Pass lines 1–10 to TO*
TO.XYZ.	*Switch output file*
M21	*Pass lines 11–20 to XYZ*
TO	
M31	*Pass lines 21–30 to TO*
TO.XYZ.	
M41	*Pass lines 31–40 to XYZ*
TO	

If you want to reuse a file, you must explicitly close it. The command

CF .filename.

closes the file with the filename you specify as the argument.

These input/output commands are useful when you want to move part of the source file to a later place in the output. For example,

Command	**Action**
TO .:T/1.	*Output to temporary file*
1000N	*Advance through source*
TO	*Revert to TO*
CF .:T/1.	*Close output file :T.1*
I2000.:T/1.	*Reuse as input file*

If you use the CF command on files you have finished with, the amount of memory you need is minimized.

4.2.9 Loops

You can type an unsigned decimal number before many commands to indicate repetition, for example,

```
24N
```

You can also specify repeat counts for command groups in the same way as for individual commands, for example,

```
12(F/handsome/; E/handsome/hansom/; 3N)
```

If you give a repeat count of zero (0), the command or command group is repeated indefinitely or until EDIT reaches the end of the source.

4.2.10 Global Operations

Global operations are operations that take place automatically as EDIT scans the source in a forward direction. You can start and stop global operations with special commands, described in the following sections.

> **WARNING:** Be careful when you move backward through the source not to leave any active or "enabled" globals. These enabled globals could undo a lot of your work!

4.2.10.1 Setting Global Changes

Three commands, GA, GB, and GE are provided for simple string changes on each line. Their syntax is as follows:

```
GA q/s/t/
GB q/s/t/
GE q/s/t/
```

These commands apply an A, B, or E command, as appropriate, to any occurrence of string "s" in a new current line. They also apply to the line that is current at the time the command is executed.

G commands do not rescan their replacement text; for example, the following command

```
GE/Tiger Lily/Tiger Lily/
```

would not loop forever, but would have no visible effect on any line. However, as a result of the "change", EDIT would verify certain lines.

EDIT applies the global changes to each new current line in the order in which you gave the commands.

4.2.10.2 Cancelling Global Changes

The REWIND command cancels all global operations automatically. You can use the CG (Cancel Global) command to cancel individual commands at any time.

When a global operation is set up by one of the commands GA, GB, or GE, the operation is allocated an identification number which is output to the verification file (for example, G1). The argument for CG is the number of the global operation to be cancelled. If CG is executed with no argument, EDIT cancels all globals.

4.2.10.3 Suspending Global Changes

You can suspend individual global operations, and later resume using them with DG (Disable Global) and EG (Enable Global) commands. These take the global identification number as their argument. If you omit the argument, all globals are turned off or on (disabled or enabled), as appropriate.

4.2.11 Displaying the Program State

Two commands beginning with SH (for SHow) output information about the state of EDIT to the verification file.

The command SHD (SHow Data) takes the form

SHD

and displays saved information values, such as the last search expression.

The command SHG (SHow Globals) takes the form

SHG

and displays the current global commands, together with their identification numbers. It also gives the number of times each global search expression matches.

4.2.12 Terminating an EDIT Run

To "wind through" the rest of the source, you use the W command (Windup). Note that W is illegal if output is not currently directed to TO. EDIT exits when it has reached the end of the source, closed all the files, and relinquished the memory. Reaching the end of the highest level command file has the same effect as W. If you call EDIT specifying only the FROM filename, EDIT renames the temporary output file it created with the same name as the original (that is, the FROM filename), while it renames the original information as the file :T/EDIT-

BACKUP. This backup file is, of course, only available until the next time EDIT is run.

The STOP command stops EDIT immediately. No further input or output is attempted. In particular, the STOP command stops EDIT from overwriting the original source file. Typing STOP ensures that no change is made to the input information.

The Q command stops EDIT from executing the current command file (EDIT initially accepts commands from the keyboard, but you can specify a command file with the WITH keyword or with the C command) and makes it revert to the previous one. A Q at the outermost level does the same as a W.

4.2.13 Current Line Verification

The following circumstances can cause automatic verification to occur:

- When you type a new line of commands for a current line that EDIT has not verified since it made the line current, or changed since the last verification.
- When EDIT has moved past a line that it has changed, but not yet verified.
- When EDIT displays an error message.

In the first two cases, the verification only occurs if the V switch is on. The command

```
V sw
```

changes the setting of the V switch. It is set ON (V+) if the initial state of EDIT is interactive (commands and verifications both connected to a terminal), and to OFF (V-) otherwise.

To explicitly request verification of the current line, you use the following command:

```
?
```

This command verifies the current line. It is performed automatically if the V switch is on and the information in the line has been changed. The verification consists of the line number (or + + + + if the line is not original), with the text on the next line.

An alternate form of verification, useful for lines containing nonprinting characters, is provided by the command

```
!
```

The ! command verifies the current line with character indicators. EDIT produces two lines of verification. The first is the current line in which EDIT replaces all the nongraphic characters with the first character of their hexadecimal value. In the second line, EDIT displays a minus sign under all the positions corresponding to uppercase letters and the second hexadecimal digit in the positions corresponding to nongraphic characters. All other positions contain space characters.

The following example uses the ? and ! commands. To verify the current line, you use the ? command. If, for instance, the following appears when you use the ? command:

```
?
1.
The Walrus and the ??
```

then you might try to use the E command to exchange "??" for "Carpenter". However, EDIT may not recognize the text it displayed with "??" as two question marks if the "??" characters correspond to two nongraphic characters. To find out what really is there, you use the ! command as follows:

```
!
1.
The Walrus and the 11
--  44
```

To correct the line, you can use the character pointer and # command to delete the spurious characters before inserting the correct text. (For further details on using the character pointer and # command, see Section 4.2.4, Line Windows.)

4.2.14 Miscellaneous Commands

This section describes all those commands that do not fit neatly into any of the previous categories. It describes how to change a termination character, turn trailing spaces off, renumber lines, and rewind the source file.

To change the terminator for text insertion, you use the Z command. The Z command has the following form:

```
Z/s/
```

where /s/ represents a string. The string may be of any length up to 16 characters. The string is matched in either case. In effect, the search for the terminator is done using the qualifiers PU. The initial terminator string is Z.

To turn trailing spaces on or off, you use the TR (TRailing spaces) command. The TR command takes the following form:

```
TR sw
```

where sw represents a switch (+ for ON; – for OFF). EDIT usually suppresses all trailing spaces. TR+ allows trailing spaces to remain on both input and output lines.

To renumber the source lines, you use the = command. The = command takes the form:

```
=n
```

where "n" represents a number. The command =n sets the current line number to "n". If you then move to the lines below the current line, EDIT renumbers all the following original and nonoriginal lines. Although, if you move back to previous lines after using the = command, EDIT marks all the previous lines in the output queue as nonoriginal. When you rewind the source file, EDIT renumbers all the lines in the file-original, nonoriginal, and those previously renumbered with the = command.

To rewind the source file, you use the REWIND command. For example,

```
REWIND
```

This command rewinds the input file so that line 1 is the current line again. First EDIT scans the rest of the source (for globals, and so forth). Then it writes the lines to the destination, which is then closed and re-opened as a new source. It closes the original source using a temporary file as a destination. Any globals that you specify are cancelled. EDIT does not necessarily require you to type the complete word (that is, REWIND). To move to the beginning, you can type any of the following: REWI, REWIN, or REWIND.

4.2.15 Abandoning Interactive Editing

To abandon most commands that read text, you press CTRL-C. In particular, if you realize that a search expression has been mistyped, then CTRL-C stops the search. Similarly the T command types to the end of the source, but CTRL-C abandons this action.

After you press CTRL-C, EDIT responds with the message

```
*** BREAK
```

and returns to reading commands. The current line does, of course, depend on exactly when you pressed CTRL-C.

Quick Reference Card

This list uses the following abbreviations:

Notation	Description
qs	Qualified string
t	String
n	Line number, or .or*(current and last line)
sw	+ or – (on or off)

Character Pointer Commands (Line Window Commands)

Command	Action
<	Move character pointer left
>	Move character pointer right
#	Delete character at pointer
$	Lower case character at pointer
%	Upper case character at pointer
_	Turn character at pointer to space
PA qs	Move character pointer to after qs
PB qs	Move character pointer to before qs
PR	Reset character pointer to start of line

Positioning Commands

Command	Action
M n	Move to line n
M +	Move to highest line in memory
M –	Move to lowest line in memory
N	Next line
P	Previous line
REWIND	Rewind input file

Search Commands

Command	Action
F qs	Find string qs
BF qs	Same as F, but move backward through file
DF qs	Same as F, but delete lines as they are passed

Text Verification

Command	Action
?	Verify current line
!	Verify with character indicators
T	Type to end of file
Tn	Type n lines
TLn	Type n lines with line numbers
TN	Type until buffer changed
TP	M-, then type to last line in buffer
V sw	Set verification on or off

Operations on the Current Line

Command	Action
A qs t	Place string t after qs
AP qs t	Same as A, but move character pointer
B qs t	Place string t before qs
BP qs t	Same as B, but move character pointer
CL t	Concatenate current line, string t, and next line
D	Delete current line
DFA qs	Delete from after qs to end of line
DFB qs	Delete from before qs to end of line
DTA qs	Delete from start of line to after qs
DTB qs	Delete from start of line to before qs
E qs t	Exchange string qs with string t
EP qs t	Same as E, but move character pointer
I	Insert material from terminal before line
I t	Insert from file t
R	Replace material from terminal
R t	Replace material from file t
SA qs	Split line after qs
SB qs	Split line before qs

Globals

Command	Action
GA qs t	Globally place t after qs
GB qs t	Globally place t before qs
GE qs t	Globally exchange qs for t
CG n	Cancel global n (all if n omitted)
DG n	Disable global n (all if n omitted)
EG n	Enable global n (all if n omitted)
SHG	Display info on globals used

Input/Output Manipulation

Command	Action
FROM	Take source from original
FROM t	Take source from file t
TO	Revert to original destination
TO t	Place output lines in file t
CF t	Close file t

Other Commands

Command	Action
'	Repeat previous A, B, or E command
= n	Set line number to n
Ct	Take commands from file t
Hn	Set halt at line n. If n =* then halt and unset h
Q	Exit from command level; windup if at level 1
SHD	Show data
STOP	Stop
TR sw	Set/unset trailing space removal
W	Windup
Zt	Set input terminator to string t

Appendix

Error Codes and Messages

The error messages that appear on the screen when you use the FAULT or WHY command fall into two general categories:

1. user errors
2. programmer errors.

This appendix gives the probable cause and a suggestion for recovery for each of these error codes. The codes appear in numerical order within their category.

User Errors

103: insufficient free store
Probable cause:
You don't have enough physical memory on the Amiga to carry this operation out.

Recovery suggestion:
First, try to stop some of the applications that are running that you don't need. For example, close any unnecessary windows. Otherwise, buy more memory. Stop some of the tasks that are less important to you and reissue the command. It may be that you have enough memory, but it has become "fragmented"; rebooting may help.

104: task table full
Probable cause:
Limited to 20 CLI tasks, or equivalent.

120: argument line invalid or too long
Probable cause:
Your argument for this command is incorrect or contains too many options.
Recovery suggestion:
Consult the command specifications in Chapter 2 of this manual for the correct argument template.

121: file is not an object module
Probable cause:
Either you misspelled the command name, or this file may not be in loadable file form.
Recovery suggestion:
Either retype the file name, or make sure that the file is a binary program file. Remember that in order to execute a command sequence the command EXECUTE must be used before the file name.

122: invalid resident library during load

202: object in use
Probable cause:
The file or directory specified is already being used by another application in a manner incompatible with the way you want to use it.
Recovery suggestion:
If another application is writing to a file, then nobody else can read from it. If another application is reading from a file, then nobody else can write to it. If an application is using a directory or reading from a file, then nobody else may delete or rename the file or directory. You must stop the other application using the file or directory and then try again.

203: object already exists
Probable cause:
The object name that you specified is that of an object that already exists.
Recovery suggestion:
You must first delete the directory or file if you really want to reuse that name.

204: directory not found

205: object not found
Probable cause:
AmigaDOS cannot find the device or file you specified. You have probably made a typographical or spelling error.
Recovery suggestion:
Check device names and filenames for correct spellings. You also get this error if you attempt to create a file in a directory that does not exist.

206: invalid window
Probable cause:
You have either made the dimensions too big or too small, or you have failed to define an entire window. (For example, you must not forget the final slash.)

You can also get this error from NEWCLI if you supply a device name that is not a window.

Recovery suggestion:

You should respecify the window.

210: invalid stream component name

Probable cause:

You have included an invalid character in the filename you have specified, or the filename is too long. Each file or directory must be less than 30 characters long. A filename cannot contain control characters.

212: object not of required type

Probable cause:

Maybe you've tried to do an operation that requires a filename and you gave it a directory name or *vice versa*. For example, you might have given the command TYPE dir, where "dir" is a directory. AmigaDOS doesn't allow you to display a directory, only files.

Recovery suggestion:

Check on the command usage in Chapter 2 of the *AmigaDOS User's Manual* in this book.

213: disk not validated

Probable cause:

Either you just inserted a disk and the disk validation process is in progress, or it may be a bad disk.

Recovery suggestion:

Wait for the disk validation process to finish—it normally only takes less than a minute. If AmigaDOS cannot validate the disk because it is bad, then the disk remains unvalidated. In this case, you can only read from the disk and you must copy your information onto another disk.

214: disk write-protected

Probable cause:

This disk is write-protected. The Amiga cannot write over information that is already on the disk. You can only read information from this disk. You cannot store any information of your own here.

Recovery suggestion:

Save your information on a disk that is not write-protected, or change the write-protect tab on the disk.

215: rename across devices attempted

Probable cause:

RENAME only changes a filename on the same device, although you can use

it to rename a file from one directory into another on the same device.
Recovery suggestion:
Copy the file to the object device and delete it from the source device.

216: directory not empty
Probable cause:
You cannot delete a directory unless it is empty.
Recovery suggestion:
Delete the contents of the directory. Study the command specification for DELETE in Chapter 2 of this manual.

218: device not mounted
Probable cause:
The word "mounted" here means "inserted into the drive"; either you've made a typographical error, or the disk with the desired name isn't mounted.
Recovery suggestion:
Check the spelling of the devices, or insert the correct disk.

220: comment too big
Probable cause:
Your filenote has exceeded the maximum number of characters allowed (80).
Recovery suggestion:
Retype the filenote adhering to these limits.

221: disk full
Probable cause:
You do not have sufficient room on the disk to do this operation.
Recovery suggestion:
Use another disk or delete some unnecessary files or directories.

222: file is protected from deletion
Probable cause:
The file or directory has been protected from deletion.
Recovery suggestion:
You either did not mean to delete that file, or you really did mean it. If you really did mean it, you must use the PROTECT command to alter the protection status. Refer to the PROTECT command in Chapter 2. Also use the LIST command to check on what the protections of this particular file or disk are.

223: file is protected from writing
Probable cause:

The file or directory has been protected from being overwritten.
Recovery suggestion:
You either did not mean to write to that file, or you really did mean it. If you really did mean it, you must use the PROTECT command to alter the protection status. Refer to the PROTECT command in Chapter 2. Also use the LIST command to check on the protections of this particular file or disk.

224: file is protected from reading
Probable cause:
The file or directory has been protected from being read.
Recovery suggestion:
You either did not mean to read from that file, or you really did mean it. If you really did mean it, you must use the PROTECT command to alter the protection status. Refer to the PROTECT command in Chapter 2. Also use the LIST command to check on the protections of this particular file or disk.

225: not a DOS disk
Probable cause:
The disk in the drive is not a formatted DOS disk.
Recovery suggestion:
Place a suitably formatted DOS disk in the drive instead, or else format the disk using the FORMAT command if you don't want any of the information on it.

226: no disk in drive
Probable cause:
You have attempted to read or write to a disk drive where there is no disk.
Recovery suggestion:
Place a suitably formatted DOS disk in the drive.

Programmer Errors

209: packet request type unknown
Probable cause:
You have asked a device handler to attempt an operation it cannot do (for example, the console handler cannot rename anything).
Recovery suggestion:
Check the request code passed to device handlers.

211: invalid object lock
Probable cause:
You have used something that is not a valid lock.

Recovery suggestion:
Check your code so that you only pass valid locks to AmigaDOS calls that expect locks.

219: seek error
Probable cause:
You have attempted to call SEEK with invalid arguments.
Recovery suggestion:
Make sure that you only SEEK within the file. You cannot SEEK outside the bounds of the file.

232: no more entries in directory
Probable cause:
There are no more entries in the directory that you are examining.
Recovery suggestion:
This error code indicates that the AmigaDOS call EXNEXT has no more entries in the directory you are examining to hand back to you. Stop calling EXNEXT.

Glossary

Arguments
Additional information supplied to commands.
Character pointer
Pointer to the left edge of a line window in EDIT. You use it to define the part of a line that EDIT may alter.
Character string
Sequence of printable characters.
Command
An instruction you give directly to the computer.
Command Line Interface (CLI)
A **process** that decodes user input.
Console handler
See **terminal handler.**
Command template
The method of defining the syntax for each **command.**
Control combination
A combination of the CTRL key and a letter or symbol. The CTRL key is pressed down while the letter or symbol is typed. It appears in the documentation, for example, in the form CTRL-A.
Current cursor position
The position the cursor is currently at.

Current directory
This is either the **root directory** or the last **directory** you set yourself in with the command CD.

Current drive
The disk drive that is inserted and declared to be current. The default is SYS:.

Current line
The line that EDIT has in its hand at any time.

Current string alteration command
An instruction that changes the current string.

Delimiter characters
Characters used at the beginning and end of a **character string.**

Destination file
File being written to.

Device name
Unique name given to a device, e.g. DF0: = floppy drive 0:.

Directory
A collection of **files.**

Editing commands
Commands input from the keyboard that control an editing session.

Extended mode
Commands appear on the command line and are not executed until you finish the command line.

File
A collection of related data.

Filename
A name given to a file for identification purposes.

Immediate mode
Commands that are executed immediately.

Keyword
Arguments to commands that must be stated explicitly.

Line windows
Parts of a line for EDIT to execute subsequent **commands** on.

Memory
This is sometimes known as store and is where a computer stores its data and instructions.

Multi-processing
The execution of two or more **processes** in parallel, that is, at the same time.

Output queue
Buffer in memory holding data before being written out to **file.**

Priority
The relative importance of a **process.**

Process
A job requested by the operating system or the user.

Qualifiers
Characters that specify additional conditions for the context in string.

Qualified string
A string preceded by one or more qualifiers.

Queue
See **Output queue.**

Root directory
The top level in the filing system. **Files** and **directories** within the root directory have their names preceded by a colon (:).

Sequential files
A **file** that can be accessed at any point by starting at the beginning and scanning sequentially until the point is reached.

Source file
File being read from.

Syntax
The format or "grammar" you use for giving a command.

Terminal handler
A **process** handling input and output from the terminal or console.

Volume name
The unique name associated with a disk.

Wild card
Symbols used to match any pattern.

AmigaDOS Developer's Manual

Contents

Using Preferences

The default text size on the Amiga allows up to 60 characters per line in a full-width CLI window. Many developers prefer to use 80 characters per line. You can change the text style by using the Preferences tool from your Workbench disk; however, the new text width will not necessarily take effect on any windows that you currently have opened. That is, any old windows in the system remain with a text size of 60. To incorporate text size into the system, you need to create a new window, select the old window, and finally delete the old window.

Follow these steps:

1. Use the NEWCLI command.
2. Select the old window.
3. Use the ENDCLI command in the old window to delete the old window.

If you alter the CLI selection, the change may not take effect immediately. If you save the new preferences and reboot, they take effect.

Chapter 1

Programming on the Amiga

This chapter introduces the reader to programming in C or Assembler under AmigaDOS.

1.1 Introduction

The AmigaDOS programming environment is available on the Amiga, Sun, and IBM PC.

This manual assumes that you have some familiarity with either C or Assembler. It does not attempt to teach either of these languages. An introduction to C can be found in the book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall. There are a number of books on writing 68000 assembler, including *Programming the MC68000* by Tim King and Brian Knight, published by Addison Wesley.

1.2 Program Development for the Amiga

This section describes how to develop programs for the Amiga. It describes what you need before you start, how you can call the system routines, and how to create a file that you can execute on the Amiga.

> **WARNING:** Before you do ANYTHING, you should make a backup copy of your system disk. For instructions, see the section, "Backing Up," at the beginning of the *AmigaDOS User's Manual* in this book.

1.2.1 Getting Started

Before you start writing programs for the Amiga, you need the following items:

1. Documentation on AmigaDOS and other system routines that you can call. For example, you need the *AmigaDOS User's Manual, ROM Kernel Manual,* and possibly the *AmigaDOS Technical Reference Manual* as well.
2. Documentation on the language you intend to use. If you intend to use Assembler or C, then this manual tells you how to use these tools although it does not contain any specific information normally found in a language reference manual.
3. Header files containing the necessary Amiga structure definitions and the values for calling the sytem routines that you need. Commodore-Amiga provides these header files as included files for either C (usually ending in .h) or assembler (ending in .i). To use a particular resident library, you must include one or more header files containing the relevant definitions. For example, to use AmigaDOS from C, you must include the file "dos.h".
4. An assembler or compiler either running on the Amiga itself or on one of the cross development environments.
5. The Amiga linker, again running on the Amiga or on another computer, as well as the standard Amiga library containing functions, interface routines, and various absolute values.
6. Tools to download programs if you are using a cross-development environment.

1.2.2 Calling Resident Libraries

You should note that there are two ways of calling system routines from a user assembly program. C programmers simply call the function as specified. You usually call a system routine in assembler by placing the library base pointer

for that resident library in register A6 and then jumping to a suitable negative offset from that pointer. The offsets are available to you as absolute externals in the Amiga library, with names of the form _LVOname. So, for instance, a call could be JSR _LVOname(A6), where you have loaded A6 with a suitable library base pointer. These base pointers are available to you from the OpenLibrary call to Exec; you can find the base pointer for Exec at location 4 (the only absolute location used in the Amiga). This location is also known as AbsExecBase which is defined in Amiga.lib. (See the *ROM Kernel Manual* for further details on Exec.)

You can call certain RAM-based resident libraries and the AmigaDOS library in this way, if required. Note that the AmigaDOS library is called "dos.library". However, you do not need to use A6 to hold a pointer to the library base; you may use any other register if you need to. In addition, you may call AmigaDOS using the resident library call feature of the linker. In this case, simply code a JSR to the entry point and the linker notes the fact that you have used a reference to a resident library. When your code is loaded into memory, the loader automatically opens the library and closes it for you when you have unloaded. The loader automatically patches references to AmigaDOS entry points to refer to the correct offset from the library base pointer.

1.2.3 Creating an Executable Program

To produce a file that you can execute on the Amiga, you should follow the four steps below. You can do each step either on the Amiga itself or on a suitable cross-development computer.

1. Get your program source into the Amiga. To do this, you can type it directly in using an editor, or you can transfer it from another computer. Note that you can use the READ and DOWNLOAD programs on the Amiga to transfer character or binary files.
2. Assemble or compile your program.
3. Link your program together, including any startup code you may require at the beginnning, and scan the Amiga library and any others you may need to satisfy any external references.
4. Load your program into the Amiga and watch it run!

1.3 Running a Program Under the CLI

There are two ways you can run a program. First, you can run your program under a CLI (Command Line Interface). Second, you can run your program under the Workbench. This section describes the first of the two ways.

Running a program under the CLI is a little like using an old-fashioned

line-oriented TTY system although you might find a CLI useful, for example, to port your program over to your Amiga as a first step in development. To load and enter your program, you simply type the name of the file that contains the binary and possibly follow this with a number of arguments.

1.3.1 Initial Environment in Assembler

When you load a program under a CLI, you type the name of the program and a set of arguments. You may also specify input or output redirection by means of the ">" and "<" symbols. The CLI automatically provides all this information for the program when it starts up.

When the CLI starts up a program, it allocates a stack for that program. This stack is initially 4000 bytes, but you may change the stack size with the STACK command. AmigaDOS obtains this stack from the general free memory heap just before you run the program; it is not, however, the same as the stack that the CLI uses. AmigaDOS pushes a suitable return address onto the stack that tells the CLI to regain control and unload your program. Below this on the stack at 4(SP) is the size of the stack in bytes, which may be useful if you wish to perform stack checking.

Your program starts with register A0 pointing to the arguments you, or anyone else running your program typed. AmigaDOS stores the argument line in memory within the CLI stack and this pointer remains valid throughout your program. Register D0 indicates the number of characters in the argument line. You can use these initial values to decode the argument line to find out what the user requires. Note that all registers may be corrupted by a user program.

To make the initial input and output file handles available, you call the AmigaDOS routines Input() and Output(). Remember that you may have to open the AmigaDOS library before you do this. The calls return file handles that refer to the standard input and output the user requires. This standard input and output is usually the terminal unless you redirected the I/O by including ">" or "<" on the argument line. You should not close these file handles with your program; the CLI opened them for you and it will close them, if required.

1.3.2 Initial Environment in C

When programming in C, you should always include the startup code as the first element in the linker input. This means that the linker enters your program at the startup code entry point. This section of code scans the argument list and makes the arguments available in "argv", with the number of arguments in "argc" as usual. It also opens the AmigaDOS library and calls Input() and Output() for you, placing the resulting file handles into "stdin" and "stdout". It then calls the C function "main".

1.3.3 Failure of Routines

Most AmigaDOS routines return a zero if they fail; the exceptions are the Read and Write calls that return -1 on finding an error. If you receive an error return, you can call IoErr() to obtain more information on the failure. IoErr() returns an integer that corresponds to a full error code, and you may wish to take different actions depending on exactly why the call failed. A complete list of error codes and messages can be found at the end of the *AmigaDOS User's Manual* in this book.

1.3.4 Terminating a Program

To exit from a program, it is sufficient to give a simple RTS using the initial stack pointer (SP). In this case, you should provide a return code in register D0. This is zero if your program succeeded; otherwise, it is a positive number. If you return a nonzero number, then the CLI notices an error. Depending on the current fail value (set by the command FAILAT), a noninteractive CLI, such as one running a command sequence set up by the EXECUTE command, terminates. A program written in C can simply return from "main" which returns to the startup code; this clears D0 and performs an RTS.

Alternatively a program may call the AmigaDOS function Exit, which takes the return code as argument. This instructs your program to exit no matter what value the stack pointer has.

It is important at this stage to stress that AmigaDOS does not control any resources; this is left entirely up to the programmer. Any files that a user program opens must be closed before the program terminates. Likewise, any locks it obtains must be freed, any code it loads must be unloaded, and any memory it allocates returned. Of course, there may be cases where you do not wish to return all resources, for example, when you have written a program that loads a code segment into memory for later use. This is perfectly acceptable, but you must have a mechanism for eventually returning any memory, file locks, and so on.

1.4 Running a Program Under the Workbench

To run a program under the Workbench, you need to appreciate the different ways in which a program may be run on the Amiga. Under the CLI your program is running as part of the CLI process. It can inherit I/O streams and other information from the CLI, such as the arguments you provided.

If a program is running under the Workbench, then AmigaDOS starts it as a new process running at the same time as Workbench. Workbench loads the program and then sends a message to get it started. You must therefore wait

for this initial message before you start to do anything. You must retain the message and return it to Workbench when your program has finished, so that Workbench can unload the code of your program.

For C programmers, this is all done by simply using a different startup routine. For assembly language programmers, this work must be done yourself.

You should also note that a program running as a new process initiated by Workbench has no default input and output streams. You must ensure that your program opens all the I/O channels that it needs, and that it closes them all when it has finished.

1.5 Cross Development

If you are using a cross-development environment, then you need to download your code onto the Amiga. This section describes the special support Commodore-Amiga gives to Sun Microsystem and MSDOS environments. It also describes how to cross-develop in other environments without this special support.

1.5.1 Cross Development on a Sun Microsystem

The tools available on the Sun Microsystem for cross development include the assembler, linker, and two C compilers. The argument formats of the assembler and linker on the Sun Microsystem are identical to those on the Amiga when running under the CLI. The Greenhills C compiler is only available on the Sun Microsystem and is described here.

The compiler is called metacc, and it accepts several types of files. It assumes that filenames ending in .c represent C source programs. The compiler then compiles these .c files and places the resulting object program in the current directory with the same filename, but ending with .obj. The suffix .obj denotes an object file. The compiler assumes that files ending in .asm are assembly source programs. You can use the assembler to assemble these and produce an object file (ending with .obj) in the current directory.

The compiler metacc takes many options with the following format:

metacc [<opt1>[,<opt2>[,..<optn>]]][<file>[,...<filen>]]

The options available are as follows:

-c -g -go -w -p -pg -O[<optflags>] -fsingle
-S -E -C -X70 -o <output> -D <name=def>
-U <name> -I <dir> -B <string> -t[p012]

The following options instruct metacc to:

-c — just compile the program, suppressing the loading phase of the compilation, and forcing an object file to be produced even if it only compiles one program.

-g — produce additional symbol table information for the debugger dbx and to pass the -lg flag to ld.

-go — produce additional symbol table information in an older format set by the adb debugger. Also, pass the -lg flag to ld.

-w — suppress all warning messages.

-p — produce profiling code to count the number of times each routine is called. If loading takes place, replace the standard startup routine by one that is automatically called by the monitor and uses a special profiling library instead of the standard C library.
Use the prof program to generate an execution profile.

-pg — produce profiling code like -p, but invoke a run-time recording mechanism that keeps more extensive statistics and produces a gmon.out file at normal termination.
Use the gprof program to generate an execution profile.

-O[<optflags>] — use the object code optimizer to improve the generated code.
If "optflags" appears, you include <optflags> in the command line to run the optimizer. You can use -O to pass option flags.

-fsingle — use single-precision arithmetic in computations involving only flo at numbers; that is, do not convert everything to double (that is, the default).
Note: Floating-point parameters are still converted to double-precision, and functions that return values still return double-precision values.

WARNING: Certain programs run much faster using the -fsingle option, but beware that you can lose significance due to lower precision intermediate values.

-S compile the specified C program(s) and leave the assembler-language output on corresponding files ending with .obj.

-E run only the C preprocessor on the named C program(s) and send the result to the standard output.

-C prevent the C preprocessor from removing comments.

-X70 generate code using Amiga floating point format. This code is compatible with the floating point math ROM library provided on the Amiga.

-o <output> name the final output file "output". If you use this option, the file a.out is left undisturbed.

-D<name=def> define "name" to the preprocessor, as if by #define. If no definition is given, define the name as "1".

-U<name> remove any initial definition of "name".

-I<dir> always look for #include files whose names do not begin with "/" first in the directory of the <file> argument, then look in the <dir> specified in the -I option, and finally look in the /usr/include directory.

-B<string> find substitute compiler passes in the files specified by <string> with the endings cpp, ccom, and c2. If "string" is empty, use a backup version.

-t[p012] find only the designated compiler passes in the files whose names are constructed by a -B option. In the absence of a -B option, assume <string> to be /usr/new/.
The letter and number combinations that you can specify for the -t option have the following meanings:

- p cpp—the C preprocessor
- 0 metacom—both phases of the C compiler, but not the optimizer.
- 1 Ignored in this system—this option would be for the second phase of a two-phase compiler but in the Sun system; ccom includes both phases.
- 2 c2—the object code optimizer.

The compiler metacc assumes that other arguments are loaded option arguments, object programs, or libraries of object programs. Unless you specify -c, -S, or -E, metacc loads these programs and libraries together with the results of any compilations or assemblies specified, (in the order given) to produce an

executable program named a.out. To override the name a.out, you can use the loader's -o <name> option.

If a single C program is compiled and loaded all at once, the intermediate .o file is deleted.

Figure 1-A lists the filenames of special metacc files and their descriptions.

Special Files

File Description	Filename
C source code	file.c
Assembler source file	file.asm
Object file	file.o
Library of object files	file.lib
Executable output files	a.out
Temporary files	/tmp/ctm
Preprocessor	/lib/cpp
Compiler	/lib/ccom
Optional optimizer	/lib/c2
Runtime startoff	/lib/crt0.o
Startoff for profiling	/lib/mcrt0.o
Startoff for gprof-profiling	/usr/lib/gcrt0.o
Standard library	/lib/libc.a
Profiling library	/usr/lib/libc_p.a
Standard directory (#include.	/usr/include
Files produced for analysis by prof	mon.out
File produced for analysis by gprof	gmon.out

Figure 1.A: Special metacc Filenames

You can download the files you produce from the linker on the Sun to your Amiga in three ways: the first, and by far the easiest, requires a BillBoard; the second requires a parallel port; and the third requires a serial line.

If you have the special hardware device called a BillBoard, you can download your linked load file (by convention this should end with .ld) as follows:

1. Startup the program "binload" on the Sun

```
binload -p &
```

(this need only be done once)

2. Then on the Amiga, type

```
download <sun filename> <amiga filename>
```

3. To run the program, type

```
<amiga filename>
```

For example:
On the Sun, type

```
binload -p &
```

On the Amiga, type

```
download test.ld test
```

or type

```
download /usr/commodore/amiga/V24/examples/DOS/test.ld test
```

then type

```
test
```

Note that DOWNLOAD gains access to files on the Sun relative to the directory where binload started. If the directory on the Sun was /usr/commodore/amiga/V24/examples/DOS as above, the filename test.ld is all that is necessary. If you cannot remember the directory where binload started, you must specify the full name. To stop binload, do a "ps" and then a "kill" on its PID. Note that the soft reset of the computer tells binload to write a message to its standard output (the default is the window where it started). If the transfer hangs, press CTRL-C at the Amiga to kill DOWNLOAD. (See Section 3.2 in the *AmigaDOS User's Manual* in this book for further information on the AmigaDOS control conventions CTRL-C, CTRL-D, CTRL-E, and CTRL-F.)

If you do not have a BillBoard, you can download files through a parallel port. To do this, follow these steps:

1. Send the download ASCII files to the Amiga via the parallel port by typing

```
send demo.ld
```

If you do not give "send" any arguments, the standard input is used. The default output device is /dev/lp0, which is usually correct. To change the default output, use the -o argument.

2. On the Amiga, type the following:

```
READ demo
```

READ then reads characters from the parallel port and places them in the file named "demo".

3. Once READ has finished, type

```
demo
```

to run the program demo.

You can also download files serially. To do this, follow these steps:

1. Convert the Binary Load File into an ASCII hex file ending with Q by typing

```
convert <demo.ld >demo.dl
```

(where .dl, by convention, stands for DownLoad). The above rule exists in the included makefile, makeamiga. (See the *AmigaDOS Technical Reference Manual*, Chapter 2, for further details on the Amiga Binary Load files.)

2. Type

```
tip amiga
```

3. On the Amiga, type

```
READ demo serial
```

4. Within tip, type

```
~> demo.dl
```

5. When the READ completes on the Amiga, type the filename "demo" to run it.

WARNING: The Sun serial link often hangs for no apparent reason. Reboot the Sun if this happens.

If the Sun serial link should happen to hang, reboot the Sun, then type

```
tip
```

and within tip, type

```
Q
```

to get the READ on the Amiga to complete. Once this is done, start a new READ and type the following symbols on the Sun:

```
~>
```

1.5.2 Cross Development Under MS-DOS

To cross-develop on a computer running MS-DOS for your Amiga, you need various tools that are supplied in the directory \V25\bin. These include the C compiler, assembler, and linker as well as commands to assist in downloading. You use the same syntax for the tools running under MS-DOS as under the CLI on the Amiga.

To download via an IBM PC serial port (called AUX), follow these steps:

1. Type on your Amiga

   ```
   READ file SERIAL
   ```

2. On the PC, type

   ```
   convert <file.ld >AUX:
   ```

3. On your Amiga, you can now type

   ```
   file
   ```

   to the program.

1.5.3 Cross Development on Other Computers

You'll need to have a suitable cross compiler or assembler, and to include files defining all the entry points. You'll also need either the Amiga linker ALINK running on your equipment or on the Amiga. Finally you'll need a way to convert a binary file into a hexadecimal stream terminated with a Q (as this

is the way that READ accepts data), and a way of putting this data out from a serial or parallel port.

Once you have created a suitable binary file, you must transfer this to the Amiga using the READ command (as described in Section 1.5.2 of this manual). If you have the Amiga linker running on your computer, then you can transfer complete binary load files; otherwise, you'll have to transfer binary object files in the format accepted by ALINK, and then perform the link step on the Amiga.

Chapter 2

Calling AmigaDOS

This chapter describes the functions provided by the AmigaDOS resident library. To help you, it provides the following: an explanation of the syntax, a full description of each function, and a quick reference card of the available functions.

2.1 Syntax

The syntax used in this chapter shows the C function call for each AmigaDOS function and the corresponding register you use when you program in assembler.

2.1.1 Register Values

The letter/number combination (D0. . .Dn) represents registers. The text to the left of an equals sign represents the result of a function. A register (that is, D0) appearing under such text indicates the register value of the result. Text to the right of an equals sign represents a function and its arguments, where the text enclosed in parentheses is a list of the arguments. A register (for example, D2) appearing under an argument indicates the register value of that argument.

Note that not all functions return a result.

2.1.2 Case

The letter case (that is, lower or upper case) IS significant. For example, you must enter the word "FileInfoBlock" with the first letter of each component word in upper case.

2.1.3 Boolean returns

-1 (TRUE or SUCCESS), 0 (FALSE or FAILURE).

2.1.4 Values

All values are long words (that is, 4 byte values or 32 bits). Values referred to as "string" are 32-bit pointers to NULL-terminated series of characters.

2.1.5 Format, Argument, and Result

Look at "Argument:" and "Result:" for further details on the syntax used after "Format:". Result describes what is returned by the function (that is, the left of the equal sign). Argument describes what the function expects to work on (that is, the list in parentheses). Figure 2-A should help explain the syntax.

Format of function	result = Function(argument)
	Register Register
Example	lock = CreateDir(name)
	D0 D1

Figure 2-A: Format of Functions and Registers

2.2 AmigaDOS Functions

This reference section describes the functions provided by the AmigaDOS resident library. Each function is arranged alphabetically under the following headings: File Handling, Process Handling, and Loading Code. These headings indicate the action of the functions they cover. Under each function name, there is a brief description of the function's purpose, a specification of the format and the register values, a fuller description of the function, and an explanation of the syntax of the arguments and result. To use any of these functions, you must link with amiga.lib.

File Handling

Close

Purpose: To close a file for input or output.
Form: *Close(file)*
D1

Argument: file—file handle
Description:
The file handle "file" indicates the file that Close should close. You obtain this file handle as a result of a call to Open. You must remember to close explicitly all the files you open in a program. However, you should not close inherited file handles opened elsewhere.

CreateDir

Purpose: To create a new directory.
Form: *lock = CreateDir(name)*
D0 D1
Argument: name-string
Result: lock - pointer to a lock
Description:
CreateDir creates a new directory with the name you specified, if possible. It returns an error if it fails. Remember that AmigaDOS can only create directories on devices which support them, for example, disks.

A return of zero means that AmigaDOS has found an error (such as: disk write protected), you should then call IoErr(); otherwise, CreateDir returns a shared read lock on the new directory.

CurrentDir

Purpose: To make a directory associated with a lock the current working directory.
Form: *oldLock = CurrentDir(lock)*
D0 D1
Argument: lock - pointer to a lock
Result: oldLock - pointer to a lock
Description:
CurrentDir makes current a directory associated with a lock. (See also LOCK.) It returns the old current directory lock.

A value of zero is a valid result here and indicates that the current directory is the root of the initial startup disk.

DeleteFile

Purpose: To delete a file or directory.
Form: *success = DeleteFile(name)*
D0 D1

Argument: name - string
Result: success - boolean
Description:
DeleteFile attempts to delete the file or directory "name". It returns an error if the deletion fails. Note that you must delete all the files within a directory before you can delete the directory itself.

DupLock

Purpose: To duplicate a lock.
Form: *newLock = DupLock(lock)*
D0 D1
Argument: lock - pointer to a lock
Result: newLock - pointer to a lock
Description:
DupLock takes a shared filing system read lock and returns another shared read lock to the same object. It is impossible to create a copy of a write lock. (For more information on locks, see LOCK.)

Examine

Purpose: To examine a directory or file associated with a lock.
Form: *success = Examine(lock, FileInfoBlock)*
D0 D1 D2
Argument: lock - pointer to a lock
FileInfoBlock - pointer to a file info block
Result: success - boolean
Description:
Examine fills in information in the FileInfoBlock concerning the file or directory associated with the lock. This information includes the name, size, creation date, and whether it is a file or directory.

Note: FileInfoBlock must be longword aligned. You can ensure this in the C language if you use Allocmem. (See the *ROM Kernal Manual* for further details on the exec call Allocmem.)

Examine gives a return code of zero of it fails.

ExNext

Purpose: To examine the next entry in a directory.
Form: *success = ExNext(lock, FileInfoBlock)*
D0 D1 D2

Argument: lock - pointer to a lock
FileInfoBlock - pointer to a file info block
Result: success - boolean
Description:
This routine is passed a lock, usually associated with a directory, and a FileInfoBlock filled in by a previous call to Examine. The FileInfoBlock contains information concerning the first file or directory stored in the directory associated with the lock. ExNext also modifies the FileInfoBlock so that subsequent calls return information about each following entry in the directory.

ExNext gives a return code of zero if it fails for some reason. One reason for failure is reaching the last entry in the directory. However, IoErr() holds a code that may give more information on the exact cause of a failure. When ExNext finishes after the last entry, it returns ERROR_NO_MORE_ENTRIES

So, follow these steps to examine a directory:

1) Use Examine to get a FileInfoBlock about the directory you wish to examine.
2) Pass ExNext the lock related to the directory and the FileInfoBlock filled in by the previous call to Examine.
3) Keep calling ExNext until it fails with the error code held in IoErr() equal to ERROR_NO_MORE_ENTRIES.
4) Note that if you don't know what you are examining, inspect the type field of the FileInfoBlock returned from Examine to find out whether it is a file or a directory which is worth calling ExNext for.

The type field in the FileInfoBlock has two values: if it is negative, then the file system object is a file; if it is positive, then it is a directory.

Info

Purpose: Returns information about the disk.
Form: *success = Info(lock, Info_Data)*
D0 D1 D2
Argument: lock - pointer to a lock
Info_Data - pointer to an Info_Data structure
Result: success - boolean
Description:
Info finds out information about any disk in use. "lock" refers to the disk, or any file on the disk. Info returns the Info_Data structure with information about the size of the disk, number of free blocks, and any soft errors. Note that Info_Data must be longword aligned.

Input

Form: *file = Input ()*
D0
Result: file - file handle
Description:
To identify the program's initial input file handle, you use Input. (To identify the initial output, see OUTPUT.)

IoErr

Purpose: To return extra information from the system.
Form: *error = IoErr()*
D0
Result: error - integer
Description:
I/O routines return zero to indicate an error. When an error occurs, call this routine to find out more information. Some routines use IoErr(), for example, DeviceProc, to pass back a secondary result.

IsInteractive

Purpose: To discover whether a file is connected to a virtual terminal or not.
Form: *bool = IsInteractive(file)*
D0 D1
Argument: file - file handle
Result: bool - boolean
Description:
The function IsInteractive gives a boolean return. This indicates whether or not the file associated with the file handle "file" is connected to a virtual terminal.

Lock

Purpose: To lock a directory or file.
Form: *lock = Lock(name, accessMode)*
D0 D1 D2
Argument: name-string
accessMode - integer
Result: lock - pointer to a lock

Description:
Lock returns, if possible, a filing system lock on the file or directory "name". If the accessMode is ACCESS_READ, the lock is a shared read lock; if the accessMode is ACCESS_WRITE, then it is an exclusive write lock. If LOCK fails (that is, if it cannot obtain a filing system lock on the file or directory) it returns a zero.

Note that the overhead for doing a Lock is less than that for doing an Open, so that, if you want to test to see if a file exists, you should use Lock. Of course, once you've found that it exists, you have to use Open to open it.

Open

Purpose: To open a file for input or output
Form: *file = Open(name, accessMode)*
D0 D1 D2
Argument: name - string accessMode - integer
Result: file - file handle
Description:
Open opens "name" and returns a file handle. If the accessMode is MODE_OLDFILE (=1005), OPEN opens an existing file for reading or writing. However, Open creates a new file for writing if the value is MODE_NEWFILE (=1006). The "name" can be a filename (optionally prefaced by a device name), a simple device such as NIL:, a window specification such as CON: or RAW: followed by window parameters, or *, representing the current window.

For further details on the devices NIL:, CON:, and RAW:, see Chapter 1 of the *AmigaDOS User's Manual* in this book. If Open cannot open the file "name" for some reason, it returns the value zero (0). In this case, a call to the routine IoErr() supplies a secondary error code.

For testing to see if a file exists, see LOCK.

Output

Form: *file = Output()*
D0
Result: file - file handle
Description:
To identify the program's initial output file handle, you use Output. (To identify the initial input, see INPUT.)

ParentDir

Purpose: To obtain the parent of a directory or file.
Form: *Lock = ParentDir(lock)*
D0 D1
Argument: lock - pointer to a lock
Result: lock - pointer to a lock
Description:
This function returns a lock associated with the parent directory of a file or directory. That is, ParentDir takes a lock associated with a file or directory and returns the lock of its parent directory.

Note: The result of ParentDir may be zero (0) for the root of the current filing system.

Read

Purpose: To read bytes of data from a file.
Form: *actualLength = Read(file, buffer, length)*
D0 D1 D2 D3
Argument: file - file handle
buffer - pointer to buffer
length - integer
Result: actualLength - integer
Description:
You can copy data with a combination of Read and Write. Read reads bytes of information from an opened file (represented here by the argument "file") into the memory buffer indicated. Read attempts to read as many bytes as fit into the buffer as indicated by the value of length. You should always make sure that the value you give as the length really does represent the size of the buffer. Read may return a result indicating that it read less bytes than you requested, for example, when reading a line of data that you typed at the terminal.

The value returned is the length of the information actually read. That is to say, when "actualLength" is greater than zero, the value of "actualLength" is the number of characters read. A value of zero means that end-of-file has been reached. Errors are indicated by a value of –1. Read from the console returns a value when a return is found or the buffer is full.

A call to Read also modifies or changes the value of IoErr(). IoErr() gives more information about an error (for example, actualLength equals -1) when it is called.

Rename

Purpose: To rename a directory or file.
Form: *success = Rename(oldName, newName)*
D0 D1 D2
Argument: oldName - string
newName - string
Result: success - boolean
Description:
Rename attempts to rename the file or directory specified as "oldName" with the name "newName". If the file or directory "newName" exists, Rename fails and Rename returns an error.

Both the "oldName" and the "newName" can be complex filenames containing a directory specification. In this case, the file will be moved from one directory to another. However, the destination directory must exist before you do this.

Note: It is impossible to rename a file from one volume to another.

Seek

Purpose: To move to a logical position in a file.
Form: *oldPosition = Seek(file, position, mode)*
D0 D1 D2 D3
Argument: file - file handle
position - integer
mode - integer
Result: oldPosition - integer
Description:
Seek sets the read/write cursor for the file "file" to the position "position". Both Read and Write use this position as a place to start reading or writing. If all goes well, the result is the previous position in the file. If an error occurs, the result is -1. You can then use IoErr() to find out more information about the error.

"Mode" can be OFFSET_BEGINNING (=1), OFFSET_CURRENT (=0) or OFFSET_END (=1). You use it to specify the relative start position. For example, 20 from current is a position twenty bytes forward from current, -20 from end is 20 bytes before the end of the current file.

To find out the current file position without altering it, you call to Seek specifying an offset of zero from the current position.

To move to the end of a file, Seek to end-of-file offset with zero position. Note that you can append information to a file by moving to the end of a file with Seek and then writing. You cannot Seek beyond the end of a file.

SetComment

Purpose: To set a comment.
Form: *Success = SetComment(name, comment)*
D0 D1 D2
Argument: name - file name
comment - pointer to a string
Result: success - boolean
Description:
SetComment sets a comment on a file or directory. The comment is a pointer to a null-terminated string of up to 80 characters.

SetProtection

Purpose: To set file, or directory, protection.
Form: *Success = SetProtection(name, mask)*
D0 D1 D2
Argument: name - file name
mask - the protection mask required
Result: success - boolean
Description:
SetProtection sets the protection attributes on a file or directory. The lower four bits of the mask are as follows:

bit 3: if 1 then reads not allowed, else reads allowed.
bit 2: if 1 then writes not allowed, else writes allowed.
bit 1: if 1 then execution not allowed, else execution allowed.
bit 0: if 1 then deletion not allowed, else deletion allowed.

Bits 31-4 Reserved.

Only delete is checked for in the current release of AmigaDOS. Rather than referring to bits by number you should use the definitions in "include/libraries/dos.h".

UnLock

Purpose: To unlock a directory or file.
Form: *Unlock(lock)*
D1
Argument: lock - pointer to a lock

Description:
UnLock removes a filing system lock obtained from Lock, DupLock, or CreateDir.

WaitForChar

Purpose: To indicate whether characters arrive within a time limit or not.
Form: *bool = WaitForChar(file, timeout)*
D0 D1 D2
Argument: file - file handle
timeout - integer
Result: bool - boolean
Description:
If a character is available to be read from the file associated with the handle "file" within a certain time, indicated by "timeout", WaitForChar returns -1 (TRUE); otherwise, it returns 0 (FALSE). If a character is available, you can use Read to read it. Note that WaitForChar is only valid when the I/O streams are connected to a virtual terminal device. "Timeout" is specified in microseconds.

Write

Purpose: To write bytes of data to a file.
Form: *returnedLength = Write(file, buffer, length)*
D0 D1 D2 D3
Argument: file - file handle
buffer - pointer to buffer
length - integer
Result: returnedLength - integer
Description:
You can copy data with a combination of Read and Write. Write writes bytes of data to the opened file "file"; "length" refers to the actual length of data to be transferred; "buffer" refers to the buffer size.

Write returns a value that indicates the length of information actually written. That is to say, when "length" is greater than zero, the value of "length" is the number of characters written. A value of -1 indicates an error. The user of this call must always check for an error return which may, for example, indicate that the disk is full.

Process Handling

CreateProc

Purpose: To create a new process.
Form: *process = CreateProc(name, pri, segment, stackSize)*
D0 D1 D2 D3 D4
Argument: name - string
pri - integer
segment - pointer to a segment
stackSize - integer
Result: process - process identifier
Description:
CreateProc creates a process with the name "name". That is to say, CreateProc allocates a process control structure from the free memory area and then initializes it.

CreateProc takes a segment list as the argument "segment". (See also under LOADSEG and UNLOADSEG.) This segment list represents the section of code that you intend to run as a new process. CreateProc enters the code at the first segment in the segment list, which should contain suitable initialization code or a jump to such.

"StackSize" represents the size of the root stack in bytes when CreateProc activates the process. "Pri" specifies the required priority of the new process. The result is the process identifier of the new process, or zero if the routine failed.

The argument "name" specifies the process name.

A zero return code implies an error of some kind.

DateStamp

Purpose: To obtain the date and time in internal format.
Form: *v:= DateStamp(v)*
Argument: v - pointer
Description:
DateStamp takes a vector of three long words that is set to the current time. The first element in the vector is a count of the number of days. The second element is the number of minutes elapsed in the day. The third is the number of **ticks** elapsed in the current minute. A tick happens 50 times a second. DateStamp ensures that the day and minute are consistent. All three elements are zero if the date is unset. DateStamp currently only returns even multiples of 50 ticks. Therefore the time you get is always an integral number of seconds.

Delay

Purpose: To delay a process for a specified time.
Form: *Delay(timeout)*
D1
Argument: timeout - integer
Description:
The function Delay takes an argument "timeout"; "timeout" allows you to specify how long the process should wait in ticks (50 per second).

DeviceProc

Purpose: To return the process identifier of the process handling that I/O.
Form: *process = DeviceProc(name)*
D0 D1
Argument: name - string
Result: process - process identifier
Description:
DeviceProc returns the process identifier of the process that handles the device associated with the specified name. If DeviceProc cannot find a process handler, the result is zero. If "name" refers to a file on a mounted device, then IoErr() returns a pointer to a directory lock.

You can use this function to determine the process identification of the handler process where the system should send its messages.

Exit

Purpose: To exit from a program.
Form: *Exit(returnCode)*
D1
Argument: returnCode - integer
Description:
Exit acts differently depending on whether you are running a program under a CLI or not. If you run, as a command under a CLI, a program that calls Exit, the command finishes and control reverts to the CLI. Exit then interprets the argument "returnCode" as the return code from the program.

If you run the program as a distinct process, Exit deletes the process and releases the space associated with the stack, segment list, and process structure.

Loading Code

Execute

Purpose: To execute a CLI command.
Form: *Success = Execute(commandString, input, output)*
D0 D1 D2 D3
Argument: commandString - string
input - file handle
output - file handle
Result: Success - boolean
Description:
This function takes a string (commandString) that specifies a CLI command and arguments, and attempts to execute it. The CLI string can contain any valid input that you could type directly at a CLI, including input and output indirection using > and <.

The input file handle will normally be zero, and in this case the EXECUTE command will perform whatever was requested in the commandString and then return. If the input file handle is nonzero then after the (possibly null) commandString is performed subsequent input is read from the specified input file handle until end of file is reached.

In most cases the output file handle must be provided, and will be used by the CLI commands as their output stream unless redirection was specified. If the output file handle is set to zero then the current window, normally specified as *, is used. Note that programs running under the Workbench do not normally have a current window.

The Execute function may also be used to create a new interactive CLI process just like those created with the NEWCLI function. In order to do this you should call Execute with an empty commandString, and pass a file handle relating to a new window as the input file handle. The output file handle should be set to zero. The CLI will read commands from the new window, and will use the same window for output. This new CLI window can only be terminated by using the ENDCLI command. For this command to work the program C:RUN must be present in C:.

LoadSeg

Purpose: To load a load module into memory.
Form: *segment = LoadSeg(name)*
D0 D1
Argument: name - string

Result: segment - pointer to a segment
Description:
The file "name" is a load module produced by the linker. LoadSeg takes this and scatter-loads the code segments into memory, chaining the segments together on their first words. It recognizes a zero as indicating the end of the chain.

If an error occurs, LoadSeg unloads any loaded blocks and returns a false (zero) result.

If all goes well (that is, LoadSeg has loaded the module correctly), then Loadseg returns a pointer to the beginning of the list or blocks. Once you have finished with the loaded code, you can unload it with a call to UnLoadSeg. (For using the loaded code, see CREATEPROC.)

UnLoadSeg

Purpose: To unload a segment previously loaded by LOADSEG.
Form: *UnLoadSeg(segment)*
D1
Argument: segment - pointer to a segment
Description:
UnLoadSeg unloads the segment identifier that was returned by LoadSeg. "segment" may be zero.

Quick Reference Card

File Handling

Close	to close a file for input or output.
CreateDir	to create a new directory.
CurrentDir	to make a directory associated with a lock the current working directory.
DeleteFile	to delete a file or directory.
DupLock	to duplicate a lock.
Examine	to examine a directory or file associated with a lock.
ExNext	to examine the next entry in a directory.
Info	to return information about the disk.
Input	to identify the initial input file handle.
IoErr	to return extra information from the system.
IsInteractive	to discover whether a file is connected to a virtual terminal or not.
Lock	to lock a file or directory.

Open	to open a file for input or output.
Output	to identify the initial output file handle.
ParentDir	to obtain the parent of a directory or file.
Read	to read bytes of data from a file.
Rename	to rename a file or directory.
Seek	to move to a logical position in a file.
SetComment	to set a comment.
SetProtection	to set file, or directory, protection.
Unlock	to unlock a file or directory.
WaitForChar	to indicate whether characters arrive within a time limit or not.
Write	to write bytes of data to a file.

Process Handling

CreateProc	to create a new process.
DateStamp	to obtain the date and time in internal format.
Delay	to delay a process for a specified time.
DeviceProc	to return the process identifier of the process handling that I/O.
Exit	to exit from a program.

Loading Code

Execute	to execute a CLI command.
LoadSeg	to load a load module into memory.
UnloadSeg	to unload a segment previously loaded by LOADSEG.

Chapter 3

The Macro Assembler

This chapter describes the AmigaDOS Macro Assembler. It gives a brief introduction to the 68000 microchip. This chapter is intended for the reader who is acquainted with an assembly language on another computer.

3.1 Introduction to the 68000 Microchip

This section gives a brief introduction to the 68000 microchip. It should help you to understand the concepts introduced later in the chapter. It assumes that you have already had experience with assembly language on another computer.

The memory available to the 68000 consists of

- the internal registers (on the chip), and
- the external main memory.

There are 17 registers, but only 16 are available at any given moment. Eight of them are **data registers** named D0 to D7, and the others are **address registers** called A0 to A7. Each register contains 32 bits. In many contexts, you may use either kind of register, but others demand a specific kind. For instance, you may use any register for operations on **word** (16-bit) and **long word** (32-bit) quantities or for indexed addressing of main memory. Although, for operations on **byte** (8-bit) operands, you may only use data registers, and for addressing main memory, you may only use address registers as stack pointers or base registers. Register A7 is the stack pointer, and this is in fact two distinct registers: the system stack pointer available in supervisor mode and the user stack pointer available in user mode.

The main memory consists of a number of bytes of memory. Each byte has an identifying number called its **address.** Memory is usually (but not always) arranged so that its bytes have addresses 0, 1, 2, . . ., N-2, N-1 where there are N bytes of memory in total. The size of memory that you can directly access is very large—up to 16 million bytes. The 68000 can perform operations on bytes, words, or long words of memory. A word is two consecutive bytes. In a word, the first byte has an even address. A long word is four consecutive bytes also starting at an even address. The address of a long word is the even address of its lowest numbered first byte.

As well as holding items of data being manipulated by the computer, the main memory also holds the **instructions** that tell the computer what to do. Each instruction occupies from one to 5 words, consisting of an **operation word** between zero and four operand words. The operation word specifies what action is to be performed (and implicitly how many words there are in the whole instruction). The **operand words** indicate where in the registers or main memory are the items to be manipulated, and where the result should be placed.

The assembler usually executes instructions one at a time in the order that they occur in memory, like the way you follow the steps in a recipe or play the notes in a piece of written music. There is a special register called the **program counter** (PC) which you use to hold the address of the instruction you want the assembler to execute next. Some instructions, called **jumps** or **branches**, upset the usual order, and force the assembler to continue executing the instruction at a specific address. This lets the computer perform an action repeatedly, or do different things depending on the values of data items.

To remember particular things about the state of the computer, you can use one other special register called the **status register** (SR).

3.2 Calling the Assembler

The command template for assem is

"PROG=FROM/A,-O/K,-V/K,-L/K,-H/K,-C/K,-I/K"

Alternatively, the format of the command line can be described as

```
assem <sourcefile>  [-o <object file>]
                    [-l <listing file>]
                    [-v <verification file>]
                    [-h <header file>]
                    [-c <options>]
                    [-i <include dirlist>]
```

The assembler does not produce an object file or a listing file unless you request them explicitly.

As the assembler is running, it generates diagnostic messages (errors, warnings, and assembly statistics) and sends them to the screen unless you specify a verification file.

To force the inclusion of the named file in the assembly at the head of the source file, you use -h <filename> on the command line. This has the same effect as using

```
INCLUDE "<filename>"
```

on line 1 of the source file.

To set up the list of directories that the assembler should search for any INCLUDEd files, you use the -i keyword. You should specify as many directories as you require after the -i, separating the directory names by a comma (,), a plus sign (+), or a space. Note that if you use a space, you must enclose the entire directory list in double quotes ("). Unix users, however, must escape any double quotes with a backslash (\").

The order of the list determines the order of the directories where the assembler should search for INCLUDEd files. The assembler initially searches the current directory before any others. Thus any file that you INCLUDE in a program must be in the current directory, or in one of the directories listed in the -i list. For instance, if the program "fred" INCLUDEs, apart from files in the current directory, a file from the directory "intrnl/incl", a file from the directory "include/asm", and a file from the directory "extrnl/incl", you can give the -i directory list in these three ways:

```
assem fred -i intrnl/incl, include/asm,extrnl/incl
assem fred -i intrnl/incl + include/asm + extrnl/incl
assem fred -i "intrnl/incl include/asm extrnl/incl"
```

or, by using the space separator on the Sun under Unix, like this

```
assem fred -i \"intrnl/incl include/asm extrnl/incl\"
```

The -c keyword allows you to pass certain options to the assembler. Each option consists of a single character (in either upper or lower case), possibly followed immediately by a number. Valid options follow here:

S produces a symbol dump as a part of the object file.
D inhibits the dumping of local labels as part of a symbol dump. (For C programmers, any label beginning with a period is considered a local label.)
C ignores the distinction between upper and lower case in labels.
X produces a cross-reference table at the end of the listing file.

Examples

```
assem fred.asm -o fred.o
```

assembles the file "fred.asm" and produces an object module in the file fred.o.

```
assem fred.asm -o fred.o -l fred.lst
```

assembles the file fred.asm, produces an object module in the file fred.o, and produces a listing file in "fred.lst".

3.3 Program Encoding

A program acceptable to the assembler takes the form of a series of input lines that can include any of the following:

- Comment or Blank lines
- Executable Instructions
- Assembler Directives

3.3.1 Comments

To introduce comments into the program, you can use three different methods:

1. Type a semicolon (;) anywhere on a line and follow it with the text of the comment. For example,

```
CMPA.L A1, A2 ; Are the pointers equal?
```

2. Type an asterisk (*) in column one of a line and follow it with the text of the comment. For example,

```
* This entire line is a comment
```

3. Follow any complete instruction or directive with at least one space and some text. For example,

```
MOVEQ #10,D0 place initial value in D0
```

In addition, note that all blank lines are treated by the assembler as comment lines.

3.3.2 Executable Instructions

The source statements have the general overall format:

[<label>] <opcode> [<operand>[,<operand>]...[<comment>]

To separate each field from the next, press the SPACEBAR or TAB key. This produces a separator character. You may use more than one space to separate fields.

3.3.2.1 Label Field

A label is a user symbol, or programmer-defined name, that either

a) Starts in the first column and is separated from the next field by at least one space, or
b) Starts in any column, and is followed immediately with a colon (:).

If a label is present, then it must be the first nonblank item on the line. The assembler assigns the value and type of the program counter, that is, the memory address of the first byte of the instruction or data being referenced, to the label. Labels are allowed on all instructions, and on some directives, or

they may stand alone on a line. See the specifications of individual directives in Section 3.7 for whether a label field is allowed.

Note: You must not give multiple definitions to labels. Also, you must not use instruction names, macro names, directives, or register names as labels.

3.3.2.2 Local Labels

Local labels are provided as an extension to the Motorola specification. They take the form nnn$ and are only valid between any proper (named) labels. Thus, in this example code segment

```
Labels    Opcodes    Operands
FOO:      MOVE.L     D6,D0
1$:       MOVE.B     (A0)+,(A1)+
          DBRA       D0,1$
          MOVEQ      #20,D0
BAA:      TRAP       #4
```

the label 1$ is only available from the line following the one labelled FOO to the line before the one labelled BAA. In this case, you could then use the label 1$ in a different scope elsewhere in the program.

3.3.2.3 Opcode Field

The Opcode field follows the Label field and is separated from it by at least one space. Entries in this field are of three types.

1. The MC68000 operation codes, as defined in the *MC68000 User Manual.*
2. Assembler Directives.
3. Macro invocations.

To enter instructions and directives that can operate on more than one data size, you use an optional Size-Specifier subfield, which is separated from the opcode by the period (.) character. Possible size specifiers are as follows:

B - Byte-sized data (8 bits)
W- Word-sized data (16 bits)
L - Long Word-sized data (32 bits)
    or Long Branch specifier
S - Short Branch specifier

The size specifier must match with the instruction or directive type that you use.

3.3.2.4 Operand Field

If present, the operand field contains one or more operands to the instruc-

tion or directive, and must be separated from it by at least one space. When you have two or more operands in the field, you must separate them with a comma (,). The operand field terminates with a space or newline character (a newline character is what the assembler receives when you press RETURN), so you must not use spaces between operands.

3.3.2.5 Comment Field

Anything after the terminating space of the operand field is ignored. So the assembler treats any characters you insert after a space as a comment.

3.4 Expressions

An expression is a combination of symbols, constants, algebraic operators, and parentheses that you can use to specify the operand field to instructions or directives. You may include relative symbols in expressions, but they can only be operated on by a subset of the operators.

3.4.1 Operators

The available operators are listed below in order of precedence.

1. Unary Minus, Logical NOT (-and~)
2. Lshift, Rshift (<<and>>)
3. Logical AND, Logical OR (& and !)
4. Multiply, Divide (* and/)
5. Add, Subtract (+ and –)

To override the precedence of the operators, enclose sub-expressions in parentheses. The assembler evaluates operators of equal precedence from left to right. Note that, normally, you should not have any spaces in an expression, as a space is regarded as a delimiter between one field and another.

3.4.2 Operand Types for Operators

In the following table, "A" represents absolute symbols, and "R" represents relative symbols. The table shows all the possible operator/operand combinations, with the type of the resulting value. "x" indicates an error. The Unary minus and the Logical operators are only valid with an absolute operand.

Operators	Operands			
	A op A	R op R	A op R	R op A
+	A	x	R	R
-	A	A	x	R
*	A	x	x	x
/	A	x	x	x
&	A	x	x	x
!	A	x	x	x
>>	A	x	x	x
<<	A	x	x	x

Table 3-A: Operand Types for Operators

3.4.3 Symbols

A symbol is a string of up to 30 characters. The first character of a symbol must be one of the following:

- An alphabetic character, that is, a through z, or A through Z.
- An underscore (_).
- A period (.).

The rest of the characters in the string can be any of these characters or also numeric (0 through 9). In all symbols, the lower case characters (a–z) are *not* treated as synonyms with their upper case equivalents (unless you use the option C when you invoke the assembler). So "fred" is different from "FRED" and "FRed". However, the assembler recognizes instruction opcodes, directives, and register names in either upper or lower case. A label equated to a register name with EQUR is also recognized by the assembler in either upper or lower case. Symbols can be up to 30 characters in length, all of which are significant. The assembler takes symbols longer than this and truncates them to 30 characters, giving a warning that it has done so. The Instruction names, Directive names, Register names, and special symbols CCR, SR, SP, and USP cannot be used as user symbols. A symbol can be one of three types:

Absolute

a) The symbol was SET or EQUated to an Absolute value.

Relative
a) The symbol was SET or EQUated to a Relative value.
b) The symbol was used as a label.

Register
a) The symbol was set to a register name using EQUR (This is an extension from the Motorola specification).

There is a special symbol "*", which has the value and type of the current program counter, that is, the address of the current instruction or directive that the assembler is acting on.

3.4.4 Numbers

You may use a number as a term of an expression, or as a single value. Numbers ALWAYS have absolute values and can take one of the following formats:

Decimal
(a string of decimal digits)
Example: 1234

Hexadecimal
($ followed by a string of hex digits)
Example: $89AB

Octal
(@ followed by a string of octal digits)
Example: @743

Binary
(% followed by zeros and ones)
Example: %10110111

ASCII Literal
(Up to 4 ASCII characters within quotes)
Examples: 'ABCD' '*'

Strings of less than 4 characters are justified to the right, using nul as the packing character.

To obtain a quote character in the string, you must use two quotes. An example of this is

'It' 's'

3.5 *Addressing Modes*

The effective address modes define the operands to instructions and directives, and you can find a detailed description of them in any good reference book on the 68000. Addresses refer to individual bytes, but instructions, word and long word references, access more than one byte, and the address for these must be word aligned.

In the following table, Dn represents one of the data registers (D0–D7), "An" represents one of the address registers (A0–A7, SP and PC), "a" represents an absolute expression, "r" represents a relative expression, and "Xn" represents An or Dn, with an optional ".W" or ".L" size specifier. The syntax for each of the modes is as follows:

Table 3-B: Macro Assembler Address Modes and Registers

Address Mode	Description and Examples
Dn	Data Register Direct
	Example: MOVE D0, D1
An	Address Register Direct
	Example: MOVEA A0,A1
(An)	Address Register Indirect
	Example: MOVE D0,(A1)
(An)+	Address Register Indirect Post Increment
	Example: MOVE (A7)+,D0
-(An)	Address Register Indirect Pre Decrement
	Example: MOVE D0,-(A7)
a(An)	Address Register Indirect with Displacement
	Example: MOVE 20(A0),D1
a(An,Xn)	Address Register Indirect with Index
	Example: MOVE 0(A0,D0),D1
	MOVE 12(A1,A0.L),D2
	MOVE 120(A0,D6.W),D4
a	Short absolute (16 bits)
	Example: MOVE $1000,D0
a	Long absolute (32 bits)
	Example: MOVE $10000,D0
r	Program Counter Relative with Displacement
	Example: MOVE ABC,D0
	(ABC is relative)
r(Xn)	Program Counter Relative with Index
	Example: MOVE ABC(D0.L),D1
	(ABC is relative)

#a		Immediate data Example: MOVE #1234,D0

USP
CCR
SR
	}	Special addressing modes
		
		Example: MOVE A0,USP
		MOVE D0,CCR
		MOVE D1,SR

3.6 Variants on Instruction Types

Certain instructions (for example, ADD, CMP) have an **address variant** (that refers to address registers as destinations), **immediate** and **quick** forms (when immediate data possibly within a restricted size range appears as an operand), and a **memory variant** (where both operands must be a postincrement address).

To force a particular variant to be used, you may append A, Q, I, or M to the instruction mnemonic. In this case, the assembler uses the specified form of the instruction, if it exists, or gives an error message.

If, however, you specify no particular variant, the assembler automatically converts to the "I", "A", or "M" forms where appropriate. However, it does not convert to the "Q" form. For example, the assembler converts the following:

```
ADD.L A2, A1
```

to

```
ADDA.L A2, A1
```

3.7 Directives

All assembler directives (with the exception of DC and DCB) are instructions to the assembler, rather than instructions to be translated into object code. At the beginning of this section, there is a list of all the directives (Table 3-C), arranged by function; at the end there is an individual description for each directive, arranged by function.

Note that the assembler only allows labels on directives where specified. For example, EQU is allowed a label. It is optional for RORG, but not allowed for LLEN or TTL.

The following table lists the directives by function:

Table 3-C: Directives

Assembly Control

Directive	Description
SECTION	Program section
RORG	Relocatable origin
OFFSET	Define offsets
END	Program end

Symbol Definition

Directive	Description
EQU	Assign permanent value
EQUR	Assign permanent register value
REG	Assign permanent value
SET	Assign temporary value

Data Definition

Directive	Description
DC	Define constants
DCB	Define Constant Block
DS	Define storage

Listing Control

Directive	Description
PAGE	Page-throw to listing
LIST	Turn on listing
NOLIST (NOL)	Turn off listing
SPC n	Skip n blank lines
NOPAGE	Turn off paging
LLEN n	Set line length (60 < = n <= 132)
PLEN n	Set page length (24 < = n < = 100)
TTL	Set program title (max 40 chars.)
NOOBJ	Disable object code output
FAIL	Generate an assembly error
FORMAT	No action
NOFORMAT	No action

Conditional Assembly

Directive	Description
CNOP	Conditional NOP for alignment
IFEQ	Assemble if expression is 0
IFNE	Assemble if expression is not 0
IFGT	Assemble if expression > 0

Directive	Description
IFGE	Assemble if expression > = 0
IFLT	Assemble if expression < 0
IFLE	Assemble if expression < = 0
IFC	Assemble if strings are identical
IFNC	Assemble if strings are not identical
IFD	Assemble if symbol is defined
IFND	Assemble if symbol is not defined
ENDC	End of conditional assembly

Macro Directives

Directive	Description
MACRO	Define a macro name
NARG	Special symbol
ENDM	End of macro definition
MEXIT	Exit the macro expansion

External Symbols

Directive	Description
XDEF	Define external name
XREF	Reference external name

General Directives

Directive	Description
INCLUDE	Insert file in the source
MASK2	No action
IDNT	Name program unit

Assembly Control Directives

SECTION Program Section
Format: [<label>] SECTION <name>[,<type>]

This directive tells the assembler to restore the counter to the last location allocated in the named section (or to zero if used for the first time).

<name> is a character string optionally enclosed in double quotes.
<type> if included, must be one of the following keywords:

CODE indicates that the section contains relocatable code. This is the default.
DATA Indicates that the section contains initialized data (only).
BSS indicates that the section contains uninitialized data.

The assembler can maintain up to 255 sections. Initially, the assembler begins with an unnamed CODE section. The assembler assigns the optional symbol <labels> to the value of the program counter after it has executed the SECTION directive. In addition, where a section is unnamed, the shorthand for that section is the keyword CODE.

RORG Set Relative Origin
Format: [<label>] RORG <absexp>

The RORG directive changes the program counter to be <absexp> bytes from the start of the current relocatable section. The assembler assigns relocatable memory locations to subsequent statements, starting with the value assigned to the program counter. To do addressing in relocatable sections, you use the "program counter relative with displacement" addressing mode. The label value assignment is the same as for SECTION.

OFFSET Define offsets
Format: OFFSET <absexp>

To define a table of offsets via the DS directive beginning at the address <absexp>, you use the OFFSET directive. Symbols defined in an OFFSET table are kept internally, but no code-producing instructions or directives may appear. To terminate an OFFSET section, you use a RORG, OFFSET, SECTION, or END directive.

END End of program
Format: [<label>] END

The END directive tells the assembler that the source is finished, and the assembler ignores subsequent source statements. When the assembler encounters the END directive during the first pass, it begins the second pass. If, however, it detects an end-of-file before an END directive, it gives a warning message. If the label field is present, then the assembler assigns the value of the current program counter to the label before it executes the END directive.

Symbol Definition Directives

EQU Equate symbol value
Format: <label> EQU <exp>

The EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The value assigned is permanent, so you may not define the label anywhere else in the program.

Note: Do not insert forward references within the expression.

EQUR Equate register value
Format: <label> EQUR <register>

This directive lets you equate one of the processor registers with a user symbol. Only the Address and Data registers are valid, so special symbols like SR, CCR, and USP are illegal here. The register is permanent, so you cannot define the label anywhere else in the program. The register must not be a forward reference to another EQUR statement. The assembler matches labels defined in this way without distinguishing between upper and lower case.

REG Define register list
Format: <label>REG<register list>

The REG directive assigns a value to label that the assembler can translate into the register list mask format used in the MOVEM instruction. <register list> is of the form

R1 [-R2][/R3[-R4]]. . .

SET Set symbol value
Format: <label> SET <exp>

The SET directive assigns the value of the expression in the operand field to the symbol in the label field. SET is identical to EQU, apart from the fact that the assignment is temporary. You can always change SET later on in the program.

Note: You should not insert forward references within the expression or refer forward to symbols that you defined with SET.

Data Definition Directives

DC Define Constant
Format: [<label>] DC[.<size>] <list>

The DC directive defines a constant value in memory. It may have any number of operands, separated by commas (,). The values in the list must be capable of being held in the data location whose size is given by the size specifier on the directive. If you do not give a size specifier, DC assumes it is .W. If the size is .B, then there is one other data type that can be used: that of the ASCII string. This is an arbitrarily long series of ASCII characters, contained within quotation marks. As with ASCII literals, if you require a quotation mark in the string, then you must enter two. If the size is .W or .L, then the assembler aligns the data onto a word boundary.

DCB Define Constant Block
Format: [<label>] DCB[.<size>] <absexp>,<exp>

You use the DCB directive to set a number (given by <absexp>) of bytes,

words, or long words to the value of the expression <exp>. DCB.<size> n, exp is equivalent to repeating n times the statement DC.<size> exp.

DS Define Storage
Format: [<label>] DS[.<size>] <absexp>

To reserve memory locations, you use the DS directive. DS, however, does no initialization. The amount of space the assembler allocates depends on the data size (that you give with the size specifier on the directive), and the value of the expression in the operand field. The assembler interprets this as the number of data items of that size to allocate. As with DC, if the size specifier is .W or .L, DS aligns the space onto a word boundary. So, DS.W 0 has the effect of aligning to a word boundary only. If you do not give a size specifier, DS assumes a default of .W. See CNOP for a more general way of handling alignment.

Listing Control Directives

PAGE Page Throw
Format: PAGE

Unless paging has been inhibited, PAGE advances the assembly listing to the top of the next page. The PAGE directive does not appear on the output listing.

LIST Turn on Listing
Format: LIST

The LIST directive tells the assembler to produce the assembly listing file. Listing continues until it encounters either an END or a NOLIST directive. This directive is only active when the assembler is producing a listing file. The LIST directive does not appear on the output listing.

NOLIST Turn off Listing
Format: NOLIST
NOL

The NOLIST or NOL directive turns off the production of the assembly listing file. Listing ceases until the assembler encounters either an END or a LIST directive. The NOLIST directive does not appear on the program listing.

SPC Space Blank Lines
Format: SPC <number>

The SPC directive outputs the number of blank lines given by the operand field, to the assembly listing. The SPC directive does not appear on the program listing.

NOPAGE Turn off Paging
Format: NOPAGE

The NOPAGE directive turns off the printing of page throws and title headers on the assembly listing.

LLEN Set Line Length
Format: LLEN <number>

The LLEN directive sets the line length of the assembly listing file to the value you specified in the operand field. The value must lie between 60 and 132, and can only be set once in the program. The LLEN directive does not appear on the assembly listing. The default is 132 characters.

PLEN Set Page Length
Format: PLEN <number>

The PLEN directive sets the page length of the assembly listing file to the value you specified in the operand field. The value must lie between 24 and 100, and you can only set it once in the program. The default is 60 lines.

TTL Set Program Title
Format: TTL <title string>

The TTL directive sets the title of the program to the string you gave in the operand field. This string appears as the page heading in the assembly listing. The string starts at the first nonblank character after the TTL, and continues until the end of line. It must not be longer than 40 characters in length. The TTL directive does not appear on the program listing.

NOOBJ Disable Object Code Generation
Format: NOOBJ

The NOOBJ directive disables the production of the object code file at the end of assembly. This directive disables the production of the code file, even if you specified a filename when you called the assembler.

FAIL Generate a user error
Format: FAIL

The FAIL directive tells the assembler to flag an error for this input line.

FORMAT No action
Format: FORMAT

The assembler accepts this directive but takes no action on receiving it. FORMAT is included for compatibility with other assemblers.

NOFORMAT No action
Format: NOFORMAT

The assembler accepts this directive but takes no action on receiving it. NOFORMAT is included for compatibility with other assemblers.

Conditional Assembly Directives

CNOP Conditional NOP
Format: [<label>] CNOP <number>,<number>

This directive is an extension from the Motorola standard and allows a section of code to be aligned on any boundary. In particular, it allows any data structure or entry point to be aligned to a long word boundary.

The first expression represents an offset, while the second expression represents the alignment required for the base. The code is aligned to the specified offset from the nearest required alignment boundary. Thus

```
CNOP 0,4
```

aligns code to the next long word boundary while

```
CNOP 2,4
```

aligns code to the word boundary 2 bytes beyond the nearest long word aligned boundary.

IFEQ Assemble if expression = 0
IFNE Assemble if expression <> 0
IFGT Assemble if expression > 0
IFGE Assemble if expression > = 0
IFLT Assemble if expression < 0
IFLE Assemble if expression < = 0
Format: IFxx <absexp>

You use the IFxx range of directives to enable or disable assembly, depending on the value of the expression in the operand field. If the condition is not TRUE (for example, IFEQ 2+1), assembly ceases (that is, it is disabled). The conditional assembly switch remains active until the assembler finds a matching ENDC statement. You can nest conditional assembly switches arbitrarily, terminating each level of nesting with a matching ENDC.

IFC Assemble if strings are identical
IFNC Assemble if strings are not identical
Format: IFC <string>,<string>
IFNC <string>,<string>

The strings must be a series of ASCII characters enclosed in single quotes,

for example, 'FOO' or (the empty string). If the condition is not TRUE, assembly ceases (that is, it is disabled). Again the conditional assembly switch remains active until the assembler finds a matching ENDC statement.

IFD Assemble if symbol defined
IFND Assemble if symbol not defined
Format: IFD <symbol name>
IFND <symbol name>

Depending on whether or not you have already defined the symbol, the assembler enables or disables assembly until it finds a matching ENDC.

ENDC End conditional assembly
Format: ENDC

To terminate a conditional assembly, you use the ENDC directive, set up with any of the 8 IFxx directives above. ENDC matches the most recently encountered condition directive.

Macro Directives

MACRO Start a macro definition
Format: <label> MACRO

MACRO introduces a macro definition. ENDM terminates a macro definition. You must provide a label, which the assembler uses as the name of the macro; subsequent uses of that label as an operand expand the contents of the macro and insert them into the source code. A macro can contain any opcode, most assembler directives, or any previously defined macro. A plus sign (+) in the listing marks any code generated by macro expansion. When you use a macro name, you may append a number of arguments, separated by commas. If the argument contains a space (for example, a string containing a space) then you must enclose the entire argument within < (less than) and > (greater than) symbols.

The assembler stores up and saves the source code that you enter (after a MACRO directive and before an ENDM directive) as the contents of the macro. The code can contain any normal source code. In addition, the symbol \ (backslash) has a special meaning. Backslash followed by a number "n" indicates that the value of the nth argument is to be inserted into the code. If the nth argument is omitted then nothing is inserted. Backslash followed by the symbol "@" tells the assembler to generate the text ".nnn", where nnn is the number of times the \@ combination it has encountered. This is normally used to generate unique labels within a macro.

You may not nest macro defintions, that is, you cannot define a macro within a macro, although you can call a macro you previously defined. There

is a limit to the level of nesting of macro calls. This limit is currently set at ten.

Macro expansion stops when the assembler encounters the end of the stored macro text, or when it finds a MEXIT directive.

NARG Special symbol
Format: NARG

The symbol NARG is a special reserved symbol and the assembler assigns it the index of the last argument passed to the macro in the parameter list (even nulls). Outside of a macro expansion, NARG has the value 0.

ENDM Terminate a macro definition
Format: ENDM

This terminates a macro definition introduced by a MACRO directive.

MEXIT Exit from macro expansion
Format: MEXIT

You use this directive to exit from macro expansion mode, usually in conjuction with the IFEQ and IFNE directives. It allows conditional expansion of macros. Once it has executed the directive, the assembler stops expanding the current macro as though there were no more stored text to include.

External Symbols

XDEF Define an internal label as an external entry point
Format: XDEF <label>[,<label>...]

One or more absolute or relocatable labels may follow the XDEF directive. Each label defined here generates an external symbol definition. You can make references to the symbol in other modules (possibly from a high-level language) and satisfy the references with a linker. If you use this directive or XREF, then you cannot directly execute the code produced by the assembler.

XREF Define an external name
Format: XREF <label>[,<label>...]

One or more labels that must not have been defined elsewhere in the program follow the XREF directive. Subsequent uses of the label tell the assembler to generate an external reference for that label. You use the label as if it referred to an absolute or relocatable value depending on whether the matching XDEF referred to an absolute or relocatable symbol.

The actual value used is filled in from another module by the linker. The

linker also generates any relocation information that may be required in order for the resulting code to be relocatable.

External symbols are normally used as follows. To specify a routine in one program segment as an external definition, you place a label at the start of the routine and quote the label after an XDEF directive. Another program may call that routine if it declares a label via the XREF directive and then jumps to the label so declared.

General Directives

INCLUDE Insert an external file
Format: INCLUDE "<file name>"

The INCLUDE directive allows the inclusion of external files into the program source. You set up the file that INCLUDE inserts with the string descriptor in the operand field. You can nest INCLUDE directives up to a depth of three, enclosing the filenames in quotes as shown. INCLUDE is especially useful when you require a standard set of macro definitions or EQUs in several programs.

You can place the definitions in a single file and then refer to them from other programs with a suitable INCLUDE. It is often convenient to place NOLIST and LIST directives at the head and tail of files you intend to include via INCLUDE. AmigaDOS searches for the file specification first in the current directory, then in each subsequent directory in the list you gave in the -i option.

MASK2 No action
Format: MASK2

The assembler accepts the MASK2 directive, but it takes no action on receiving it.

IDNT Name program unit
Format: IDNT <string>

A program unit, which consists of one or more sections, must have a name. Using the IDNT directive, you can define a name consisting of a string optionally enclosed in double quotes. If the assembler does not find an IDNT directive, it outputs a program unit name that is a null string.

Chapter 4

The Linker

This chapter describes the AmigaDOS Linker. The AmigaDOS Linker produces a single binary load file from one or more input files. It can also produce overlaid programs.

4.1 Introduction

ALINK produces a single binary output file from one or more input files. These input files, known as **object files**, may contain external symbol information. To produce object files, you use your assembler or language translator. Before producing the output, or **load file**, the linker resolves all references to symbols.

The linker can also produce a link map and symbol cross-reference table.

Associated with the linker is an **overlay supervisor**. You can use the overlay supervisor to overlay programs written in a variety of languages. The linker produces load files suitable for overlaying in this way.

You can drive the linker in two ways:

1. As a **Command line.** You can specify most of the information necessary for running the linker in the command parameters.
2. As a **Parameter file.** As an alternative, if a program is being linked repetitively, you can use a parameter file to specify all the data for the linker.

These two methods can take three types of input files:

1. **Primary binary input.** This refers to one or more object files that form the initial binary input to the linker. These files are always output to the load file, and the primary input must not be empty.
2. **Overlay files.** If overlaying, the primary input forms the root of the overlay tree, and the overlay files form the rest of the structure.
3. **Libraries.** This refers to specified code that the linker incorporates automatically. Libraries may be resident or scanned. A **resident library** is a load file which may be resident in memory, or loaded as part of the "library open" call in the operating system. A **scanned library** is an object file within an archive format file. The linker only loads the file if there are any outstanding external references to the library.

The linker works in two passes.

1. In the first pass, the linker reads all the primary, library, and overlay files, and records the code segments and external symbol information. At the end of the first pass, the linker outputs the map and cross-reference table, if required.
2. If you specify an output file, then the linker makes a second pass through the input. First it copies the primary input files to the output, resolving symbol references in the process, and then it copies out the required library code segments in the same way. Note that the library code segments form part of the root of the overlay tree. Next, the linker produces data for the overlay supervisor, and finally outputs the overlay files.

In the first pass, after reading the primary and overlay input files, the linker inspects its table of symbols, and if there are any remaining unresolved references, it reads the files, if any, that you specified as the library input. The linker then marks any code segments containing external definitions for these unresolved references for subsequent inclusion in the load file. The linker only includes those library code segments that you have referenced.

4.2 Using the Linker

To use the linker, you must know the command syntax, the type of input and output that the linker uses, and the possible errors that may occur. This section attempts to explain these things.

4.2.1 Command Line Syntax

The ALINK command has the following parameters:

```
ALINK [FROM | ROOT] files [TO file] [WITH file]
  [VER file] [LIBRARY | LIB files] [MAP file]
  [XREF file] [WIDTH n]
```

The keyword template is

```
"FROM=ROOT,TO/K,WITH/K,VER/K,LIBRARY=LIB/K,
MAP/K,XREF/K,WIDTH/K"
```

In the above, "file" means a single file name, "files" means zero or more file names, separated by a comma or plus sign, and "n" is an integer.

The following are examples of valid uses of the ALINK command:

```
ALINK a
ALINK ROOT a+b+c+d MAP map-file WIDTH 120
ALINK a,b,c TO output LIBRARY :flib/lib,obj/newlib
```

When you give a list of files, the linker reads them in the order you specify. The parameters have the following meanings:

FROM: Specifies the object files that you want as the primary binary input. The linker always copies the contents of these files to the load file to form part of the overlay root. At least one primary binary input file must be specified. ROOT is a synonym for FROM.

TO: Specifies the destination for the load file. If this parameter is not given, the linker omits the second pass.

WITH: Specifies files containing the linker parameters, for example, normal command lines. Usually you only use one file here, but, for completeness, you can give a list of files. Note that parameters on the command line override those in WITH files. You can find a full description of the syntax of these files in Section 4.2.2 of this manual.

VER: specifies the destination of messages from the linker. If you do not specify VER, the linker sends all messages to the standard output (usually the terminal).

LIBRARY: specifies the files that you want to be scanned as the library. The linker includes only reference code segments. LIB is a valid alternative for LIBRARY.

MAP: specifies the destination of the link map.

XREF: specifies the destination of the cross-reference output.

WIDTH: specifies the output width that the linker can use when producing the link map and cross-reference table. For example, if you send output to a printer, you may need this parameter.

4.2.2 WITH Files

WITH files contain parameters for the linker. You use them to save typing a long and complex ALINK command line many times.

A WITH file consists of a series of parameters, one per line, each consisting of a keyword followed by data. You can terminate lines with a semicolon (;), where the linker ignores the rest of the line. You can then use the rest of the line after the semicolon to include a comment. The linker ignores blank lines.

The keywords available are as follows:

```
FROM (or ROOT) files
TO     file
LIBRARY     files
MAP     [file]
XREF     [file]
OVERLAY
tree specification
#
WIDTH     n
```

where "file" is a single filename, "files" is one or more filenames, "[file]" is an optional filename, and "n" is an integer. You may use an asterisk symbol (*) to split long lines; placing one at the end of a line tells the printer to read the next line as a continuation line. If the filename after MAP or XREF is omitted, the output goes to the VER file (the terminal by default).

Parameters on the command line override those in a WITH file, so that you can make small variations on standard links by combining command line parameters and WITH files. Similarly, if you specify a parameter more than once in WITH files, the linker uses the first occurrence.

Note: In the second example below, this is true even if the first value given to a parameter is null.

Examples of WITH files and the corresponding ALINK calls:

```
ALINK WITH link-file
```

where "link-file" contains

```
FROM      obj/main,obj/s
TO        bin/test
LIBRARY   obj/lib
MAP
XREF      xo
```

is the same as specifying

```
ALINK FROM obj/main,obj/s TO bin/test LIBRARY obj/lib XREF xo
```

The command

```
ALINK WITH lkin LIBRARY ""
```

where 'lkin' contains

```
FROM      bin/prog,bin/subs
LIBRARY   nag/fortlib
TO        linklib/prog
```

is the same as the command line

```
ALINK FROM bin/prog,bin/subs TO linklib.prog
```

Note: In the example above, the null parameter for LIBRARY on the command line overrides the value "nag/fortlib" in the WITH file, and so the linker does not read any libraries.

4.2.3 Errors and Other Exceptions

Various errors can occur while the linker is running. Most of the messages are self-explanatory and refer to the failure to open files, or to errors in command or binary file format. After an error, the linker terminates at once.

There are a few messages that are warnings only. The most important

ones refer to undefined or multiply-defined symbols. The linker should not terminate after receiving a warning.

If any undefined symbols remain at the end of the first pass, the linker produces a warning, and outputs a table of such symbols. During the second pass, references to these symbols become references to **location zero.**

If the linker finds more than one definition of a symbol during the first pass, it puts out a warning, and ignores the later definition. The linker does not produce this message if the second definition occurs in a library file, so that you can replace library routines without it producing spurious messages. A serious error follows if the linker finds inconsistent symbol references, and linking then terminates at once.

Since the linker only uses the first definition of any symbol, it is important that you understand the following order in which files are read.

1. Primary (FROM or ROOT) input.
2. Overlay files.
3. LIBRARY files.

Within each group, the linker reads the files in the order that you specify in the file list. Thus definitions in the primary input override those in the overlay files, and those in the libraries have lowest priority.

4.2.4 MAP and XREF Output

The link map, which the linker produces after the first pass, lists all the code segments that the linker output to the load file in the second pass, in the order that they must be written.

For each code segment, the linker outputs a header, starting with the name of the file (truncated to eight letters), the code segment reference number, the type (that is, data, code, bss, or COMMON), and size. If the code segment was in an overlay file, the linker also gives the overlay level and overlay ordinate.

After the header, the linker prints each symbol defined in the code segment, together with its value. It prints the symbols in ascending order of their values, appending an asterisk (*) to absolute values.

The value of the WIDTH parameter determines the number of symbols printed per line. If this is too small, then the linker prints one symbol on each line.

The cross-reference output also lists each code segment, with the same header as in the map.

The header is followed by a list of the symbols with their references. Each reference consists of a pair of integers, giving the offset of the reference and the number of the code segment in which it occurs. The code segment number refers to the number given in each header.

4.3 Overlaying

The automatic overlay system provided by the linker and the overlay supervisor allows programs to occupy less memory when running, without any alterations to the program structure.

When using overlaying, you should consider the program as a **tree** structure. That is, with the **root** of the tree as the primary binary input, together with library code segments and COMMON blocks. This root is always resident in memory. The overlay files then form the other nodes of the tree, according to specifications in the OVERLAY directive.

The output from the linker when overlaying, as in the usual case, is a single binary file, which consists of all the code segments, together with information giving the location within the file of each node of the overlay tree. When you load the program only the root is brought into memory. An overlay supervisor takes care of loading and unloading the overlay segments automatically. The linker includes this overlay supervisor in the output file produced from a link using overlays. The overlay supervisor is invisible to the program running.

4.3.1 OVERLAY Directive

To specify the tree structure of a program to the linker, you use the OVERLAY directive. This directive is exceptional in that you can only use it in WITH files. As with other parameters, the linker uses the first OVERLAY directive you give it.

The format of the directive is

```
OVERLAY
Xfiles
.
.
.
#
```

Note: The overlay directive can span many lines. The linker recognizes a hash sign (#) or the end-of-file as a terminator for the directive.

Each line after OVERLAY specifies one node of the tree, and consists of a count "X" and a file list.

The level of a node specifies its "depth" in the tree, starting at zero, which is the level of the root. The count "X", given in the directive, consists of zero or more asterisks, and the overlay level of the node is given by X + 1.

As well as the level, each node other than the root has an **ordinate** value.

This refers to the order in which the linker should read the descendents of each node, and starts at 1, for the first "offspring" of a parent node.

Note: There may be nodes with the same level and ordinate, but with different parents.

While reading the OVERLAY directive, the linker remembers the current level, and, for each new node, compares the level specified with this value. If less, then the new node is a descendent of a previous one. If equal, the new node has the same parent as the current one. If greater, the new node is a direct descendant of the current one, and so the new level must be one greater than the current value.

A number of examples may help to clarify this:

Directive	Level	Ordinate	Tree
OVERLAY			ROOT
a	1	1	↙↓↘
b	1	2	a b c
c	1	3	
#			
OVERLAY			ROOT
a	1	1	↙ ↘
b	1	2	a b
*c	2	1	↙↓
*d	2	2	c d
#			
OVERLAY			ROOT
a	1	1	
b	1	2	↙↓ ↓ ↓↘
*c	2	1	a b e f 1
*d	2	2	↙↓ ↙↓↘
e	1	3	c d g h k
f	1	4	↙↓
*g	2	1	i j
*h	2	2	
**i	3	1	
**j	3	2	
*k	2	3	
1	1	5	
#			

Figure 4-A

The level and ordinate values given above refer to the node specified on the same line. Note that all the files given in the examples above could have been file lists. Single letters are for clarity. For example, Figure 4-B:

```
ROOT bin/mainaaa
OVERLAY
bin/mainbbb,bin/mainccc,bin/mainddd
*bin/makereal
*bin/trbblock,bin/transint,bin/transr*
 bin/transri
bin/outcode
#
```

Figure 4-B

specifies the tree in the following figure:

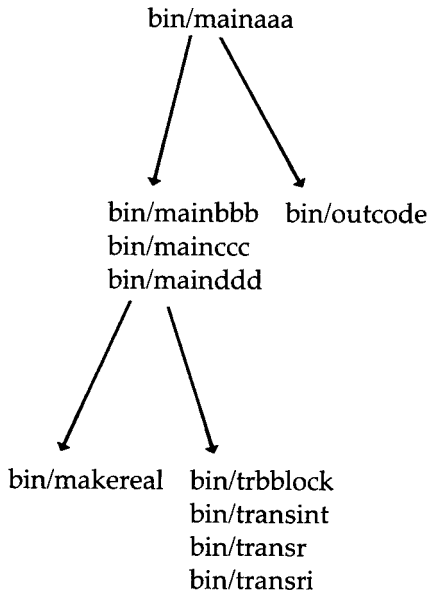


Figure 4-C

During linking, the linker reads the overlay files in the order you specified in the directive, line by line. The linker preserves this order in the map and cross reference output, and so you can deduce the exact tree structure from the overlay level and ordinate the linker prints with each code segment.

4.3.2 References to Symbols

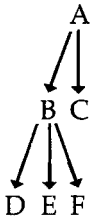
While linking an overlaid program, the linker checks each symbol reference for validity.

Suppose that the reference is in a tree node "R", and the symbol in a node "S". Then the reference is legal if one of the following is true.

(a) R and S are the same node.
(b) R is a descendent of S.
(c) R is the parent of S.

References of the third type above are known as **overlay references**. In this case, the linker enters the overlay supervisor when the program is run. The overlay supervisor then checks to see if the code segment containing the symbol is already in memory. If not, first the code segment, if any, at this level, and all its descendents are unloaded, and then the node containing the symbol is brought into memory. An overlaid code segment returns directly to its caller, and so is not unloaded from memory until another node is loaded on top of it.

For example, suppose that the tree is:



When the linker first loads the program, only A is in memory. When the linker finds a reference in A to a symbol in B, it loads and enters B. If B in turn calls D then again a new node is loaded. When B returns to A, both B and D are left in memory, and the linker does not reload them if the program requires them later. Now suppose that A calls C. First the linker unloads the code segments that it does not require, and which it may overwrite. In this case, these are B and D. Once it has reclaimed the memory for these, the linker can load C.

Thus, when the linker executes a given node, all the node's "ancestors", up to the root are in memory, and possibly some of its descendents.

4.3.3 Cautionary Points

The linker assumes that all overlay references are jumps or subroutine calls, and routes them through the overlay supervisor. Thus, you should not use overlay symbols as data labels.

Try to avoid impure code when overlaying because the linker does not always load a node that is fresh from the load file.

The linker gives each symbol that has an overlay reference an **overlay number.** It uses this value, which is zero or more, to construct the overlay supervisor entry label associated with that symbol. This label is of the form "OVLYnnnn", where nnnn is the overlay number. You should not use symbols with this format elsewhere.

The linker gathers together all program sections with the same section name. It does this so that it can then load them continuously in memory.

4.4 Error Codes and Messages

These errors should be rare. If they do occur, the error is probably in the compiler and not in your program. However, you should first check to see that you sent the linker a proper program (for example, an input program must have an introductory program unit that tells the linker to expect a program).

Invalid Object Modules

2 Invalid use of overlay symbol
3 Invalid use of symbol
4 Invalid use of common
5 Invalid use of overlay reference
6 Nonzero overlay reference
7 Invalid external block relocation
8 Invalid bss relocation
9 Invalid program unit relocation
10 Bad offset during 32 bit relocation
11 Bad offset during 6/8 bit relocation
12 Bad offset with 32 bit reference
13 Bad offset with 6/8 bit reference
14 Unexpected end of file
15 Hunk.end missing
16 Invalid termination of file
17 Premature termination of file
18 Premature termination of file

Internal Errors

19 Invalid type in hunk list
20 Internal error during library scan
21 Invalid argument freevector
22 Symbol not defined in second pass

Appendix

Console Input and Output on the Amiga

Note: Throughout this appendix, the characters "<CSI>" represent the Control Sequence Introducer. For output, you may either use the two character sequence Esc-[or the one byte value $9B (hex). For input, you receive $9Bs.

Introduction

This appendix describes several ways to do console (keyboard and screen) input and output on the Amiga. You can open the console as you would any other AmigaDOS file (with "*", "CON:", "RAW:") or do direct calls to console.library. The advantages of using each are listed below:

* "Asterisk" does not open any windows; it just uses the existing CLI window. You do not receive any complex character sequences. You do receive lower case letters a–z, uppercase letters A–Z, numbers, ASCII special symbols, and control characters. Basically, if a teletype can generate the character with a single keystroke, you can receive it. In addition to these characters, you can receive each of them with the high-order bit set ($80-$FF). Line editing is also performed for you. This means AmigaDOS accepts <BACKSPACE> and CTRL-X for character and line deletions. You do not have to deal with these. Any <CSI> sequence is swallowed for you as well as control characters: C, D, E, F, H, and X. Any <CR> or CTRL-M characters are converted to CTRL-J (new-line).

CON: Is just like "*" except that you also get to define a new window.

RAW: The simple case: With RAW: (as compared to CON:) you lose the line editing functions and you gain access to the function and arrow keys. These are sent as sequences of characters which you must parse in an intelligent manner.
The "complex" cases: By issuing additional commands to the console processor (by doing writes to RAW:), you can get even more detailed information. For example, you can request key

press and release information or data on mouse events. See "Selection of RAW Input Events" on page 224 for details on requesting this information.

console.device: With this method, you have full control over the console device. You may change the KeyMap to one of your own design and completely "redesign" your keyboard.

Helpful AmigaDOS Commands

Two very helpful AmigaDOS commands let you play with these functions. The first:

```
TYPE RAW:10/10/100/30/ opt h
```

accepts input from a RAW: window and displays the results in hex and ASCII. If you want to know for sure what characters the keyboard is sending, this command provides a very simple way.

The second:

```
COPY "RAW:10/10/100/30/RAW Input" "RAW:100/10/200/100/RAW Output"
```

lets you type sequences into the input window and watch the cursor movements in the output window. COPY cannot detect end-of-file on RAW: input, so you have to reboot when you are finished with this command.

CON Keyboard Input

If you read from the CON: device, the keyboard inputs are preprocessed for you.

You get the ASCII characters like "B". Most normal text gathering programs read from the CON: device. Special programs like word processors and music keyboard programs use RAW:.

To generate the international and special characters at the keyboard, you can press either ALT key. This sets the high bit of the ASCII code returned for the key pressed.

Generating $FF (umlaut y) is a special case. If it followed the standard convention, it would be generated by ALT-DEL. But since the ASCII code <Del> (hex 7F) is not generally a printable character and it is our philosophy that Alt-nonprinting character should not generate a printing character, we have substituted ALT-numeric pad"-".

Table A-1 lists the characters you can display on the Amiga. The characters NBSP (nonbreak space) and SHY (soft hyphen) are used to render a space and hyphen in text processing with additional meaning about the properties of the character.

b_4 b_3 b_2 b_1	b_8 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	b_7 0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	b_6 0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	b_5 0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0 0 0 0	00		SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð
0 0 0 1	01		!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
0 0 1 0	02		"	2	B	R	b	r			¢	²	Â	Ò	â	ò
0 0 1 1	03		#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
0 1 0 0	04		$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
0 1 0 1	05		%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
0 1 1 0	06		&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
0 1 1 1	07		'	7	G	W	g	w			§	·	Ç	×	ç	÷
1 0 0 0	08		(	8	H	X	h	x			¨	¸	È	Ø	è	ø
1 0 0 1	09		)	9	I	Y	i	y			©	¹	É	Ù	é	ù
1 0 1 0	10		*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
1 0 1 1	11		+	;	K	[	k	{			«	»	Ë	Û	ë	û
1 1 0 0	12		,	<	L	\	l	|		¬	¼	Ì	Ü	ì	ü
1 1 0 1	13		-	=	M	]	m	}			SHY	½	Í	Ý	í	ý
1 1 1 0	14		.	>	N	^	n	~			®	¾	Î	Þ	î	þ
1 1 1 1	15		/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ

Table A-1: International Character Code

Note: AmigaDOS uses CON: input for the CLI and most other commands. When it does this, it filters out ALL of the function key and cursor key inputs. Programs that run under AmigaDOS can (and some do) still open the RAW: console handler and process function key input.

Note: "NBSP" is a nonbreak sequence.
"SHY" is a soft-hyphen.

CON Screen Output

CON: screen output is just like RAW: screen output except that <LF> (hex 0A) is translated into a newline character. The net effect is that the cursor moves to the first column of the next line whenever a <LF> is displayed.

RAW Screen Output

ANSI ×3.64 CODES SUPPORTED For writing text to the display:

Independent Control Functions (no introducer):

Ctrl	Hex	Name	Definition	
H	08	BS	BACKSPACE	Move the cursor left 1 column
I	09	TAB	TAB	Move right 1 column
J	0A	LF	LINE FEED	
K	0B	VT	VERTICAL TAB	Move cursor up 1, scroll if necessary
L	0C	FF	FORM FEED	Clear the screen
M	0D	CR	CARRIAGE RETURN	Move to first column
N	0E	SO	SHIFT OUT	Set MSB of each character before displaying
O	OF	SI	SHIFT IN	Undo SHIFT OUT
[	1B	ESC	ESCAPE	See below

Precede the following characters with <ESC> to perform the indicated actions.

Chr	Name	Definition
c	RIS	RESET TO INITIAL STATE

Precede the following characters with <Esc> or press CTRL-ALT and the letter to perform the indicated actions.

Hex	Chr	Name	Definition
845tD	IND	INDEX:	move the active position down one line
85	E	NEL	NEXT LINE:
8D	M	RI	REVERSE INDEX:
9B	[	CSI	CONTROL SEQUENCE INTRODUCER: see next list

Control sequences (introduced by <CSI>) with parameters. The first character in the following table (under the <CSI> column) represents the number of allowable parameters, as follows:

"0" indicates no parameters allowed.
"1" indicates 0 or 1 numeric parameters.
"2" indicates 2 numeric parameters ("14;94").
"3" indicates any number of numeric parameters, separated by semicolons.
"4" indicates exactly 4 numeric parameters.
"8" indicates exactly 8 numeric parameters.

<CSI>	Name	Definition	
1 @	ICH	INSERT CHARACTER	Inserts 1 or more spaces, shifting the remainder of the line to the right.
1 A	CUU	CURSOR UP	
1 B	CUD	CURSOR DOWN	
1 C	CUF	CURSOR FORWARD	
1 D	CUB	CURSOR BACKWARD	
1 E	CNL	CURSOR NEXT LINE	Down n lines to column 1
1 F	CPL	CURSOR PRECEDING LINE	Up n lines to column 1
2 H	CUP	CURSOR POSITION	"<CSI>row;columnH"
1 J	ED	ERASE IN DISPLAY	(only to end of display)
1 K	EL	ERASE IN LINE	(only to eol)
1 L	IL	INSERT LINE	Inserts a blank line BEFORE the line containing the cursor.
1 M	DL	DELETE LINE	Removes the current line. Moves all lines below up by one. Blanks the bottom line.
1 P	DCH	DELETE CHARACTER	
2 R	CPR	CURSOR POSITION REPORT	(in Read stream only) Format of report: "<CSI>row;columnR"
1 S	SU	SCROLL UP	Removes line from top of screen. Moves all other lines up one. Blanks last line.
1 T	SD	SCROLL DOWN	Removes line from bottom of screen. Moves all other lines down one. Blanks top line.
3 h	SM	SET MODE	<CSI>2Oh causes RAW: to convert <LF> to <newline> on output.
3 1	RM	RESET MODE	<CSI>201 undoes SET MODE 20
3 m	SGR	SELECT GRAPHIC RENDITION	
1 n	DSR	DEVICE STATUS REPORT	

The following are not ANSI standard sequences; rather, they are private Amiga sequences.

```
1 t   aSLPP SET PAGE LENGTH
1 u   aSLL SET LINE LENGTH
1 x   aSLO SET LEFT OFFSET
1 y   aSTO SET TOP OFFSET
3 {   aSRE SET RAW EVENTS
8 |   aIER INPUT EVENT REPORT (read)
3 }   aRRE RESET RAW EVENTS
1 ~   aSKR SPECIAL KEY REPORT (read)
1 p   aSCR SET CURSOR RENDITION
      <Esc> p turns the cursor off
0 q   aWSR WINDOW STATUS REQUEST
4 r   aWBR WINDOW BOUNDS REPORT (read)
```

Examples:

Move cursor right by 1:

<CSI>C or <Tab> or <CSI>1C

Move cursor right by 20:

<CSI>20C

Move cursor to upper left corner (home):

<CSI>H or <CSI>1;1H or <CSI>;1H or <CSI>1;H

Move cursor to the fourth column of the first line of the window:

<CSI>1;4H or <CSI>;4H

Clear the screen:

```
<FF> or CTRL-L          {clear screen character} or
<CSI>H<CSI>J            {home and clear to end of screen} or
<CSI>H<CSI>23M          {home and delete 23 lines} or
<CSI>1;1H<CSI>23L       {home and insert 23 lines}
```

RAW Keyboard Input

Reading input from the RAW: console device returns an ANSI ×3.64 standard byte stream. This stream may contain normal characters and/or RAW input event information. You may also request other RAW input events using the SET RAW EVENTS (aSRE) and RESET RAW EVENTS (aRRE) control sequences discussed below. See "Selection of RAW Input Events" below for details.

If you issue a RAW input request and there is no pending input, the read command waits until some input is received. You can test for characters pending by doing "WaitforChar" requests.

In the default state, the function and arrow keys cause the following sequences to be sent to your process:

Key	Unshifted Sends	Shifted Sends
F1	<CSI>0~	CSI>10~
F2	<CSI>1~	<CSI>11~
F3	<CSI>2~	<CSI>12~
F4	<CSI>3~	<CSI>13~
F5	<CSI>4~	<CSI>14~
F6	<CSI>5~	<CSI>15~
F7	<CSI>6~	<CSI>16~
F8	<CSI>7~	<CSI>17~
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
HELP	<CSI>?~	<CSI>?~ (same)

Arrow keys:

Up	<CSI>A	<CSI>T~
Down	<CSI>B	<CSI>S~
Left	<CSI>C	<CSI> A~ (note space)
Right	<CSI>D	<CSI> @~ (note space)

Selection of RAW Input Events:

If you are using RAW by default, you get the ANSI data and control sequences mentioned above. If this does not give you enough information about input events, you can request additional information from the console driver.

If, for example, you need to know when each key is pressed and released, you would request "RAW keyboard input." This is done by writing "<CSI>1{" to the console. The following is a list of valid RAW input requests:

RAW Input Event Types

Request Number	Description	
0	nop	Used internally.
1	RAW keyboard input	
2	RAW mouse input	
3	Event	Sent whenever your window is made active.
4	Pointer position	
5	(unused)	
6	Timer	
7	Gadget pressed	
8	Gadget released	
9	Requester activity	
10	Menu numbers	
11	Close Gadget	
12	Window resized	
13	Window refreshed	
14	Preferences changed	
15	Disk removed	
16	Disk inserted	

If you select any of these events, you start to get information about the events in the following form:

```
<CSI><class>
<subclass>
<keycode>
<qualifiers>
<x>
<y>
<seconds>
<microseconds>
```

<CSI> is a one byte field. It is the Control Sequence Introducer, 9B hex.
<class> is the RAW input event type, from the above table.
<subclass> is not currently used and is always zero (0).
<keycode> indicates which key number was pressed (see Figure A-1 and Table A-2). This field can also be used for mouse information.
The <qualifiers> field indicates the state of the keyboard and system. The qualifiers are defined as follows:

Bit	Mask	Key	
0	0001	left shift	
1	0002	right shift	
2	0004	caps lock	* special, see below
3	0008	control	
4	0010	left alt	
5	0020	right alt	
6	0040	left Amiga key pressed	
7	0080	right Amiga key pressed	
8	0100	numeric pad	
9	0200	repeat	
10	0400	interrupt	Not currently used
11	0800	multi broadcast	This (active) or all windows
12	1000	left mouse button	
13	2000	right mouse button	
14	4000	middle mouse button (not available on std mouse)	
15	8000	relative mouse	Indicates mouse coordinates are relative, not absolute

The CAPS LOCK key is handled in a special manner. It only generates a keycode when it is pressed, not when it is released. However, the up and down bit (80 hex) is still used and reported. If pressing the CAPS LOCK key turns on the LED, then key code 62 (CAPS LOCK pressed) is sent. If pressing the CAPS LOCK key extinguishes the LED, then key code 190 (CAPS LOCK released) is sent. In effect, the keyboard reports this key being held down until it is struck again.

The <seconds> and <microseconds> fields are system time stamp taken at the time the event occurred. These values are stored as long words by the system and as such could (theoretically) reach 4 billion.

With RAW: keyboard input, selected keys no longer return a simple 1 character "A" to "Z" but rather return raw keycode reports with the following form:

```
<CSI>1;0;<keycode>;<qualifiers>;0;0;<secs>;<microsecs>|
```

For example, if the user pressed and released the "B" key with the left SHIFT and right Amiga keys also pressed, you might receive the following data:

```
<CSI>1;0;35;129;0;0;23987;99|
<CSI>1;0;163;129;0;0;24003;18|
```

The "0;0;" fields are not used for keyboard input but are, rather, used if you select mouse input. For mouse input, these fields would indicate the X and Y coordinates of the mouse.

The <keycode> field is an ASCII decimal value representing the key pressed or released. Adding 128 to the pressed key code results in the released keycode. Figure A-1 lets you convert quickly from a key to its keycode. Table A-2 lets you convert quickly from a keycode to a key.

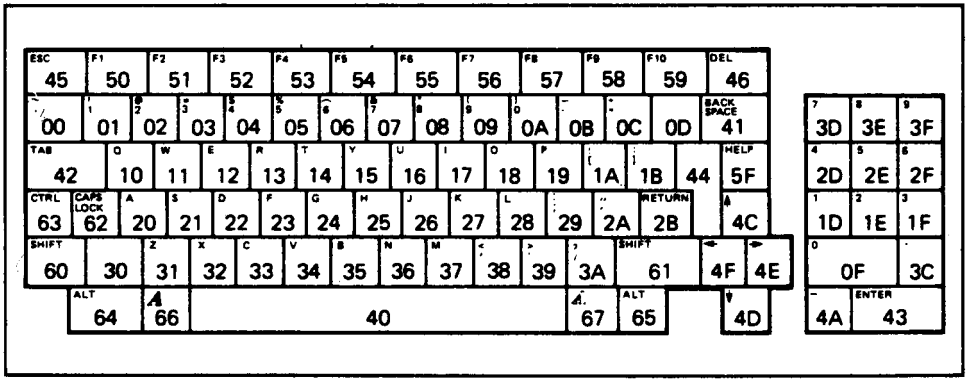


Figure A-1: Reduced copy of keyboard template

The default values given in the following correspond to:

1) The values the CON: device returns when these keys are pressed, and
2) The keycaps as shipped with the standard American keyboard.

Table A-2: Converting from Keycodes to Keys

Raw Key Number	Unshifted Default Value	Shifted Default Value
00	´<Accent grave>	~ <tilde>
01	1	!
02	2	@
03	3	#
04	4	$
05	5	%
06	6	^
07	7	&
08	8	*
09	9	(
0A	0	)
0B	- <Hyphen>	_<Underscore>
0C	=	+
0D	\	|
0E	undefined	
0F	0	0 <Numeric pad>
10	Q	q
11	W	w
12	E	e
13	R	r
14	T	t
15	Y	y
16	U	u
17	I	i
18	O	o
19	P	p
1A	{	[
1B	}	]
1C	undefined	
1D	1	1 <Numeric pad>
1E	2	2 <Numeric pad>
1F	3	3 <Numeric pad>

Raw Key Number	Unshifted Default Value	Shifted Default Value
20	A	a
21	S	s
22	D	d
23	F	f
24	G	g
25	H	h
26	J	j
27	K	k
28	L	l
29	:	;
2A	"	' <single quote>
2B	<RESERVED>	(RESERVED)
2C	undefined	
2D	4	4 <Numeric pad>
2E	5	5 <Numeric pad>
2F	6	6 <Numeric pad>
30	<RESERVED>	(RESERVED)
31	Z	z
32	X	x
33	C	c
34	V	v
35	B	b
36	N	n
37	M	m
38	<	, <comma>
39	>	. <period>
3A	?	/
3B	undefined	
3C	.	. <Numeric pad>
3D	7	7 <Numeric pad>
3E	8	8 <Numeric pad>
3F	9	9 <Numeric pad>
40	Space	
41	BACKSPACE	
42	TAB	
43	ENTER	ENTER <Numeric pad>
44	RETURN	
45	Escape	<Esc>
46	DEL	

Raw Key Number	Unshifted Default Value	Shifted Default Value
47	undefined	
48	undefined	
49	undefined	
4A	-	- <Numeric pad>
4B	undefined	
4C	Cursor Up	Scroll down
4D	Cursor Down	Scroll up
4E	Cursor Forward	Scroll left
4F	Cursor Backward	Scroll right
50	F1	<CSI>10~
51	F2	<CSI>11~
52	F3	<CSI>12~
53	F4	<CSI>13~
54	F5	<CSI>14~
55	F6	<CSI>15~
56	F7	<CSI>16~
57	F8	<CSI>17~
58	F9	<CSI>18~
59	F10	<CSI>19~
5A	undefined	
5B	undefined	
5C	undefined	
5D	undefined	
5E	undefined	
5F	Help	
60	SHIFT <left of space bar>	
61	SHIFT <right of space bar>	
62	Caps Lock	
63	Control	
64	Left Alt	
65	Right Alt	
66	"Amiga" <left of space bar>	
67	"Amiga" <right of space bar>	
68	Left Mouse Button	
	<not converted>	Inputs are only for the
69	Right Mouse Button	
	<not converted>	mouse connected to Intuition,

Raw Key Number	Unshifted Default Value	Shifted Default Value
6A	Middle Mouse Button <not converted>	currently "gameport" one.
6B	undefined	
6C	undefined	
6D	undefined	
6E	undefined	
6F	undefined	
70-7F	undefined	
80-F8	Up transition <release or unpress key> of one of the above keys. 80 for 00, F8 for 7F.	
F9	Last keycode was bad (was sent in order to resync)	
FA	Keyboard buffer overflow.	
FB	undefined; reserved for keyboard processor catastrophe	
FC	Keyboard self-test failed.	
FD	Power-up key stream start. Keys pressed or stuck at power-up are sent between FD and FE.	
FE	Power-up key stream end.	
FF	(undefined, reserved)	
FF	Mouse event, movement only, No button change. <not converted>	

Notes about the preceding table:

1) "undefined" indicates that the current keyboard design should not generate this number. If you are using "SetKeyMap" to change the key map, the entries for these numbers must still be included.
2) The "<not converted>" refers to mouse button events. You must use the sequence "<CSI>2{" to inform the console driver that you wish to receive mouse events; otherwise, these are not transmitted.
3) "(RESERVED)" indicates that these keycodes have been reserved for non-U.S. keyboards. The "2B" code key is between the double quote and return keys. The "30" code key is between the SHIFT and "Z" keys.

AmigaDOS Technical Reference Manual

Contents

Chapter 1

The Filing System

This chapter describes the AmigaDOS filing system. It includes information on how to patch a disk corrupted by hardware errors.

1.1 AmigaDOS File Structure

The AmigaDOS file handler uses a disk that is formatted with blocks of equal size. It provides an indefinitely deep hierarchy of directories, where each directory may contain other directories and files, or just files. The structure is a pure tree—that is, loops are not allowed.

There is sufficient redundancy in the mechanism to allow you to patch together most, if not all, of the contents of a disk after a serious hardware error, for example. To patch the contents of a disk, you use the DISKED command. For further details on the syntax of DISKED, see Section 1.2, "DISKED—The Disk Editor," later in this chapter. Before you can patch together the contents of a disk, you must understand the layout. The subsections below describe the layout of disk pages.

1.1.1 Root Block

The root of the tree is the Root Block, which is at a fixed place on the disk. The root is like any other directory, except that it has no parent, and its secondary type is different. AmigaDOS stores the name of the disk volume in the name field of the root block.

Each filing system block contains a checksum, where the sum (ignoring overflow) of all the words in the block is zero.

The figure below describes the layout of the root block.

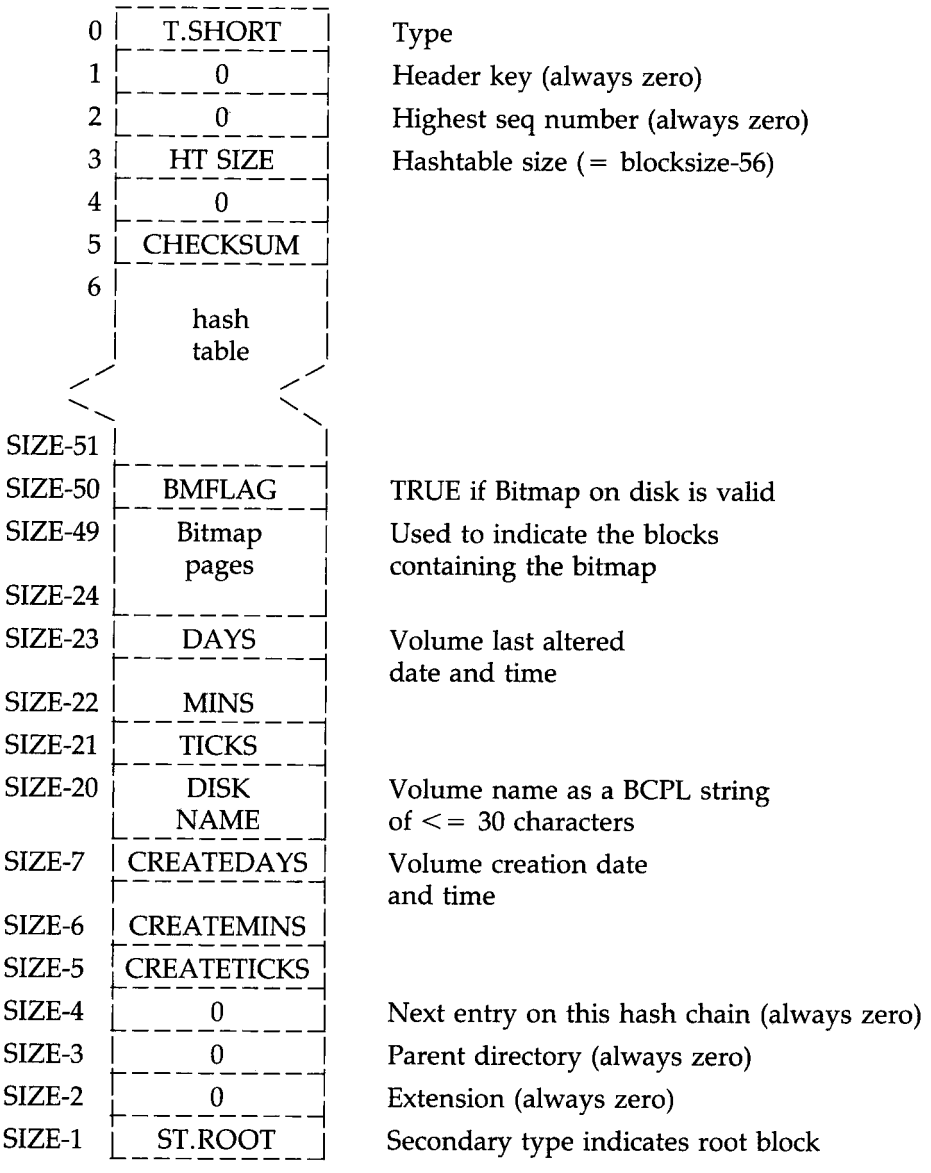


Figure 1-A: Root Block

1.1.2 User Directory Blocks

The following figure describes the layout of the contents of a user directory block.

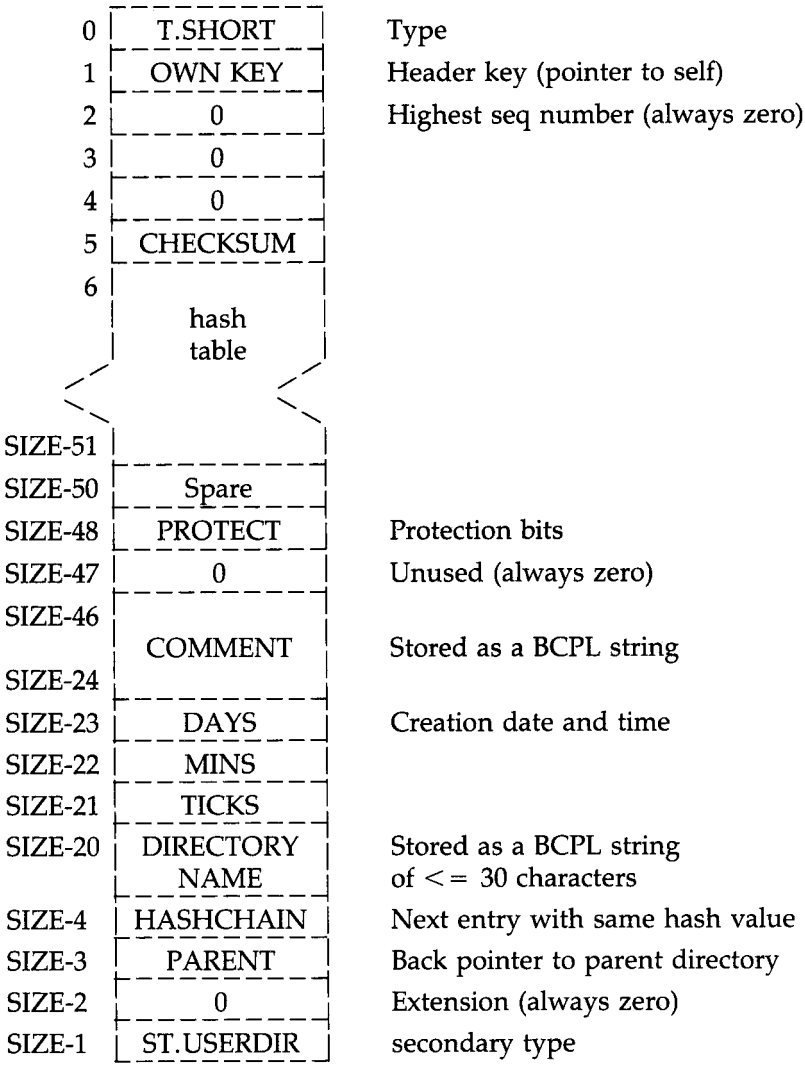


Figure 1-B: User Directory Blocks

User directory blocks have type T.SHORT and secondary type ST.USER-DIRECTORY. The six information words at the start of the block also indicate the block's own key (that is, the block number) as a consistency check and the size of the hash table. The 50 information words at the end of the block contain the date and time of creation, the name of the directory, a pointer to the next file or directory on the hash chain, and a pointer to the directory above.

To find a file or subdirectory, you must first apply a hash function to its name. This hash function yields an offset in the hash table, which is the key of the first block on a chain linking those with the same hash value (or zero, if there are none). AmigaDOS reads the block with this key and compares the name of the block with the required name. If the names do not match, it reads the next block on the chain, and so on.

1.1.3 File Header Block

The following figure describes the layout of the file header block.

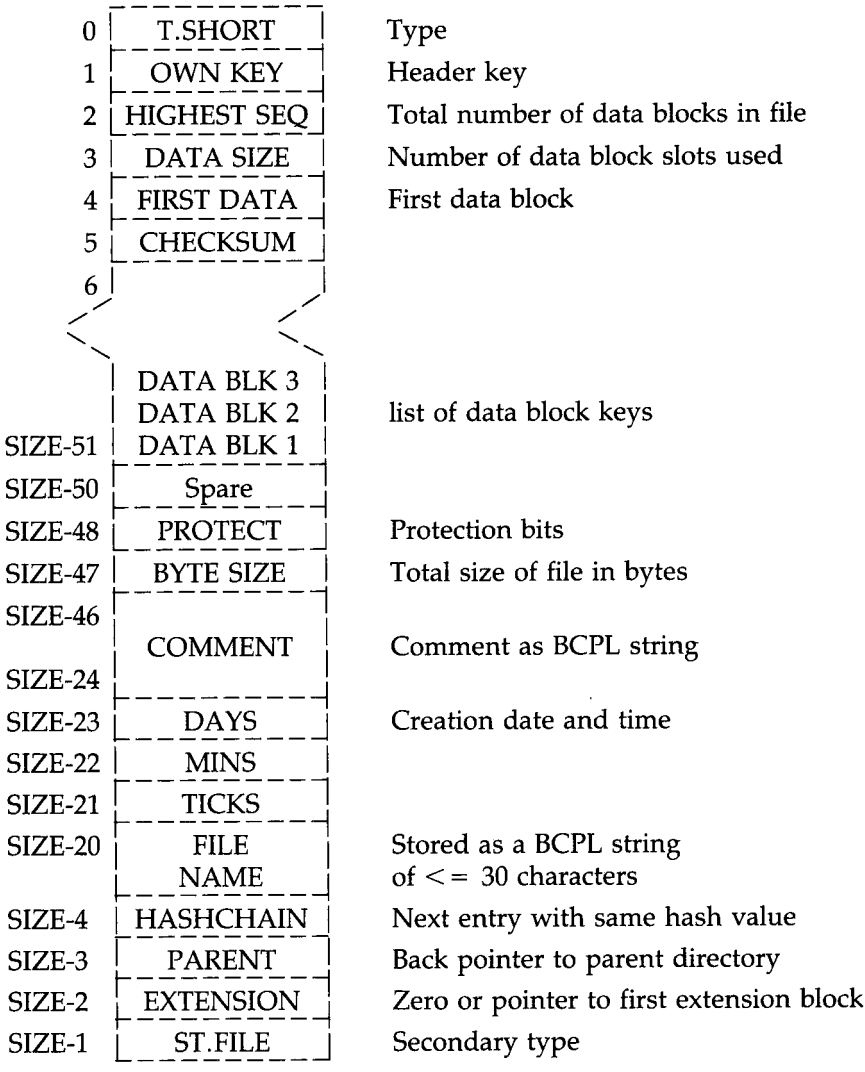


Figure 1-C: File Header Block

Each terminal file starts with a file header block, which has type T.SHORT and secondary type ST.FILE. The start and end of the block contain name, time, and redundancy information similar to that in a directory block. The body of the file consists of Data blocks with sequence numbers from 1 upward. AmigaDOS stores the addresses of these blocks in consecutive words downward from offset size-51 in the block. In general, AmigaDOS does not use all the space for this list and the last data block is not full.

1.1.4 File List Block

If there are more blocks in the file than can be specified in the block list, then the EXTENSION field is nonzero and points to another disk block which contains a further data block list. The following figure explains the structure of the file list block.

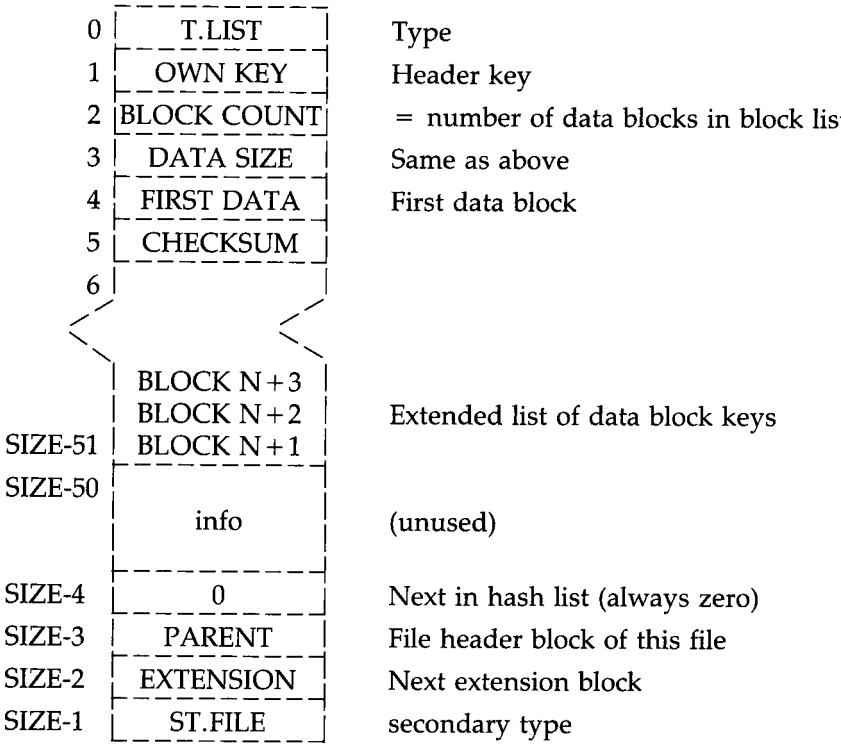


Figure 1-D: File List Block

There are as many file extension blocks as required to list the data blocks that make up the file. The layout of the block is very similar to that of a file header block, except that the type is different and the date and filename fields are not used.

1.1.5 Data Block

The following figure explains the layout of a data block.

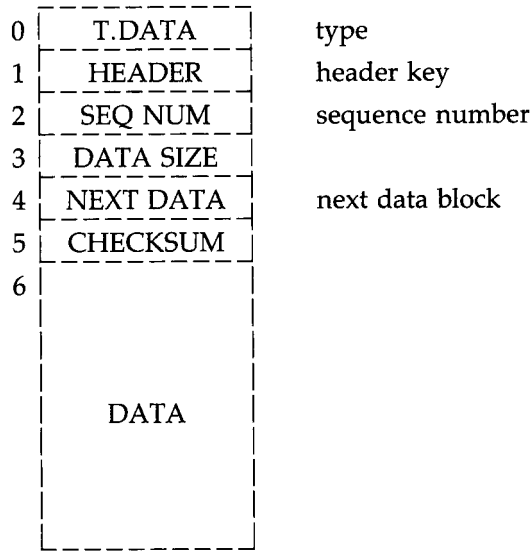


Figure 1-E: Data Block

Data blocks contain only six words of filing system information. These six words refer to the following:

- type (T.DATA)
- pointer to the file header block
- sequence number of the data block
- number of words of data
- pointer to the next data block
- checksum

Normally, all data blocks except the last are full (that is, they have a size = blocksize-6). The last data block has a forward pointer of zero.

1.2 DISKED—The Disk Editor

To inspect or patch disk blocks, you may use the AmigaDOS disk editor, DISKED. Because DISKED writes to the disk directly, you should use it with care. Nevertheless, you can use it to good effect in recovering information from a corrupt floppy disk, for example. A disk does not have to be inserted to be examined by DISKED.

You should only use DISKED with reference to the layout of an AmigaDOS disk. (For a description of the layout, see Subsections 1.1.1 through 1.1.5 in the first part of this chapter.) DISKED knows about this structure—for example, the R (Root Block) command prints the key of the root block. The G (Get block) command followed by this key number reads the block into memory, whereupon the I (Information) command prints out the information contained in the first and last locations, which indicate the type of block, the name, the hash links, and so on. If you specify a name after an H (Hash) command, DISKED gives you the offset on a directory page that stores as the first key headers with names that hash to the name you supplied. If you then type the number that DISKED returns followed by a slash (/), DISKED displays the key of that header page. You can then read this with further G commands, and so on.

Consider deleting a file that, due to hardware errors, makes the filing system restart process fail. First, you must locate the directory page that holds the reference to the file. You do this by searching the directory structure from the root block, using the hash codes. Then, you must locate the slot that references the file—this is either the directory block or a header block on the same hash chain. This slot should contain the key of the file's header block. To set the slot to zero, you type the slot offset, followed by a slash (/) followed by zero (that is, <offset>/0). Then correct the checksum with the K (checKsum) command. You should disable the write protection with X and write back the updated block with P (for Put block) or W (for Windup). There is no need to do anything else, as the blocks that the file used in error become available once more after the RESTART process has successfully scanned the disk.

DISKED commands are all single characters, sometimes with arguments.

The following is a complete list of the available commands.

The following is a complete list of the available commands.

Command	Function
B n	Set logical block number base to n
C n	Display n characters from current offset
G [n]	Get block n from disk (default is the current block number)
H name	Calculate Hash value of name
I	Display block Information
K	Check block checKsum (and correct if wrong)
L[lwb upb]	Locate words that match Value under Mask (lwb and upb restrict search)
M n	Set Mask (for L and N commands) to n
N[lwb upb]	Locate words that do not match Value under Mask
P n	Put block in memory to block n on disk (default is the current block number)
R	Display block number of root block
Q	Quit (do not write to disk)
S char	Set display Style char = C -> characters S -> string O -> octal X -> hex D -> decimal
T lwb upb	Type range of offsets in block
V n	Set Value for L and N commands
W	Windup (=PQ)
X	Invert write protect state
Y n	Set cYlinder base to n
Z	Zero all words of buffer
number	Set current word offset in block = Display values set in program
/[n]	Display word at current offset or update value to n
'chars'	Put chars at current offset
"chars"	Put string at current offset

Table 1-A: DISKED Commands

To indicate octal or hex, you can start numbers with # or #X (that is, # for octal, #X for hex). You can also include BCPL string escapes (*N and so forth) in strings.

Chapter 2

Amiga Binary File Structure

This chapter describes:

2.1 Introduction

Chapter 2 details the structure of Binary Object files for the Amiga, as produced by assemblers and compilers. It also describes the format of Binary Load files, which are produced by the linker and read into memory by the loader. The format of load files supports overlaying. Apart from describing the

format of load files, this chapter explains the use of common symbols, absolute external references, and program units.

2.1.1 Terminology

Some of the technical terms used in this chapter are explained below.

External References

You can use a name to specify a reference between separate program units. The data structure lets you have a name longer than 16M bytes, although the linker restricts names to 255 characters. When you link the object files into a single load file, you must ensure that all external references match corresponding external definitions. The external reference may be of byte size, word, or long word; external definitions refer to relocatable values, absolute values, or resident libraries. Relocatable byte and word references refer to PC relative address modes and these are entirely handled by the linker. However, if you have a program containing long word relocatable references, relocation may take place when you load the program.

Note that these sizes only refer to the length of the relocation field; it is possible to load a word from a long external address, for example, and the linker makes no attempt to check that you are consistent in your use of externals.

Object File

An assembler or compiler produces a binary image, called an object file. An object file contains one or more program units. It may also contain external references to other object files.

Load File

The linker produces a binary image from a number of object files. This binary image is called a load file. A load file does not contain any unresolved external references.

Program Unit

A program unit is the smallest element the linker can handle. A program unit can contain one or more hunks; object files can contain one or more program units. If the linker finds a suitable external reference within a program unit when it inspects the scanned libraries, it includes the entire program unit in the load file. An assembler usually produces a single program unit from one assembly (containing one or more hunks); a compiler such as FORTRAN produces a program unit for each subroutine, main program, or Block Data. Hunk numbering starts from zero within

each program unit; the only way you can reference other program units is through external references.

Hunks

A hunk consists of a block of code or data, relocation information, and a list of defined or referenced external symbols. Data hunks may specify initialized data or uninitialized data (bss). bss hunks may contain external definitions but no external references nor any values requiring relocation. If you place initialized data blocks in overlays, the linker should not normally alter these data blocks, since it reloads them from disk during the overlay process. Hunks may be named or unnamed, and they may contain a symbol table in order to provide symbolic debugging information. They may also contain further debugging information for the use of high level language debugging tools. Each hunk within a program unit has a number, starting from zero.

Resident Library

Load files are also known as "libraries". Load files may be resident in memory; alternatively, the operating system may load them as part of the "library open" call. You can reference resident libraries through external references; the definitions are in a hunk containing no code, just a list of resident library definitions. Usually, to produce these hunks, you assemble a file containing nothing but absolute external definitions and then pass it through a special software tool to convert the absolute definitions to resident library definitions. The linker uses the hunk name as the name of the resident library, and it passes this through into the load file so that the loader can open the resident library before use.

Scanned Library

A scanned library consists of object files that contain program units which are only loaded if there are any outstanding external references to them. You may use object files as libraries and provide them as primary input to the linker, in which case the input includes all the program units the object files contain. Note that you may concatenate object files.

Node

A node consists of at least one hunk. An overlaid load file contains a root node, which is resident in memory all the time that the program is running, and a number of overlay nodes which are brought into memory as required.

2.2 Object File Structure

An object file is the output of the assembler or a language translator. To use an object file, you must first resolve all the external references. To do this, you pass the object file through the linker. An object file consists of one or more program units. Each program unit starts with a header and is followed by a series of hunks joined end to end, each of which contains a number of "blocks" of various types. Each block starts with a long word which defines its type, and this is followed by zero or more additional long words. Note that each block is always rounded up to the nearest long word boundary. The program unit header is also a block with this format.

The format of a program unit is as follows:

- Program unit header block
- hunks

The basic format of a hunk is as follows:

- hunk name block
- Relocatable block
- Relocation information block
- External symbol information block
- Symbol table block
- Debug block
- End block

You may omit all these block types, except the end block.

The following subsections describe the format of each of these blocks. The value of the type word appears in decimal and hex after the type name, for example, hunk_unit has the value 999 in decimal and 3E7 in hex.

2.2.1 hunk_unit (999/3E7)

This specifies the start of a program unit. It consists of a type word, followed by the length of the unit name in long words, followed by the name itself padded to a long word boundary with zeros, if required. In diagramatic form, the format is as follows:

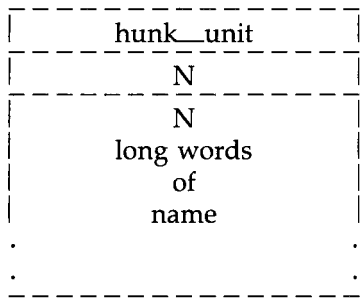


Figure 2-A: hunk_unit (999/3E7)

2.2.2 *hunk_name (1000/3E8)*

This defines the name of a hunk. Names are optional; if the linker finds two or more named hunks with the same name, it combines the hunks into a single hunk. Note that 8- or 16-bit program counter relative external references can only be resolved between hunks with the same name. Any external references in a load format file are between different hunks and require 32-bit relocatable references; although, as the loader scatterloads the hunks into memory, you cannot assume that they are within 32K of each other. Note that the length is in long words and the name block, like all blocks, is rounded up to a long word boundary by padding with zeros. The format is as follows:

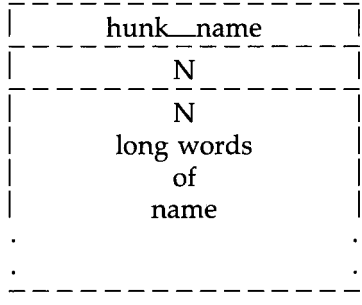


Figure 2-B: hunk_name (1000/3E8)

2.2.3 *hunk_code (1001/3E9)*

This defines a block of code that is to be loaded into memory and possibly relocated. Its format is as follows:

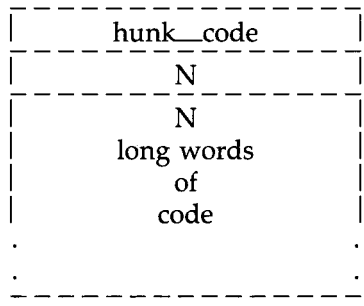


Figure 1-C: hunk_code (1001/3E9)

2.2.4 hunk_data (1002/3EA)

This defines a block of initialized data which is to be loaded into memory and possibly relocated. The linker should not alter these blocks if they are part of an overlay node, as it may need to reread them from disk during overlay handling. The format is as follows:

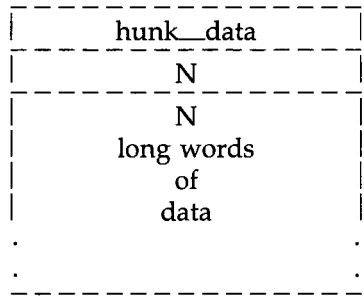


Figure 1-D: hunk_data (1002/3EA)

2.2.5 hunk_bss (1003/3EB)

This specifies a block of uninitialized workspace which is allocated by the loader. bss blocks are used for such things as stacks and for FORTRAN COMMON blocks. It is not possible to relocate inside a bss block, but symbols can be defined within one. Its format is as follows:

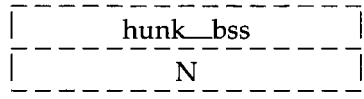


Figure 1-E: hunk_bss (1003/3EB)

where N is the size of block you require in long words. The memory used for bss blocks is zeroed by the loader when it is allocated.

The relocatable block within a hunk must be one of hunk_code, hunk_data, or hunk_bss.

2.2.6 *hunk_reloc32 (1004/3EC)*

A hunk_reloc32 block specifies 32-bit relocation that the linker is to perform within the current relocatable block. The relocation information is a reference to a location within the current hunk or any other within the program unit. Each hunk within the unit is numbered, starting from zero. The linker adds the address of the base of the specified hunk to each of the long words in the preceding relocatable block that the list of offsets indicates. The offset list only includes referenced hunks and a count of zero indicates the end of the list. Its format is as follows:

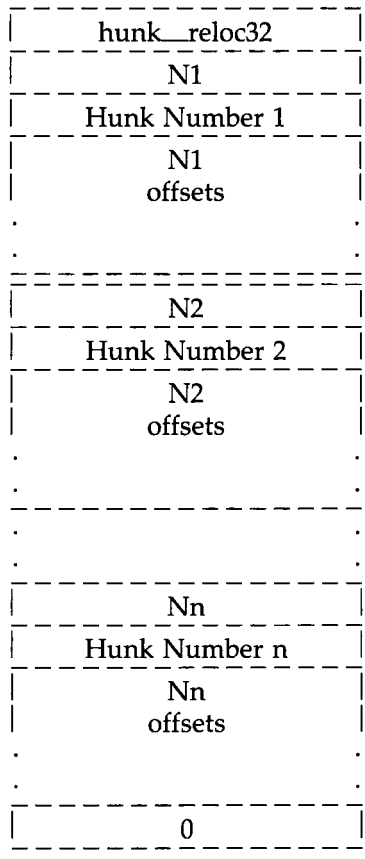


Figure 2-F: hunk_reloc32 (1004/3EC)

2.2.7 hunk_reloc16 (1005/3ED)

A hunk_reloc 16 block specifies 16-bit relocation that the linker should perform within the current relocatable block. The relocation information refers to 16 bit program counter relative references to other hunks in the program unit. The format is the same as hunk_reloc32 blocks. These references must be to hunks with the same name, so that the linker can perform the relocation while it coagulates (that is, gathers together) similarly named hunks.

2.2.8 hunk_reloc8 (1006/3EE)

A hunk_reloc8 block specifies 8-bit relocation that the linker should perform within the current relocatable block. The relocation information refers to 8-bit program counter relative references to other hunks in the program unit. The format is the same as hunk_reloc32 blocks. These references must be to hunks with the same name, so that the linker can perform the relocation while it coagulates similarly named hunks.

2.2.9 hunk_ext (1007/3EF)

This block contains external symbol information. It contains entries both defining symbols and listing references to them. Its format is as follows:

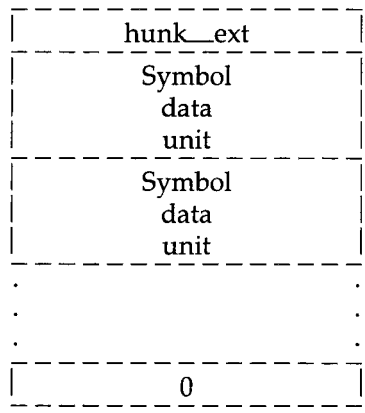


Figure 2-G: hunk_ext (1007/3EF)

where there is one "symbol data unit" for each symbol used, and the block ends with a zero word.

Each symbol data unit consists of a type byte, the symbol name length (three bytes), the symbol name itself, and further data. You specify the symbol name length in long words, and pad the name field to the next long word boundary with zeros.

The type byte specifies whether the symbol is a definition or a reference, and so forth. AmigaDOS uses values 0-127 for symbol definitions, and 128-255 for references.

At the moment, the values are as follows:

Name	Value	Meaning
ext_symb	0	Symbol table—see symbol block below
ext_def	1	Relocatable definition
ext_abs	2	Absolute definition
ext_res	3	Resident library definition
ext_ref32	129	32-bit reference to symbol
ext_common	130	32-bit reference to COMMON
ext_ref16	131	16-bit reference to symbol
ext_ref8	132	8-bit reference to symbol

Table 2-A: External Symbols

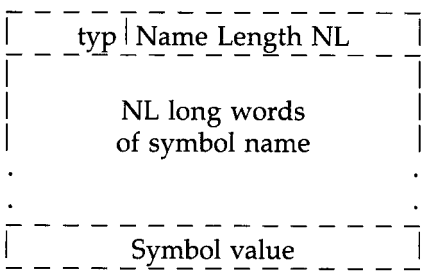
The linker faults all other values. For ext_def there is one data word, the value of the symbol. This is merely the offset of the symbol from the start of the hunk. For ext_abs there is also one data value, which is the absolute value to be added into the code. The linker treats the value for ext_res in the same way as ext_def, except that it assumes the hunk name is the library name and it copies this name through to the load file. The type bytes ext_ref32, ext_ref16, and ext_ref8 are followed by a count and a list of references, again specified as offsets from the start of the hunk.

The type ext_common has the same structure except that it has a COMMON block size before the count. The linker treats symbols specified as common in the following way: if it encounters a definition for a symbol referenced as common, then it uses this value (the only time a definition should arise is in the FORTRAN Block Data case). Otherwise, it allocates suitable bss space using the maximum size you specified for each common symbol reference.

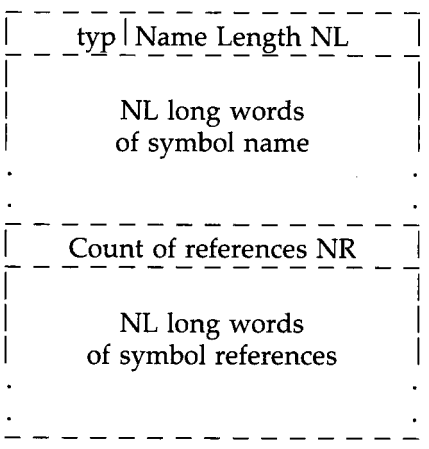
The linker handles external references differently according to the type of the corresponding definition. It adds absolute values to the long word, or byte field and gives an error if the signed value does not fit. Relocatable 32-bit references have the symbol value added to the field and a relocation record is produced for the loader. 16- and 8-bit references are handled as PC relative references and may only be made to hunks with the same name so that the hunks are coagulated by the linker before they are loaded. It is also possible for PC relative references to fail if the reference and the definition are too far apart. The linker may only access resident library definitions with 32-bit references, which it then handles as relocatable 32-bit references. The symbol data unit formats are as follows:

ext_def/abs/res

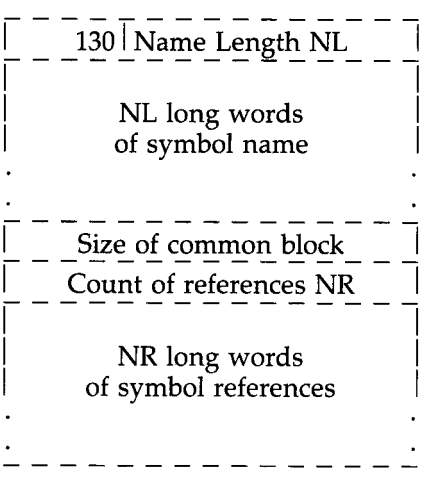
Figure 2-H: Symbol Data Unit



ext_ref32/16/8



ext_common



2.2.10 *hunk_symbol (1008/3F0)*

You use this block to attach a symbol table to a hunk so that you can use a symbolic debugger on the code. The linker passes symbol table blocks through attached to the hunk and, if the hunks are coagulated, coagulates the symbol tables. The loader does not load symbol table blocks into memory; when this is required, the debugger is expected to read the load file. The format of the symbol table block is the same as the external symbol information block with symbol table units for each name you use. The type code of zero is used within the symbol data units. The value of the symbol is the offset of the symbol from the start of the hunk. Thus the format is as follows:

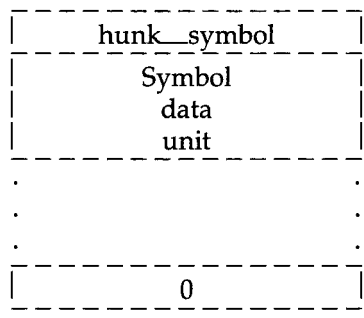


Figure 2-I: hunk_symbol (1008/3F0)

where each symbol data unit has the following format:

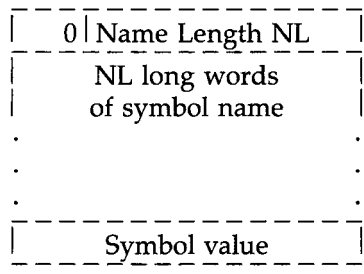


Figure 2-J: Symbol Data Unit

2.2.11 hunk_debug (1009/3F1)

AmigaDOS provides the debug block so that an object file can carry further debugging information. For example, high level language compilers may need to maintain descriptions of data structures for use by high level debuggers. The debug block may hold this information. AmigaDOS does not impose a format on the debug block except that it must start with the hunk_debug long word and be followed by a long word that indicates the size of the block in long words. Thus the format is as follows:

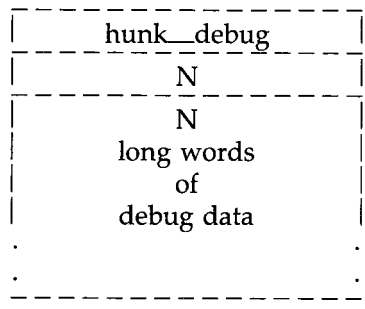


Figure 2-K: hunk_debug (1009/3F1)

2.2.12 hunk_end (1010/3F2)

This specifies the end of a hunk. It consists of a single long word, hunk_end.

2.3 Load Files

The format of a load file (that is, the output from the linker) is similar to that of an object file. In particular, it consists of a number of hunks with a similar format to those in an object file. The main difference is that the hunks never contain an external symbol information block, as all external symbols have been resolved, and the program unit information is not included. In a simple load file that is not overlaid, the file contains a header block which indicates the total number of hunks in the load file and any resident libraries the program referenced. This block is followed by the hunks, which may be the result of coagulating a number of input hunks if they had the same name. This complete structure is referred to as a node. Load files may also contain overlay information. In this case, an overlay table follows the primary node, and a special break block separates the overlay nodes. Thus the load file structure can be summarized as follows, where the items marked with an asterisk (*) are optional.

- Primary node
- Overlay table block (*)
- Overlay nodes separated by break blocks (*)

The relocation blocks within the hunks are always of type hunk_reloc32, and indicate the relocation to be performed at load time. This includes both the 32-bit relocation specified with hunk_reloc32 blocks in the object file and extra relocation required for the resolution of external symbols.

Each external reference in the object files is handled as follows. The linker searches the primary input for a matching external definition. If it does not find one, it searches the scanned library and includes in the load file the entire program unit where the definition was defined. This may make further external references become outstanding. At the end of the first pass, the linker knows all the external definitions and the total number of hunks that it is going to use. These include the hunks within the load file and the hunks associated with the resident libraries. On the second pass, the linker patches the long word external references so that they refer to the required offset within the hunk which defines the symbol. It produces an extra entry in the relocation block so that, when the hunks are loaded, it adds to each external reference the base address of the hunk defining the symbol. This mechanism also works for resident libraries.

Before the loader can make these cross-hunk references, it needs to know the number and size of the hunks in the nodes. The header block provides this information, as described below. The load file may also contain overlay information in an overlay table block. Break blocks separate the overlay nodes.

2.3.1 hunk_header (1011/3F3)

This block gives information about the number of hunks that are to be loaded, and the size of each one. It also contains the names of any resident libraries which must be opened when the node is loaded.

The format of the hunk_header is described in Figure 2-L. The first part of the header block contains the names of resident libraries that the loader must open when this node is loaded. Each name consists of a long word indicating the length of the name in long words and the text name padded to a long word boundary with zeros. The name list ends with a long word of zero. The names are in the order in which the loader is to open them.

When it loads a primary node, the loader allocates a table in memory which it uses to keep track of all the hunks it has loaded. This table must be large enough for all the hunks in the load file, including the hunks in overlays. The loader also uses this table to keep a copy of the hunk tables associated with any resident libraries. The next long word in the header block is therefore this table size, which is equal to the maximum hunk number referenced plus one.

The next long word F refers to the first slot in the hunk table the loader should use when loading. For a primary node that does not reference a resident library, this value is zero; otherwise, it is the number of hunks in the resident libraries. The loader copies these entries from the hunk table associated with the library following a library open call. For an overlay node, this value is the number of hunks in any resident libraries plus the number of hunks already loaded in ancestor nodes.

The next long word L refers to the last hunk slot the loader is to load as part of this loader call. The total number of hunks loaded is therefore L - F + 1.

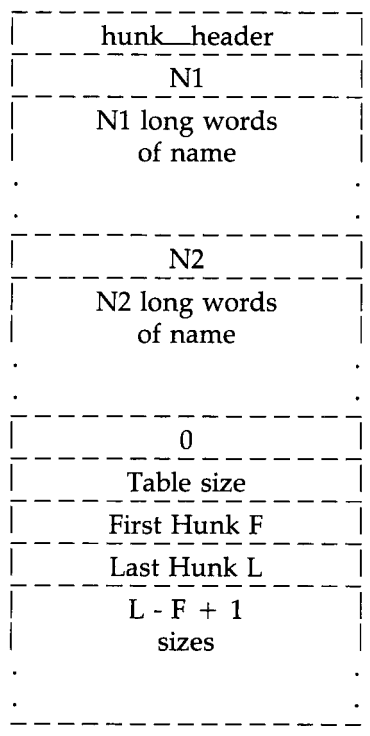


Figure 2-L: hunk_header (1011/3F3)

The header block continues with L - F + 1 long words which indicate the size of each hunk which is to be loaded as part of this call. This enables the loader to preallocate the space for the hunks and hence perform the relocation between hunks which is required as they are loaded. One hunk may be the bss hunk with a size given as zero; in this case the loader uses an operating system variable to give the size as described in hunk_bss on page 248.

2.3.2 *hunk_overlay (1013/3F5)*

The overlay table block indicates to the loader that it is loading an overlaid program, and contains all the data for the overlay table. On encountering it, the loader sets up the table, and returns, leaving the input channel to the load file still open. Its format is as follows:

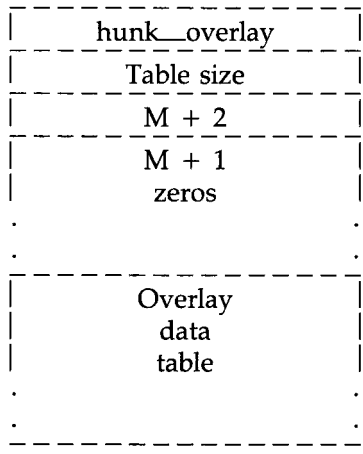


Figure 2-M: hunk_overlay (1013/3F5)

The first long word is the upper bound of the complete overlay table (in long words).

M is the maximum level of the overlay tree uses with the root level being zero. The next M + 1 words form the ordinate table section of the overlay table.

The rest of the block is the overlay data table, a series of eight-word entries, one for each overlay symbol. If 0 is the maximum overlay number used, then the size of the overlay data table is (0 + 1)*8, since the first overlay number is zero. So, the overlay table size is equal to (0 + 1)*8 + M + 1.

2.3.3 *hunk_break (1014/3F6)*

A break block indicates the end of an overlay node. It consists of a single long word, hunk_break.

2.4 *Examples*

The following simple sections of code show how the linker and loader handle external symbols. For example,

```
                                IDNT    A
                                XREF    BILLY, JOHN
                                XDEF    MARY
* The next long word requires relocation
0000' 0000 0008                 DC.L    FRED
0004' 123C 00FF                 MOVE.B  #$FF,D1
0008' 7001              FRED    MOVEQ   #1,D0
*External entry point
000A' 4E71              MARY    NOP
000C' 4EB9 0000 0000            JSR     BILLY     Call external
0012' 2239 0000 0000            MOVE.L  JOHN,D1 Reference external
                                END
```

produces the following object file:

```
hunk_unit
00000001    Size in long words
41000000    Name, padded to long word
hunk_code
00000006    Size in long words
00000008 123C00FF 70014E71 4EB90000 00002239 00000000
hunk-reloc32
00000001    Number in hunk 0
00000000    hunk 0
00000000    Offset to be relocated
00000000    Zero to mark end
hunk_ext
01000001    XDEF, Size 1 long word
4D415259    MARY
0000000A    Offset of definition
81000001    XREF, Size 1 long word
4A4F484E    JOHN
00000001    Number of references
00000014    Offset of reference
81000002    XREF, Size 2 long words
42494C4C    BILLY
59000000    (zeros to pad)
00000001    Number of references
0000000E    Offset of reference
00000000    End of external block
hunk_end
```

The matching program to this is as follows:

```
                                    IDNT    B
                                    XDEF    BILLY,JOHN
                                    XREF    MARY
0000' 2A3C AAAA AAAA                MOVE.L  #$AAAAAAAA,D5
* External entry point
0006' 4E71                  BILLY   NOP
* External entry point
0008' 7201                  JOHN    MOVEQ   #1,D1
*Call external reference
000A' 4EF9 0000 0000                JMP     MARY
                                    END
```

and the corresponding output code would be:

```
hunk_unit
00000001    Size in long words
42000000    Unit name
hunk_code
00000004    Size in long words
2A3CAAAA AAAA4E71 72014EF9 00000000
hunk_ext
01000001    XDEF, Size 1 long word
4A4F484E    JOHN
00000008    Offset of definition
01000002    XDEF, Size 2 long words
42494C4C    BILLY
59000000    (zeros to pad)
00000006    Offset of definition
81000001    XREF, Size 1 long word
4D415259    MARY
00000001    Number of references
0000000C    Offset of reference
00000000    End of external block
hunk_end
```

Once you passed this through the linker, the load file would have the following format:

```
hunk_header
00000000    No hunk name
00000002    Size of hunk table
00000000    First hunk
00000001    Last hunk
```

```
00000006      Size of hunk 0
00000004      Size of hunk 1
hunk_code
00000006  Size of code in long words
00000008 123C00FF 70014E71 4EB90000 00062239 00000008
hunk_reloc32
00000001      Number in hunk 0
00000000      hunk 0
00000000      Offset to be relocated
00000002      Number in hunk 1
00000001      hunk 1
00000014      Offset to be relocated
0000000E      Offset to be relocated
00000000      Zero to mark end
hunk_end
hunk_code
00000004      Size of code in long words
2A3CAAAA AAAA4E71 72014EF9 0000000A
hunk_reloc32
00000001      Number in hunk 0
00000000      hunk 0
0000000C      Offset to be relocated
00000000      Zero to mark end
hunk_end
```

When the loader loads this code into memory, it reads the header block and allocates a hunk table of two long words. It then allocates space by calling an operating system routine and requesting two areas of sizes 6 and 4 long words respectively. Assuming the two areas it returned were at locations 3000 and 7000, the hunk table would contain 3000 and 7000.

The loader reads the first hunk and places the code at 3000; it then handles relocation. The first item specifies relocation with respect to hunk 0, so it adds 3000 to the long word at offset 0 converting the value stored there from 00000008 to 00003008. The second item specifies relocation with respect to hunk 1. Although this is not loaded, we know that it will be loaded at location 7000, so this is added to the values stored at 300E and 3014. Note that the linker has already inserted the offsets 00000006 and 00000008 into the references in hunk 0 so that they refer to the correct offset in hunk 1 for the definition. Thus the long words specifying the external references end up containing the values 00007006 and 00007008, which is the correct place once the second hunk is loaded.

In the same way, the loader loads the second hunk into memory at location

7000 and the relocation information specified alters the longword at 700C from 0000000A (the offset of MARY in the first hunk) to 0000300A (the address of MARY in memory).

The loader handles references to resident libraries in the same way, except that, after it has opened the library, it copies the locations of the hunks comprising the library into the start of the hunk table. It then patches references to the resident library to refer to the correct place by adding the base of the library hunks.

Chapter 3

AmigaDOS Data Structures

This chapter describes AmigaDOS data structures in memory and in files. It does not describe the layout of a disk, which is described in Chapter 1.

3.1 Introduction

AmigDOS provides device independent input and output. It achieves this by creating a handler process for each device you use. The handler process accepts a standard set of I/O requests and converts these to device specific requests where required. All AmigaDOS clients refer to the handler process rather than the device directly, although it is possible to use a device without the handler if this is required. This chapter describes the data structures within AmigaDOS, including the format of a process, central shared data structures, and the structure of handler requests.

In addition to normal Amiga values such as LONG and APTR, AmigaDOS uses BPTR. BPTR is a BCPL pointer, which is a pointer to a long word-aligned memory block divided by 4. So, to read a BPTR in C, you simply shift left by 2. To create a BPTR, you must either use memory obtained via a call to AllocMem or a structure on your stack when you know you have only allocated long words

on the stack so far (the initial stack is long word aligned). You should then shift this pointer right by 2 to create the BPTR.

AmigaDOS also has a BSTR, which is a BCPL string. BSTR consists of a BPTR to memory that contains the length of the string in the first byte, and the bytes within the string following.

A number of references to the Global Vector appear within this chapter. The Global Vector is a jump table used by BCPL and is a pointer to a standard shared Global Vector. Some processes, such as the file handler, use a private global vector.

The include files dos.h and dosextens.h contain C language definitions for the following structures. The .i files for assembly language.

3.2 Process Data Structures

These values are created as part of an AmigaDOS process; there is a complete set for each process.

A process is an Exec task with a number of extra data structures appended. The process structure consists of:

Exec task structure
Exec message port
AmigaDOS process values

The process identifier AmigaDOS uses internally is a pointer to the Exec message port (from which the Exec task may be obtained).

AmigaDOS process values are as follows:

Value	Function	Description
BPTR	SegArray	Array of SegLists used by this process
LONG	StackSize	Size of process stack in bytes
APTR	GlobVec	Global Vector for this process
LONG	TaskNum	CLI Task number or zero if not a CLI
BPTR	StackBase	Pointer to high memory end of process stack
LONG	IoErr	Value of secondary result from last call
BPTR	CurrentDir	Lock associated with current directory
BPTR	CIS	Current CLI input stream
BPTR	COS	Current CLI output stream
APTR	CoHand	Console handler process for current window
APTR	FiHand	File handler process for current drive
BPTR	CLIStruct	Pointer to additional CLI information
APTR	ReturnAddr	Pointer to previous stackframe
APTR	PktWait	Function to be called when awaiting message
APTR	WindowPtr	Pointer to window

To identify the segments that a particular process uses, you use SegArray. SegArray is an array of long words with its size in SegArray[0]. Other elements are either zero or a BPTR to a SegList. CreateProc creates this array with the first two elements of the array pointing to resident code and the third element being the SegList passed as argument. When a process terminates, FreeMem is used to return the space for the SegArray.

StackSize indicates the size of the process stack, as supplied by the user when calling CreateProc. Note that the process stack is not the same as the command stack a CLI uses when it calls a program. The CLI obtains its command stack just before it runs a program and you may alter the size of this stack with the STACK command. When you create a process, AmigaDOS obtains the process stack and stores the size in StackSize. The pointer to the space for the process control block and the stack is also stored in the MemEntry field of the task structure. When the process terminates this space is returned via a call to FreeMem. You can also chain any memory you obtain into this list structure so that it, too, gets put back when the task terminates.

If a call to CreateProc creates the process, GlobVec is a pointer to the Shared Global Vector. However, some internal handler processes use a private GlobVec.

The value of TaskNum is normally zero; a CLI process stores the small integer that identifies the invocation of the CLI here.

The pointer StackBase points to the high-memory end of the process stack. This is the end of the stack when using languages such as C or Assembler; it is the base of the stack for languages such as BCPL.

The values of IoErr and CurrentDir are those handled by the similarly named AmigaDOS calls. CIS and COS are the values Input and Output return and refer to the file handles you should use when running a program under the CLI. In other cases CIS and COS are zero.

CoHand and FiHand refer to the console handler for the current window and the file handler for the current device. You use these values when attempting to open the * device or a file by a relative path name.

The CLIStruct pointer is nonzero only for CLI processes. In this case it refers to a further structure the CLI uses with the following format:

Value	Function	Description
LONG	Result2	Value of IoErr from last command
BSTR	SetName	Name of current directory
BPTR	CommandDir	Lock associated with command directory
LONG	ReturnCode	Return code from last command
BSTR	CommandName	Name of current command
LONG	FailLevel	Fail level (set by FAILAT)
BSTR	Prompt	Current prompt (set by PROMPT)
BPTR	StandardIn	Default (terminal) CLI input
BPTR	CurrentIn	Current CLI input

Value	Function	Description
BSTR	CommandFile	Name of EXECUTE command file
LONG	Interactive	Boolean; True if prompts required
LONG	Background	Boolean; True if CLI created by RUN
BPTR	CurrentOut	Current CLI output
LONG	DefaultStack	Stack size to be obtained (in long words)
BPTR	StandardOut	Default (terminal) CLI output
BPTR	Module	SegList of currently loaded command

The Exit function uses the value of ReturnAddr which points to just above the return address on the currently active stack. If a program exits by performing an RTS on an empty stack, then control passes to the code address pushed onto the stack by CreateProc or by the CLI. If a program terminates with a call to Exit, then AmigaDOS uses this pointer to extract the same return address.

The value of PktWait is normally zero. If it is nonzero, then AmigaDOS calls PktWait whenever a process is about to go to sleep to await a signal indicating that a message has arrived. In the same way as GetMsg, the function should return a message when one is available. Usually, you use this function to filter out any private messages arriving at the standard process message port that are not intended for AmigaDOS.

The value of WindowPtr is used when AmigaDOS detects an error that normally requires the user to take some action. Examples of these errors are attempting to write to a write-protected disk, or when the disk is full. If the value of WindowPtr is -1, then the error is returned to the calling program as an error code from the AmigaDOS call of Open, Write, or whatever. If the value is zero, then AmigaDOS places a request box on the Workbench screen informing the user of the error and providing the opportunity to retry the operation or to cancel it. If the user selects cancel, then AmigaDOS returns the error code to the calling program. If the user selects retry, or inserts a disk, then AmigaDOS attempts the operation once more.

If you place a positive value into the WindowPtr field, then AmigaDOS takes this to be a pointer to a Window structure. Normally you would place the Window structure of the window you are currently using here. In this case, AmigaDOS displays the error message within the window you have specified, rather than using the Workbench screen. You can always leave the WindowPtr field as zero, but if you are using another screen, then the messages AmigaDOS displays appear on the Workbench screen, possibly obscured by your own screen.

The initial value of WindowPtr is inherited from the process that created the current one. If you decide to alter WindowPtr from within a program that runs under the CLI, then you should save the original value and restore it when you finish; otherwise, the CLI process contains a WindowPtr that refers to a window that is no longer present.

3.3 Global Data Structure

This data structure only exists once; however, all AmigaDOS processes use it. If you make a call to OpenLibrary, you can obtain the library base pointer. The base of the data structure is a positive offset from the library base pointer. The library base pointer points to the following structure:

Library Node structure
APTR to DOS RootNode
APTR to DOS Shared Global Vector
DOS private register dump

All internal AmigaDOS calls use the Shared Global Vector, which is a jump table. You should not normally use it, except through the supplied interface calls, as it is liable to change without warning.

The RootNode structure is as follows:

Value	Function	Description
BPTR	TaskTable	Array of CLI processes currently running
BPTR	CLISegList	SegList for the CLI
LONG	Days	Number of days in current time
LONG	Mins	Number of minutes in current time
LONG	Ticks	Number of ticks in current time
BPTR	RestartSeg	SegList for the disk validator process
BPTR	Info	Pointer to the Info substructure

The TaskTable is an array with the size of the array stored in TaskTable[0]. The processid (in other words, the MsgPort associated with the process) for each CLI is stored in the array. The process id for the CLI with TaskNum "n" is stored in TaskTable[n]. An empty slot is filled with a zero. The commands RUN and NEWCLI scan the TaskTable to identify the next free slot, and use this as the TaskNum for the CLI created.

The CLISegList is the SegList for the code of the CLI. RUN and NEWCLI use this value to create a new instance of a CLI.

The rootnode stores the current date and time; normally you should use the AmigaDOS function DateStamp to return a consistent set of values. The values Days, Mins, and Ticks specify the date and time. The value of Days is the number of days since January 1st, 1978. The value of Mins is the number of minutes since midnight. A tick is one fiftieth of a second, but the time is only updated once per second.

The RestartSeg is the SegList for the code of the disk validator, which is a process that AmigaDOS creates whenever you insert a new disk into a drive.

3.3.1 Info Substructure

To access a further substructure with the following format, you use the Info pointer.

Value	Function	Description
BPTR	McName	Network name of this machine; currently zero
BPTR	DevInfo	Device list
BPTR	Devices	Currently zero
BPTR	Handlers	Currently zero
APTR	NetHand	Network handler process id, currently zero

Most of the fields in the Info substructure are empty at the moment, but Commodore-Amiga intend to use them for expanding the system.

The DevInfo structure is a linked list. You use it to identify all the device names that AmigaDOS knows about; this includes ASSIGNed names and disk volume names. There are two possible formats for the list entries depending on whether the entry refers to a disk volume or not. For an entry describing a device or a directory (via ASSIGN) the entry is as follows:

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
LONG	Type	List entry type (device or dir)
APTR	Task	Handler process or zero
BPTR	Lock	File system lock or zero
BSTR	Handler	File name of handler or zero
LONG	StackSize	Stack size for handler process
LONG	Priority	Priority for handler process
LONG	Startup	Startup value to be passed to handler process
BPTR	SegList	SegList for handler process or zero
BPTR	GlobVec	Global Vector for handler process or zero
BSTR	Name	Name of device or ASSIGNed name

The Next field links all the list entries together, and the name of the logical device name is held in the Name field.

The Type field is 0 (dt_device) or 1 (dt_dir). You can make a directory entry with the ASSIGN command. This command allocates a name to a directory that you can then use as a device name. If the list entry refers to a directory, then the TASK refers to the file system process handling that disk, and the Lock field contains a pointer to a lock on that directory.

If the list entry refers to a device, then the device may or may not be resident. If it is resident, the Task identifies the handler process, and the Lock

is normally zero. If the device is not resident, then the TASK is zero and AmigaDOS uses the rest of the list structure.

If the SegList is zero, then the code for the device is not in memory. The Handler field is a string specifying the file containing the code (for example, SYS:L/RAM-HANDLER). A call to LoadSeg loads the code from the file and inserts the result into the SegList field.

AMigaDOS now creates a new handler process with the SegList, StackSize, and Pri values. The new process is a BCPL process and requires a Global Vector; this is either the value you specified in GlobVec or a new private global vector if GlobVec is zero.

The new process is passed a message containing the name originally specified, the value stored in Startup and the base of the list entry. The new handler process may then decide to patch into the Task slot the process id or not as required. If the Task slot is patched, then subsequent references to the device name use the same handler task; this is what the RAM: device does. If the Task slot is not patched, then further references to the device result in new process invocations; this is what the CON: device does.

If the Type field within the list entry is equal to 2 (dt_volume), then the format of the list structure is slightly different.

Value	Function	Description
BPTR	Next	Pointer to next list entry or zero
LONG	Type	List entry type (volume)
APTR	Task	Handler process or zero
BPTR	Lock	File system lock
LONG	VolDays	Volume creation date
LONG	VolMins	
LONG	VolTicks	
BPTR	LockList	List of active locks for this volume
LONG	DiskType	Type of disk
LONG	Spare	Not used
BSTR	Name	Volume name

In this case, the Name field is the name of the volume, and the Task field refers to the handler process if the volume is currently inserted; or to zero if the volume is not inserted. To distinguish disks with the same name, AmigaDOS timestamps the volume on creation and then saves the timestamp in the list structure. AmigDOS can therefore compare the timestamps of different volumes whenever necessary.

If a volume is not currently inserted, then AmigaDOS saves the list of currently active locks in the LockList field. It uses the DiskType field to identify

the type of disk. Currently, this is always an AmigaDOS disk. The disk type is up to four characters packed into a long word and padded on the right with nulls.

3.4 Memory Allocation

AmigaDOS obtains all the memory it allocates by calling the AllocMem function provided by Exec. In this way, AmigaDOS obtains structures such as locks and file handles; it usually places them back in the free pool by calling FreeMem. Each memory segment allocated by AmigaDOS is identified by a BPTR to the second long word in the structure. The first long word always contains the length of the entire segment in bytes. Thus the structure of allocated memory is as follows:

Value	Function	Description
LONG	BlockSize	Size of memory block
LONG	FirstData	First data segment, BPTR to block points here

3.5 Segment Lists

To obtain a segment list, you call LoadSeg. The result is a BPTR to allocated memory, so that the length of the memory block containing each list entry is stored at -4 from the BPTR. This length is 8 more than the size of the segment list entry, allowing for the link field and the size field itself.

The SegList is a list linked together by BPTRs and terminated by zero. The remainder of each segment list entry contains the code loaded. Thus the format is

Value	Function	Description
LONG	NextSeg	BPTR to next segment or zero
LONG	FirstCode	First value from binary file

3.6 File Handles

File handles are created by the AmigaDOS function Open, and you use them as arguments to other functions such as Read and Write. AmigaDOS returns them as a BPTR to the following structure:

Value	Function	Description
LONG	Link	Not used
LONG	Interact	Boolean, TRUE if interactive
LONG	ProcessID	Process id of handler process
BPTR	Buffer	Buffer for internal use
LONG	CharPos	Character position for internal use
LONG	BufEnd	End position for internal use
APTR	ReadFunc	Function called when buffer exhausted
APTR	WriteFunc	Function called when buffer full
APTR	CloseFunc	Function called when handle closed
LONG	Arg1	Argument; depends on file handle type
LONG	Arg2	Argument; depends on file handle type

Most of the fields are only used by AmigaDOS internally; normally Read or Write uses the file handle to indicate the handler process and any arguments to be passed. Values should not be altered within the file handle by user programs, except that the first field may be used to link file handles into a singly linked list.

This description does NOT match dosextens.h or .i. Use the include file information instead.

3.7 Locks

The filing system extensively uses a data structure called a lock. This structure serves two purposes. First, it serves as the mechanism to open files for multiple reads or a single write. Note that obtaining a shared read lock on a directory does not stop that directory being updated.

Second, the lock provides a unique identification for a file. Although a particular file may be specified in many ways, the lock is a simple handle on that file. The lock contains the actual disk block location of the directory or file header and is thus a shorthand way of specifying a particular file system object. The structure of a lock is as follows:

Value	Function	Description
BPTR	NextLock	BPTR to next in chain, else zero
LONG	DiskBlock	Block number of directory or file header
LONG	AccessType	Shared or exclusive access
APTR	ProcessID	Process id of handler task
BPTR	VolNode	Volume entry for this lock

Because AmigaDOS uses the NextLock field to chain locks together, you should not alter it. The filing system fills in DiskBlock field to represent the

location on disk of the directory block or the file header block. The AccessType serves to indicate whether this is a shared read lock, when it has the value -2, or an exclusive write lock when it has the value -1. The Process ID field contains a pointer to the handler process for the device containing the file to which this lock refers. Finally the VolNode field points to the node in the DevInfo structure that identifies the volume to which this lock refers. Volume entries in the DevInfo structure remain there if a disk is inserted or if there are any locks open on that volume.

Note that a lock can also be a zero. The special case of lock zero indicates that the lock refers to the root of the initial filing system, and the FiHand field within the process data structure gives the handler process.

3.8 Packets

Packet passing handles all communication performed by AmigaDOS between processes. A packet is a structure built on top of the message-passing mechanism provided by the Exec kernel.

An Exec message is a structure, described elsewhere, that includes a Name field. AmigaDOS uses the field as an APTR to another section of memory called a packet. A packet must be long word aligned, and has the following general structure.

Value	Function	Description
APTR	MsgPtr	Pointer back to message structure
APTR	MsgPort	Message port where the reply should be sent
LONG	PktType	Packet type
LONG	Res1	First result field
LONG	Res2	Second result field
LONG	Arg1	Argument; depends on packet type
LONG	Arg2	Argument; depends on packet type
	. . .	
LONG	ArgN	Argument; depends on packet type

The format of a specific packet depends on its type; but in all cases, it contains a back pointer to the Message structure, the MgsPort for the reply, and two result fields. When AmigaDOS sends a packet, the reply port is overwritten with the process identifier of the sender so that the packet can be returned. Thus, when sending a packet to an AmigaDOS handler process, you must fill in the reply MsgPort each time; otherwise, when the packet returns, AmigaDOS has overwritten the original port. AmigaDOS maintains all other fields except the result fields.

All AmigaDOS packets are sent to the message port created as part of a

process; this message port is initialized so that arriving messages cause signal 8 to be set. An AmigaDOS process which is waiting for a message waits for signal 8 to be set. When the process wakes up because this event has occurred, GetMsg takes the message from the message port and extracts the packet address. If the process is an AmigaDOS handler process, then the packet contains a value in the PktType field which indicates an action to be performed, such as reading some data. The argument fields contain specific information such as the address and size of the buffer where the characters go.

When the handler process has completed the work required to satisfy this request, the packet returns to the sender, using the same message structure. Both the message structure and the packet structure must be allocated by the client and must not be deallocated before the reply has been received. Normally AmigaDOS is called by the client to send the packet, such as when a call to Read is made. However, there are cases when asynchronous IO is required, and in this case the client may send packets to the handler processes as required. The packet and message structures must be allocated, and the processid field filled in with the message port where this packet must return. A call to PutMsg then sends the message to the destination. Note that many packets may be sent out returning to either the same or different message ports.

3.8.1 Packet Types

AmigaDOS supports the following packet types. Not all types are valid to all handlers, for example a rename request is only valid to handlers supporting a filing system. For each packet type the arguments and results are described. The actual decimal code for each type appears next to the symbolic name. In all cases, the Res2 field contains additional information concerning an error (indicated by a zero value for Res1 in most cases). To obtain this additional information, you can call IoErr when making a standard AmigaDOS call.

Open Old File

```
Type    LONG    Action.FindInput (1005)

Arg1    BPTR    FileHandle
Arg2    BPTR    Lock
Arg3    BSTR    Name

Res1    LONG    Boolean
```

Attempts to open an existing file for input or output (see the function Open in Chapter 2, "Calling AmigaDOS," of the *AmigaDOS Developer's Manual* in this book for further details on opening files for I/O). To obtain the value of lock,

you call DeviceProc to obtain the handler ProcessId and then IoErr which returns the lock. Alternatively the lock and ProcessId can be obtained directly from the DevInfo structure. Note that the lock refers to the directory owning the file, not to the file itself.

The caller must allocate and initialize FileHandle. This is done by clearing all fields to zero except for the CharPos and BufEnd fields which should be set to -1. The ProcessID field within the FileHandle must be set to the process id of the handler process.

The result is zero if the call failed, in which case the Res2 field provides more information on the failure and the FileHandle should be released.

Open New File

Type	LONG	Action. FindOutput (1006)
Arg1	BPTR	FileHandle
Arg2	BPTR	Lock
Arg3	BSTR	Name
Res1	LONG	Boolean

Arguments as for previous entry.

Read

Type	LONG	Action.Read (82)
Arg1	BPTR	FileHandle Arg1
Arg2	APTR	Buffer
Arg3	LONG	Length
Res1	LONG	Actual Length

To read from a file handle, the process id is extracted from the ProcessID field of the file handle, and the Arg1 field from the handle is placed in the Arg1 field of the packet. The buffer address and length are then placed in the other two argument fields. The result indicates the number of characters read—see the function Read for more details. An error is indicated by returning -1 whereupon the Res2 field contains more information.

Write

Type	LONG	Action.Write (87)
Arg1	BPTR	FileHandle Arg1
Arg2	APTR	Buffer
Arg3	LONG	Length
Res1	LONG	Actual Length

The arguments are the same as those for Read. See the Write function for details of the result field.

Close

Type	LONG	Action.End (1007)
Arg1	BPTR	FileHandle Arg1
Res1	LONG	TRUE

You use this packet type to close an open file handle. The process id of the handler is obtained from the file handle. The function normally returns TRUE. After a file handle has been closed, the space associated with it should be returned to the free pool.

Seek

Type	LONG	Action.Seek (1008)
Arg1	BPTR	FileHandle Arg1
Arg2	LONG	Position
Arg3	LONG	Mode
Res1	LONG	OldPosition

This packet type corresponds to the SEEK call. It returns to the old position, or -1 if an error occurs. The process id is obtained from the file handle.

WaitChar

Type	LONG	Action.WaitChar (20)
Arg1	LONG	Timeout
Res1	LONG	Boolean

This packet type implements the WaitForChar function. You must send the packet to a console handler process, with the timeout required in Arg1. The packet returns when either a character is waiting to be read, or when the timeout expires. If the result is TRUE, then at least one character may be obtained by a subsequent READ.

ExamineObject

Type	LONG	Action.ExamineObject (23)
Arg1	BPTR	Lock
Arg2	BPTR	FileInfoBlock
Res1	LONG	Boolean

This packet type implements the Examine function. It extracts the process id of the handler from the ProcessID field of the lock. If the lock is zero, then it uses the default file handler, which is kept in the FiHand field of the process. The result is zero if it fails, with more information in Res2. The FileInfoBlock returns with the name and comment fields as BSTRs.

ExamineNext

Type	LONG	Action.ExamineNext (24)
Arg1	BPTR	Lock
Arg2	BPTR	FileInfoBlock
Res1	LONG	Boolean

This call implements the ExNext function, and the arguments are similar to those for Examine above. Note that the BSTR representing the filename must not be disturbed between calls of ExamineObject and different calls to ExamineNext, as it uses the name as a place saver within the directory being examined.

DiskInfo

Type	LONG	Action.DiskInfo (25)
Arg1	BPTR	InfoData
Res1	LONG	TRUE

This implements the Info function. A suitable lock on the device would normally obtain the process id for the handler. This packet can also be sent to a console handler process, in which case the Volume field in the InfoData contains the window pointer for the window opened on your behalf by the console handler.

Parent

Type	LONG	Action.Parent (29)
Arg1	BPTR	Lock
Res1	LONG	ParentLock

This packet returns a lock representing the parent of the specified lock, as provided by the ParentDir function call. Again it must obtain the process id of the handler from the lock, or from the FiHand field of the current process if the lock is zero.

DeleteObject

Type	LONG	Action.DeleteObject (16)
Arg1	BPTR	Lock
Arg2	BSTR	Name
Res1	LONG	Boolean

This packet type implements the Delete function. It must obtain the lock from a call to IoErr() immediately following a successful call to DeviceProc which returns the process id. The lock actually refers to the directory owning the object to be deleted, as in the Open New and Open Old requests.

CreateDir

Type	LONG	Action.CreateDir (22)
Arg1	BPTR	Lock
Arg2	BSTR	Name
Res1	BPTR	Lock

This packet type implements the CreateDir function. Arguments are the same as for DeleteObject. The result is zero or a lock representing the new directory.

LocateObject

Type	LONG	Action.LocateObject (8)
Arg1	BPTR	Lock
Arg2	BSTR	Name
Arg3	LONG	Mode
Res1	BPTR	Lock

This implements the lock function and returns the lock or zero. Arguments as for CreateDir with the addition of the Mode as Arg3. Mode refers to the type of lock, shared or exclusive.

CopyDir

Type	LONG	Action.CopyDir (19)
Arg1	BPTR	Lock
Res1	BPTR	Lock

This implements the DupLock function. If the lock requiring duplication is zero, then the duplicate is zero. Otherwise, the process id is extracted from the lock and this packet type sent. The result is the new lock or zero if an error was detected.

FreeLock

Type	LONG	Action.FreeLock (15)
Arg1	BPTR	Lock
Res1	LONG	Boolean

This call implements the UnLock function. It obtains the process id from the lock. Note that freeing the zero lock takes no action.

SetProtect

Type	LONG	Action.SetProtect (21)
Arg1	Not used	
Arg2	BPTR	Lock
Arg3	BSTR	Name
Arg4	LONG	Mask
Res1	LONG	Boolean

This implements the SetProtection function. The lock is a lock on the owning directory obtained from DeviceProc as described for DeleteObject above. The least significant four bits of "Mask" represent Read, Write, Execute, and Delete in that order. Delete is bit zero.

SetComment

Type	LONG	Action.SetComment (28)
Arg1	Not used	
Arg2	BPTR	Lock
Arg3	BSTR	Name
Arg4	BSTR	Comment
Res1	LONG	Boolean

This implements the SetComment function. Arguments as for SetProtect above, except that Arg4 is a BSTR representing the comment.

RenameObject

Type	LONG	Action.RenameObject (17)
Arg1	BPTR	FromLock
Arg2	BPTR	FromName
Arg3	BPTR	ToLock
Arg4	BPTR	ToName
Res1	LONG	Boolean

This implements the Rename function. It must contain an owning directory lock and a name for both the source and the destination. The owning directories are obtained from DeviceProc as mentioned under the entry for the DeleteObject.

Inhibit

Type	LONG	Action.Inhibit (31)
Arg1	LONG	Boolean
Res1	LONG	Boolean

This packet type implements a filing system operation that is not available as an AmigaDOS call. The packet contains a Boolean value indicating whether the filing system is to be stopped from attempting to verify any new disks placed into the drive handled by that handler process. If the Boolean is true, then you may swap disks without the filesystem process attempting to verify the disk. While disk change events are inhibited, the disk type is marked as "Not a DOS disk" so that other processes are prevented from looking at the disk.

If the Boolean is false, then the file system reverts to normal after having verified the current disk in the drive.

This request is useful if you wish to write a program such as DISKCOPY where there is much swapping of disks that may have a half completed structure. If you use this packet request then you can avoid having error messages from the disk validator while it attempts to scan a half completed disk.

RenameDisk

Type	LONG	Action.RenameDisk (9)
Arg1	BPTR	NewName
Res1	BPTR	Boolean

Again, this implements an operation not normally available through a function call. The single argument indicates the new name required for the disk currently mounted in the drive handled by the filesystem process where the packet is sent. The volume name is altered both in memory and on the disk.

Chapter 4

AmigaDOS Additional Information for the Advanced Developer

This chapter describes certain topics which are likely to be of interest to the advanced developer who may wish to create new devices to be added to the Amiga or who wish their code to run with Amiga computers which have been expanded beyond a 512K memory size.

The following topics are covered here:

Overlay Hunk Description
: for developers putting together large programs

ATOM utility
: works on a new binary file format to change allow developer to set the appropriate load bits. Assures that program code and data that must be resident in CHIP memory (the lowest 512K of the system) for the program to function will indeed be placed there by AmigaDOS when it is loaded. Otherwise the program code may not work on an extended memory machine.

Linking in a new DISK-device to AmigaDOS
: lets a developer add a hard disk or disk-like device as a name-addressable part of the filing system.

Linking in a new non-disk-device to AmigaDOS
: lets a developer add such things as additional serial ports, parallel ports, graphics tablets, RAM-disks or what-have-you to AmigaDOS (non filing system related).

Using AmigaDOS without using Intuition
: for developers who may prefer to install and use their own screen handling in place of that provided by Intuition.

Hunk Overlay Table—Overview

When overlays are used, the linker basically produces one very large file containing all of the object modules as hunks of relocatable code. The hunk overlay table contains a data structure that describes the hunks and their relationship to each other.

When you are designing a program to use overlays, you must keep in mind how the overlay manager (also called the overlay supervisor) handles the interaction between the various segments of the file. What you must do, basically, is build a tree that reflects the relationships between the various code modules that are a part of the overall program and tell the linker how this tree should be constructed.

The hunk overlay table is generated as a set of 8 long words, each describing a particular overlay node that is part of the overall file. Each 8 long word entry is comprised of the following data:

HUNK OVERLAY SYMBOL TABLE-ENTRY DATA STRUCTURE:

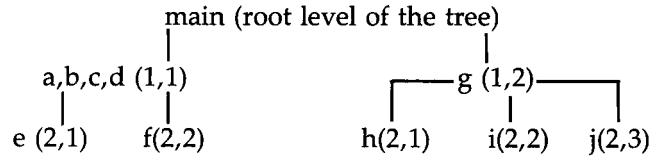
```
long seekOffset;     /* where in the file to find this node */
long dummy1;         /* a value of 0 . . . compatibility item */
long dummy2;         /* a value of 0 . . . compatibility item */
long level;          /* level in the tree */
long ordinate;       /* item number at that level */
long firstHunk;      /* hunk number of the first hunk containing
                      * this node. */
long symbolHunk;     /* the hunk number in which this symbol is
                      * located */
long symbolOffsetX;  /* (offset + 4), where offset is the offset
                      * within the symbol hunk at which this
                      * symbol's entry is located. */
```

Each of these items is explained further in the sections that follow.

Designing an Overlay Tree

Let's say that you have, for example, the files main, a, b, c, d, e, f, g, h, i, and j, and that main can call a,b,c, and d and that each of these files can call main. Additionally let's say that routine e can be called from a,b,c,d, or main, but has no relationship to routine f. Thus, if a routine in e is to be run, then a,b,c, and d need to be memory-resident as well. Routine f is like e; that is, it needs nothing in e to be present, but can be called from a, b, c, or d. This means that the overlay manager can share the memory space between routines e and f, since neither need ever be memory-coresident with the other in order to run.

If you consider routine g to share the same space as the combination of a,b,c, and d and routines h,i, and j sharing the same space, you have the basis for constructing the overlay tree for this program structure:



Not only have we drawn the tree, but we have labeled its branches to match the hunk overlay (level, ordinate) numbers that are found in the hunk overlay table that matches the nodes to which they are assigned.

From the description above, you can see that if main is to call any routine in program segment a–d, then all of those segments should be resident in memory at the same time. Thus they have all been assigned to a single node by the linker. While a–d are resident, if you call routines in e, the linker will automatically load routine e from disk, and reinitialize the module (each time it is again brought in), so that its subroutines will be available to be run. If any segment a–d calls a routine in f, the linker replaces e with the contents of f and initializes it. Thus a–d are at level 1 in the overlay tree, and routines e and f are at level 2, requiring that a–d be loaded before e or f can be accessed and loaded for execution.

Note: A routine can only perform calls to routines in other nodes which either are currently memory resident (the ancestors of the node in which the routine now in use is located), or a routine in a direct child node. That is, main cannot call e directly, but e can call routines in main since main is an ancestor.

Note also that within each branch of each subnode, the ordinate numbers begin again with number 1 for a given level.

Describing the Tree

You create the tree by telling the overlay linker about its structure. The numerical values, similar to those noted in the figure above, are assigned sequentially by the linker itself and appear in the hunk node table. Here is the sequence of overlay link statements that cause the figure above to be built:

```
OVERLAY
a,b,c,d
*e
*f
g
*h
*i
*j
```

This description tells the linker that a,b,c,d are part of a single node at a given level (in this case level 1), and the asterisk in front of e and f each say that these are one each on the next level down from a–d, and accessible only through a–d or anything closer toward the root of the tree. The name g has no asterisk, so it is considered on the same level as a–d, telling the linker that

either a–d or g will be memory-resident, but not both simultaneously. Names h,i, and j are shown to be related to g, one level down.

The above paragraphs have explained the origin of the hunk node level and the hunk ordinate in the hunk overlay symbol table.

Seek Offset Amount

The first value for each node in the overlay table is the seek offset. As specified earlier, the overlay linker builds a large single file containing all of the overlay nodes. The seek offset number is that value that can be given to the seek(file, byte_offset) routine to point to the first byte of the hunk header of a node.

initialHunk

The initialHunk value in the overlay symbol table is used by the overlay manager when unloading a node. It specifies the initial hunk that must have been loaded in order to have loaded the node that contains this symbol. When a routine is called at a different level and ordinate (unless it is a direct, next level, child of the current node), it will become necessary to free the memory utilized by invalid hunks, so as to make room to overlay with the hunk(s) containing the desired symbol.

SymbolHunk and SymbolOffsetX

These table entries for the symbols are used by the overlay manager to actually locate the entry point once it has either determined it is already loaded or has loaded it. The symbolHunk shows in which hunk to locate the symbol. SymbolOffsetX-4 shows the offset from the start of that hunk at which the entry point is actually located.

ATOM: (Alink Temporary Object Modifier)

This document describes the ATOM utility, including its development history, the manner in which it has been implemented, and alternatives to its use.

The "problem":

Programmers need/want to be able to specify that parts of their program go into "chip" memory (the first 512K) so that the custom chips can access it. They also need/want to treat this data just like any other data in their program and therefore have it link and load normally.

Previous Solutions

The recommended way of dealing with this was to do an AllocMem with the chip memory bit set and copy data from where it was loaded ("fast" memory) to where it belonged (chip memory), then use pointers to get to it. This involved having two copies of your data in memory, the first loaded with your program, the second copied into the first 512K of memory.

The other "solution" is to have the program not run in machines with more than 512K. This should quickly become an unacceptable solution.

The ATOM Solution

1. Compile or assemble normally.
2. Pass the object code through a post- (or pre-) processor called "ATOM". ATOM will interact with the user and the object file(s). It will flag the desired hunks (or all hunks) as "for chip memory" by changing the hunk type.
3. The linker will now take nine (9) hunk types instead of 3. The old types were hunk_code, hunk_data, and hunk_bss. The new ones will be:

   hunk_code_chip = hunk_code + bit 30 set
   hunk_code_fast = hunk_code + bit 31 set
   hunk_data_chip = hunk_data + bit 30 set
   hunk_data_fast = hunk_data + bit 31 set
   hunk_bss_chip = hunk_bss + bit 30 set
   hunk_bss_fast = hunk_bss + bit 31 set

   The linker will pass all hunk types through to the LOADER (coagulating if necessary). The LOADER uses the hunk header information when loading.

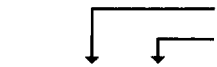
   You will recall from the information provided in the linker documentation that CODE hunks contain executable (68000) machine language, DATA hunks contain initialized data (constants, . . .) and BSS hunks contain unintialized data (arrays, variable declarations, . . .).
4. The LOADER will load according to information from step 3 above. Hunks will go into the designated memory type.
5. Old versions of the LOADER will interpret the new hunk types as VERY large hunk and not load (error 103, not enough memory).

Future Solutions

The assembler and Lattice "C" may be changed to generate the new hunk types under programmer control.

How the Bits Work

The hunk size is a word containing the number of words in the hunk. Therefore, for the foreseeable future, including 32-bit address space machines, the upper 2 bits are unused. The bits have been redefined as follows:



0	0	If neither bit is set, then get whatever memory is available; this is "backward" compatible. Preference is given to "Fast" memory.
1	0	Loader must get FAST memory, or fail.
0	1	Loader must get CHIP memory, or fail.
1	1	If Bit31 and Bit30 are both set, then there is extra information available following this long word. This is reserved for future expansion, as needed. It is not currently used.

Perceived Impact

Old programs, programs that have not been compiled or assembled with the new options, and programs that have not been run through ATOM will run (or not run) as well as ever. This includes crashing in extended memory, if poorly programmed. The "previous solutions" mentioned at the beginning of this chapter still hold.

Program development and test on a 512K machine could follow EXACTLY the same loop you have now—edit, compile, link, execute, test, edit, . . . UNTIL you are about to release. Then you edit, compile, ATOM, Alink, add external memory (>512K) and test. This works well for all three environments (Amiga, IBM, and Sun).

For native (Amiga) development on a >512K machine you may want to ATOM the few required object files so you can both run your linked program in an extended memory machine and take advantage of a large RAM: disk. The development cycle then becomes: edit, compile, optionally ATOM (if this code or data contains items needed by the blitter), link, exteute, test, edit. . . .

"New programs" will not load in a V1.0 Kickstart environment. The result will be error 103 (not enough memory).

Old (V1.0 and before) versions of dumpobj and OMD will not work on files after ATOM has been run on them.

Working Environment

To get all of this to work together you need Release 1.1 compatible copies of:

ATOM (Version 1.0 or later)
Alink (Version 3.30 or later)
Kickstart (Release 1.1 or later) for DOS LOADER.
DumpObj (Version 2.1) Needed if you wish to examine programs modified by ATOM.

ATOM Command Line Syntax

The command line syntax is:

```
ATOM <infile> <outfile> [-I]
```

or

```
ATOM <infile> <outfile> [-C[CDB]] [-F[CDB]] [-P[CDB]]
```

Where:

<infile>	Represents an object file, just compiled, assembled or ATOMed (Yes, you can re-ATOM an object file).
<outfile>	The destination for the converted file.
-C	Change memory to CHIP
-F	Change memory to FAST
-P	Change memory to "Public". (Any type of memory available.)

C	Change CODE hunks
D	Change DATA hunks
B	Change BSS hunks

Command Line Examples

Example #1

In most cases there is no need to place CODE hunks in chip memory. Sometimes DATA and BSS hunks do need to be placed in chip memory; therefore the following is a fairly common usage of ATOM. To cause all Code hunks to go into Public RAM, Data and BSS hunks to go into chip RAM type:

```
ATOM infile.obj outfile.obj -pc -cdb
```

Example #2

To cause all the hunks in object file to be loaded into chip memory type:

```
ATOM infile.obj outfile.obj -c
```

Example #3

To set all data hunks to load into chip memory type:

```
atom myfile.o myfile.set.o -cd
```

Example #4

This is an interactive example. User input is in lower case, computer output is in UPPER CASE. In this example the code hunk is set to "Fast", the data hunk is set to "Chip". There were no BSS hunks. Note that help was requested in the beginning.

```
2> atom from.o from.set -i
AMIGA OBJECT MODIFIER V1.0
UNIT NAME FROM
HUNK NAME NONE
HUNK TYPE CODE                    {Note: code hunk}
MEMORY ALLOCATION PUBLIC
DISPLAY SYMBOLS [Y/N] y
__base..
__xcovf.
__CXD22..
__printf.
__main...
MEMORY TYPE? [F|C|P] ?            {Note: request for help}
Please enter F for fast
             C for Chip           {Memory type.
             P for Public
```

```
                Q to quit                 {cancels the operation, no output file is
                                          created}
                W to windup               {does not change the rest of the file,
                                          just passes it through}
                N for Next hunk           {skip this hunk, show next}
MEMORY TYPE? [FCP] f
UNIT NAME 0000
HUNKNAME NONE
HUNK TYPE DATA                            {Note: data hunk}
MEMORY ALLOCATION PUBLIC
DISPLAY SYMBOLS? [Y/N] n
MEMORY TYPE? [FCP] c
UNIT NAME 0000
HUNKNAME NONE
HUNK TYPE BSS
MEMORY ALLOCATION PUBLIC
DISPLAY SYMBOLS? [Y/N] y
MEMORY TYPE? [FCP] p
2>_
```

Error Messages

Error Bad Args:

a) An option does not start with a "-"
b) wrong number of parameters
c) "-" not followed by I, C, F, or P.
d) -x supplied in addition to -I, etc.

Error Bad infile:

File not found.

Error Bad Outfile:

File cannot be created.

Error Bad Type ##:

ATOM has detected a hunk type that it does not recognize. The object file may be corrupt.

Error empty input:

Input file does not contain any data.

Error ReadExternals:

External reference or definition if of an undefined type. Object file may be corrupt.

Error premature end of file:

An end of file condition (out of data) was detected while ATOM was still expecting input. Object file may be corrupt.

Error This utility can only be used on files that have NOT been passed through ALINK:

The input file you specified has already been processed by the linker. External symbols have been removed and hunks coagulated. You need to run ATOM on the object files produced by the C compiler or Macro Assembler BEFORE they are linked.

Creating a New Device to Run Under AmigaDOS

This section provides information about adding devices that are NOT part of the DOS filing system. The next section provides information about adding file-system-related devices (hard disks, floppy disks)—that is, devices that DOS can use to read and write files with their associated directories.

You would want to use this information to add a new device such as a new serial port or a new parallel port. In this case you may be creating a device named "SER2:" which is to act just like "SER:" as far as DOS is concerned.

There are two steps involved here. First, you must create a suitable device, a process that is not addressed here.

Note: The code for creating a skeleton disk-resident device is contained in the *Amiga ROM Kernel Manual.*

Second, you must make this new device available as an AmigaDOS device. This process involves writing a suitable device handler (see *ROM Kernel Manual*) and installing it into the AmigaDOS structures.

This installation is handled by creating a suitable device-node structure for your new device. This is similar to creating a DevInfo slot for a new disk device, except that the startup argument can be anything you want. The Segment list slot is zero, and the file name of your disk-resident device handler is placed in the Filename slot.

0	Next
0	dt_device
0	Task (or process id—see below)
0	Lock
BSTR	Filename of handler code
NNN	Stacksize required
NN	Priority required
XXX	Startup information
0	SegList (nonzero if you load the code)
0	Global vector required
BSTR	Device Name

The device handler is the interface between your device and an application program. This is normally written in BCPL, and the AmigaDOS kernel will attempt to load the code of the handler and create a new process for it when it is first referenced. This is handled automatically when the kernel notices that the Task field in the DevInfo structure is zero. If the code is already loaded, the code segment pointer is placed in the SegList field. If this field is zero, the

kernel loads the code from the filename given in the Filename field and updates the SegList field.

If you want this automatic loading and process initialization to work, you must create a code module, which is written in BCPL or is written in assembler to look like a BCPL module. This ensures that the dynamic linking used by the kernel will work correctly.

If you are writing in assembler, the format of the code section must be as shown below. Note that you may use DATA and BSS sections, but each section must have the same format as described here.

```
StartModule DC.L  (EndModule-StartModule)/4  Size of module in 1 words
EntryPoint  ....
            ....  (your code)
            CNOP  0,4                        Align to 1word boundary
            DC.L  0                          End marker
            DC.L  1                          Define Global 1
            DC.L  EntryPoint-StartModule     Offset of entry point
            DC.L  1                          Highest global used
            END
```

In assembler, you will be started with register D1 holding a BCPL pointer to the initial packet passed from the kernel.

If you are writing in BCPL, a skeleton routine will appear as follows. The main job of the device handler is to convert Open, Read, Write, and Close requests into the device read and write requests. Other packet types are marked as an error.

```
"Include files containing useful constants"

GET "LIBHDR"
GET "IOHDR"
GET "MANHDR"
GET "EXECHDR"

This is a handler for a skeleton Task.
When the task is created, the parameter packet contains the following:

parm.pkt!pkt.arg1 = BPTR to BCPL string of device name, (i.e., "SKEL:")
parm.pkt!pkt.arg2 = extra info (if needed)
parm.pkt!pkt.arg3 = BPTR to device info node

MANIFEST
$(
IO.blocksize = 30 (size of devices IO blocks)
$)

LET start (parm.pkt) BE
$(
LET extrainfo    = parm.pkt!pkt.arg2
```

```
LET read.pkt     = 0
LET write.pkt    = 0
LET openstring   = parm.pkt!pkt.arg1
LET inpkt        = VEC pkt.res1
LET outpkt       = VEC pkt.res1
LET IOB          = VEC IO.blocksize
LET IOBO         = VEC IO.blocksize
LET error        = FALSE
LET devname      = "serial.device*X00"
LET open         = FALSE (flag to show whether device has been "opened"
                   with act.findinput or act.findoutput).
LET node         = parm.pkt!pkt.arg3
```

(Zero the block first so that we can see what goes into it when we call Opendevice.)

```
FOR i=0 TO IO.blocksize DO IOB!i := 0
IF OpenDevice ( IOB, devname, 0, 0 ) = 0 THEN error := TRUE

IF error THEN
$( returnpkt (parm.pkt,FALSE,error,objectinuse)
   return
$)
```

(Copy all the necessary info to the Output buffer too.)

```
FOR i=0 TO IO.blocksize DO IOBO!i := IOB!i

outpkt!pkt.type := act.write
inpkt!pkt.type := act.read
node!dev.task := taskid()        (Insert process id into device node.)
```

(Finished with parameter packet . . . send back. . . .)

```
returnpkt (parm.pkt, TRUE )
```

(This is the main repeat loop waiting for an event.)

```
$( LET p = taskwait ()
SWITCHON p!pkt.type INTO
$(
CASE act.findinput:                (Open.)
CASE act.findoutput:
$( LET scb = p!pkt.arg1
   open := TRUE
   scb!scb.id := TRUE              (Interactive.)
   returnpkt (p,TRUE)
   LOOP
$)
```

```
CASE act.end:                          (Close.)
   node!dev.task := 0                  (Remove process id from device node.)
   open := FALSE
   returnpkt (p,TRUE)
   LOOP
CASE act.read:                         (Read request returning.)
   inpkt := p
   handle.return (IOBO,read.pkt)
   LOOP
CASE act.write:                        (Write request returning.)
   outpkt := p
   handle.return(IOBO,write.pkt)
   LOOP
CASE 'R':                              (A read request.)
   read.pkt := p
   handle.request(IOB,IOC.read,p,inpkt)
   inpkt := 0
   LOOP
CASE 'W':                              (A write request.)
   write.pkt := p
   handle.request(IOBO,IOC.write,p,outpkt)
   outpkt := 0
   LOOP
     DEFAULT:
   UNLESS open DO node!dev.task := 0  (Remove process id unless open.)
     $)
   $) REPEATWHILE open | outpkt = 0 | inpkt = 0
   Termination
   CloseDevice( IOB )
```

(Handle an IO request. Passed command, transmission packet (tp) and request packet (rp). rp contains buffer and length in arg2/3.) AND handle.request (IOB, command rp, tp) BE

```
LET buff = rp!pkt.arg2
LET len = rp!pkt.arg3
SetIO( IOB, command, ?, rp!pkt.arg3, 0 )
putlong ( IOB, IO.data, buff )
SendIO(IOB, tp )
```

Handle a returning IO request. The user request packet is passed as p, and must be returned with success/failure message. AND handle.return (IOB, p) BE

```
$(
LET errcode = IOB 0.error
LET len = getlong( IOB, IO.actual )
```

```
TEST errcode = 0 THEN      (No error.)
   returnpkt(p, len )
ELSE
   returnpkt(p, -1, errcode )
```

If you wish to write your device handler in C, you cannot use the automatic load and process creation provided by the kernel. In this case, you must load the code yourself and use a call to CreateProc to create a process. The result from this call should be stored in the Task field of the DevInfo structure. You must then send a message to the new process to get it started. This message might contain such things as the unit number of the device involved. The handler process should then wait for Open, Read, Write, and Close calls and handle them as described in the example above. C code does not need to insert the process id into the device node because this is done when code is loaded, as described above.

Making New Disk Devices

To create a new disk device, you must construct a new device node as described in Section 3.3.1 of the *AmigaDOS Technical Reference Manual*. You must also write a device driver for the new disk device.

A device driver for a new disk device must mimic the calls that are performed by the trackdisk device (described in the Amiga ROM Kernel Manual). It must include the ability to respond to commands such as Read, Write, Seek, and return status information in the same way as described for the trackdisk driver.

For the following description, note that most pointers are of the type BPTR (as described earlier in the *AmigaDOS Technical Reference Manual*), a machine pointer to some long word-aligned memory location (such as returned by AllocMem) shifted right by two.

Construct the new node with the following fields:

0	Next
0	dt_device
0	Task
0	Lock
0	Handler
210	Stacksize
10	Priority
BPTR to startup info	
Seglist	
0	Global vector
BSTR to name	

The BSTR to a name is a BCPL pointer to the name of your new device (such as HD0:) represented as the length of the string in the first byte, and the characters following.

The Seglist must be the segment list of the filing system task. To obtain

this, you must access a field in the process base of one of the filing system tasks.

The code as follows can be used for this purpose:

```
UBYTE *port;
port = DeviceProc( "DF0:");                    /* Returns msg port of
                                                  filesystem task */
task = (struct Task *) (port-sizeof(struct Task); /* Task structure is
                                                  below port */
list = ( task.pr_Seglist )                     /* make machine ptr
                                                  from SegArray */
segl = list[3];                                /* Third element in
                                                  SegArray is filesystem
                                                  seglist */
```

Next, you must set up the startup info (again, remember to use BPTRs where needed). This info consists of a BPTR to three long words which contain:

- Unit number (do not use unit zero)
- Device driver name, stored as a BPTR to the device driver name which must be terminated by a null byte which is included in the count (e.g., 4/'H'/'D'/'O'/O) BPTR to disk information

The disk size information contains the following long word fields:

11	Size of table
128	Disk block size in long words (assuming 512-byte blocksize)
0	Sector origin (i.e., first sector is sector zero)
Number of surfaces	(e.g., 2 for floppy disk)
1	Number of sectors per block
Number of blocks per track	(e.g. 11 for floppy disk)
2	(or more, indicating number of blocks to be reserved at start)
0	Preallocation factor
0	Interleave factor
Lowest cylinder number	(commonly 0)
Highest cylinder number	(e.g., 79 for floppy disk)
5	(or more, indicating number of cache blocks)

Finally, the device node must be attached to the end of the list (note the Next fields are all BPTRs) of device nodes within the Info substructure.

WARNING: The list to which this refers is NOT the same kind of list that is referenced in the Exec portion of the *Amiga ROM Kernel Manual,* but is instead the kind of list described in this book.

To partition a hard disk you make two or more device nodes and set the lowest and highest cylinder numbers to partition the disk as desired.

Using AmigaDOS Without Workbench/Intuition

This information is provided to give developers some information about how AmigaDOS and Intuition interact with each other. As of this writing, it is not possible to fully close down Intuition or the input device. It is possible to install one's own input handler within the input stream (as is demonstrated in the *Amiga ROM Kernel Manual,* Input Device description) and thereby handle input events yourself, after your program has been loaded and started by AmigaDOS. If, after that point, you take over the machine in some manner, you can prevent AmigaDOS from trying to put up system requesters or otherwise interacting with the screen by modifying DOS as shown below. Basically, your own program must provide alternate ways to handle errors that would normally cause DOS to put up a requester.

Another alternative for taking over the machine is to ignore the AmigaDOS filing system altogether, and use the trackdisk.device to boot your code and data on your own. You will find details about the disk boot block and the track formatting in the *Amiga ROM Kernel Manual,* allowing this alternate means if you so choose.

Here are the details about AmigaDOS and Intuition:

AmigaDOS initializes itself and opens Intuition. It then attempts to open the configuration file (created by Preferences) and passes this to Intuition. It then opens the initial CLI window via Intuition and attempts to run the first CLI command. This is commonly a loadwb (load Workbench), followed by an endcli on the initial CLI.

An application program can be made to behave like Workbench, in that it spawns a new process. The next CLI command is then endcli, which closes everything down, leaving only the new process running (along with the filesystem processes). This process would set the pr_WindowPtr field to –1, which indicates that the DOS should report errors quietly. Note that the application MUST handle all errors. There are further details on this in Chapter 3. DOS will also have initialized the TrapHandler field of the user task to point to code that will display a requester after an error; this should be replaced by a user-provided routine. This will stop all uses of Intuition from the user task, provided there are no serious memory corruption problems found, in which case DOS will call Exec Alert directly.

There is still the problem that the filesystem processes may ask for a requester, in the event of a disk error or if the filesystem task crashes due to

memory corruption. To stop this, the pr_WindowPtr and tc_TrapHandler fields of the filesystem tasks must be set to –1 and a private Trap handler must be provided in the same way as was done for the user task. This is easily done as shown below.

Find the message port for each filesystem task by calling DeviceProc(), passing DF0, DF1, etc. An error indicates that the device is not present. From the message port you can find the task base for each filesystem task, and hence patch these two slots. This should be repeated for each disk unit.

The application program can now close Intuition. Workbench has, of course, never been invoked. Note that as of this writing, it is not possible to stop DOS from opening Intuition.

Note that if the applications want to use any other device such as SER:, the handler process must be patched in exactly the same way as the filesystem processes. The application should obviously not attempt to open the CON: or RAW: once Intuition has become inactive.

Index