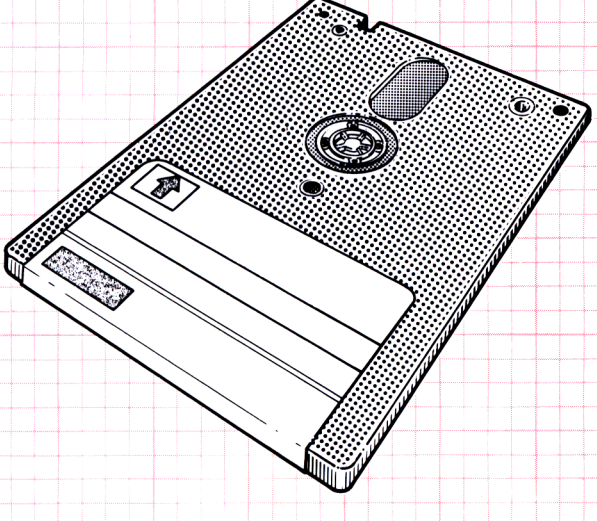
Using Your Amstrad CPC Disc Drives

J.W. PENFOLD



USING YOUR AMSTRAD CPC DISC DRIVES

OTHER BOOKS OF INTEREST

BP153 An Introduction to Programming to Amstrad CPC 464 and 664
BP159 How to Write Amstrad CPC 464 Games Programs
BP175 How to Write Word Game Programs for the Amstrad CPC 464, 664 and 6128
BP191 Simple Applications of the Amstrad CPCs for Writers

* * *

BP187 A Practical Reference Guide to Word Processing on the Amstrad PCW 8256 and PCW 8512
BP188 Getting Started with BASIC and LOGO on the Amstrad PCW 8256 and PCW 8512

* * *

BP112 A Z80 Workshop Manual
BP152 An Introduction to Z80 Machine Code

USING YOUR AMSTRAD CPC DISC DRIVES

by

J. W. PENFOLD

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

PLEASE NOTE

Although every care has been taken with the production of this book to ensure that any projects, designs, modifications and/or programs etc. contained herein, operate in a correct and safe manner and also that any components specified are normally available in Great Britain, the Publishers do not accept responsibility in any way for the failure, including fault in design, of any project, design, modification or program to work correctly or to cause damage to any other equipment that it may be connected to or used in conjunction with, or in respect of any other damage or injury that may be so caused, nor do the Publishers accept responsibility in any way for the failure to obtain specified components.

Notice is also given that if equipment that is still under warranty is modified in any way or used or connected with home-built equipment then that warranty may be void.

First published – November 1986

British Library Cataloguing in Publication Data
Penfold, J.W.
Using your Amstrad CPC disc drives
1. Amstrad CPC464 (Computer).
2. Amstrad CPC664 (Computer)
3. Amstrad CPC6128 (Computer)
4. Data disc drives
I. Title
004.5'6 QA76.8.A4

ISBN 0 85934 163 1

Printed and bound in Great Britain by Cox & Wyman Ltd, Reading

Preface

The Amstrad 6128 offers remarkable value for money, not least because of the built-in disc drive. The purpose of this book is to show how this disc drive can be used to best effect.

The book covers all aspects of disc drive usage, from loading programs to using the discs to store data files from applications, and using the discs to help write your own programs in BASIC and other languages.

Both the native AMSDOS and the CP/M operating systems are covered, and the use of some of the most popular applications programs are described (including TASWORD 6128, which is actually being used to write this book, and DR DRAW, with which some of the illustrations were prepared).

Though this book is mainly written for users of the Amstrad model 6128, much of the information also applies to users of the 664 and the 464 with DD1. Some of the general information and CP/M Plus details also apply to the Amstrad PCW 8256 and PCW 8512.

It is hoped this book will help users make fuller use of this remarkably versatile computer.

J. W. Penfold

CONTENTS

Chapter 1

DISCS – WHAT AND WHY

All computers need some form of storage for programs and data which will enable them to be kept when the computer is turned off. With the exception of some small pocket computers which have batteries to enable memory contents to be retained when the computer is not in use, all memory contents is lost when a machine is turned off.

For home computers, the original form of data and program storage was on ordinary audio cassettes, recorded using ordinary cassette recorders plugged into the computer. This had the advantage of being inexpensive, especially if you already had the cassette recorder. There were two problems. Firstly, both recording and loading data or programs was slow, because of the limited quality of the signal that could be recorded. The second problem was reliability. Not all cassette recorders would work successfully with all computers. Some computers would not work reliably with any cassette recorder!

To help with these problems, special data recorders were introduced. These had special equalisation and phasing controls to produce signals more suited to computer use, and were also equipped with tape counters to help find programs in the middle of long tapes, but they had the disadvantage that they could be quite expensive, and they didn't overcome all the problems caused by the slow tape speed.

Thus, most home computer users aspired to own a disc drive. Disc drives are fast, and generally reliable, but were expensive. At first they could cost easily double the price of a home computer. However, prices have been falling, and now the Amstrad 6128 can be bought with a disc drive built in, at a very competitive price. It may well be that all future computers will have to have a disc drive built in if they are to be successful.

To understand why discs are fast you need to know how they work. A disc is made of a plastic material coated on both sides with a magnetic layer, essentially similar to the coating on magnetic tape. The actual disc is enclosed in some form of

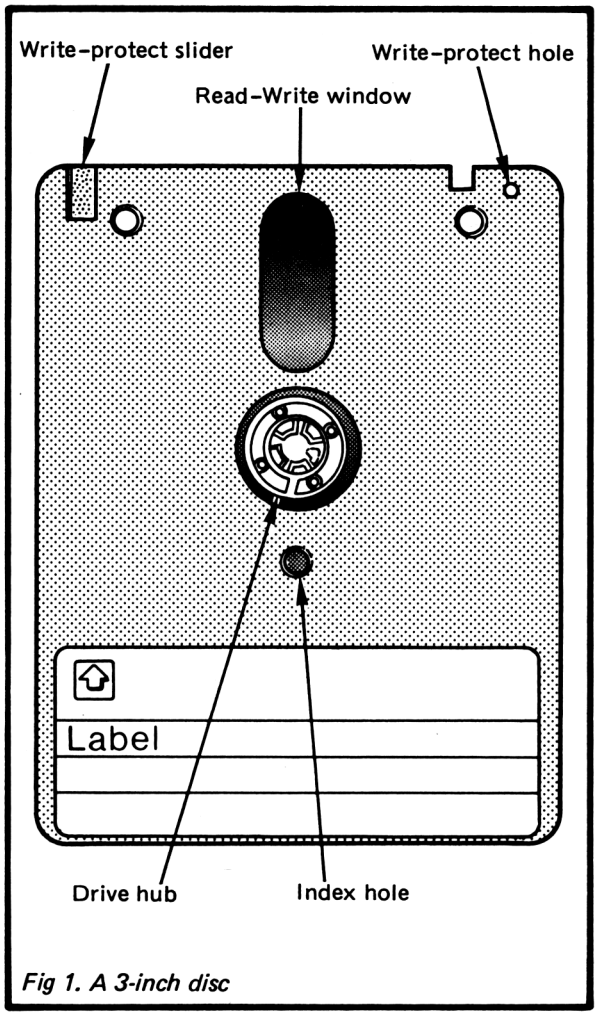


Fig 1. A 3-inch disc

protective casing. On early discs, this was an envelope of fibrous material, but on the latest small discs, like those used in the Amstrad drives, the casing is of rigid plastic, offering much better protection.

All discs are coated on both sides, though some are sold as 'single-sided' as one side does not pass the inspection tests of the manufacturer. All 3-inch discs for the Amstrad are, however, double-sided. The second side is accessed by turning the disc over, like a gramophone record, and the two sides are treated as separate entities. These discs may however, sometimes be described as 'single-sided reversible', to distinguish them from the discs for the PCW 8256 second drive. This drive can access both sides of a disc, using two heads.

Discs are sometimes called 'floppy discs', because the plastic is flexible. However, this is really something of a misnomer, as the discs are actually quite stiff. The discs are spun in the drive at some speed (hundreds of revolutions per second) and the effect of this spinning is to hold the disc quite rigid by centrifugal force. The name 'floppy' is actually to distinguish these discs from the hard discs (or 'Winchesters') used with large business machines. These offer much larger storage and much faster access speeds than floppies. Hard discs really are hard, being aluminium coated with the magnetic material.

The data is recorded on the disc in the form of tracks. These tracks take the form of concentric circles round the disc, rather than a single spiral track like a gramophone record. The read/write head in the disc drive can step quickly from track to track. As used on the Amstrad computers (with the exception of the second drive for the PCW 8256 and 8512) the discs have 40 tracks. This is described as 'single-density'. Double density, with 80 tracks on the disc, is used by the PCW 8256 and 8512 second drive.

Each track is divided into nine sectors, by means of special signals recorded onto the disc. By reading these signals, the computer can always know where on the disc the head is reading from or writing to. Because of the speed at which the disc rotates, and the speed at which the head can step from track to track, the computer can move to and access any particular place on the disc very quickly.

40 tracks with 9 sectors on each gives a total of 360 sectors.

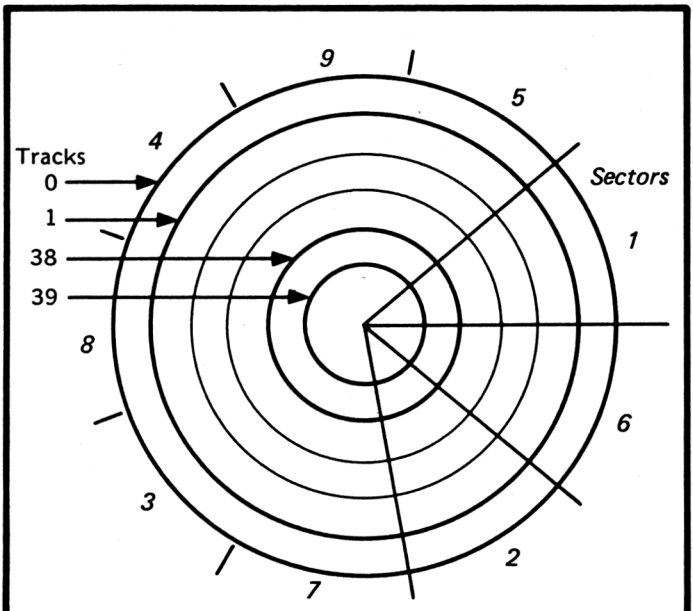


Fig 2. Arrangement of tracks and sectors

As each sector can store half a kilobyte (512 bytes) this gives a total disc storage of 180K, However, some of this is used for particular purposes, so the actual amount of data which can be stored will always be slightly less than this.

Because of the fast rotational speed of the discs, the data can be recorded on to them at a very fast rate, much faster than it can be recorded onto tape cassettes.

These two factors, the ability to move quickly to anywhere on the disc, and the high rate of data transfer, combine to give disc drives their speed.

Because of this ability to move quickly to anywhere on the disc, discs are a 'random access' medium, as opposed to tape, which is a 'serial access' medium, as you have to run through the whole tape to find a file in the middle (albeit that you can speed things up a bit by using the fast forward/rewind controls).

As supplied, discs do not have the sector markings on them. These have to be recorded onto the disc by the user before the disc can be used, a process known as 'formatting'. A special program is provided as part of the CP/M system, to enable this to be done. Discs for the Amstrad can be prepared in a number of different formats, depending on how they are to be used. Some formats allow more data to be recorded on the disc than others.

Even given the sector and track arrangement of discs, it would still take the computer some time to find a particular file, be it program or data, if the computer had to search the whole disc for it. To obviate this necessity, part of the disc is given over to a 'directory'. This contains details of all the files on the disc, together with the tracks/sectors where the file is recorded. This directory is updated whenever you save a program or data file, and at other times during file operations.

By reading the directory, the computer is able to move directly to the right place on the disc and begin reading in the data. The directory information is also used to find a free space of suitable size on the disc when recording data.

When a file is no longer required, it can be erased. In fact, when this is done, it is only the directory entry which is actually erased. The actual data remains on the disc, but can no longer be read, and may be overwritten by subsequent 'save' operations.

The creation and maintenance of the directory is completely automatic, and does not require any specific action by the user. However, you can consult the directory to see what files are on the disc. This is done by the CAT command from BASIC, or by the CP/M DIR (obviously short for directory) command. There is also an AMSDOS external |DIR command, which is different to the CP/M DIR and the BASIC CAT.

Care of Discs

Being a magnetic medium, the data recorded on discs can be corrupted by magnetic fields. For this reason, discs should not be

stored too close to the computer, which generates magnetic fields, or to peripherals such as monitors and printers. The electric motors in trains have also been blamed for corrupting discs, but this has not been conclusively proved. However, if travelling by train with discs, it would be prudent to choose a seat as far from the power car as possible. In the case of tube trains, do not place the bag containing the discs on the floor of the train, as all the electrics are under the floor.

By far the most damaging source of magnetism, and the one which is used by computer authors who may sometimes need to deliberately corrupt a disc (to see how a computer copes) is the telephone. If you place a disc under a telephone and wait for it to ring, the data on the disc can be guaranteed to be dead and gone forever.

The plastic cases of 3 inch discs offer good protection from physical damage, and have metal shutters to cover the window through which the drive read/write head reaches the disc. This helps prevent the ingress of dust and dirt, and also accidental contact from fingers. This window is opened by a slider on the side of the disc when it is inserted. It is not a good idea to open this shutter by hand for any reason.

Discs should not be left lying around on a desk where they can gather dust. They should always be kept in the protective plastic cases. If a dusty disc is placed in a drive, some of the dust will get into the drive mechanism, which will do it no good at all.

Discs can be worn out by extensive use, so with discs containing important programs, it is recommended to make copies and work from these, keeping the original 'master' discs in a safe place. With program discs it is always a good idea to open the 'write protect' notch to prevent accidental erasure. However, a few programs can only work if the disc is write-enabled.

The other main enemy of discs is heat. If a disc becomes too hot the magnetic recording can 'fade', resulting in loss of data. In extreme cases of overheating, the disc may distort, and this could lead to damage both to the disc and to the drive mechanism.

The Disc Drive

The disc drive is built into the right-hand end of the the Amstrad CPC 6128 computer. There is a slot with a dust flap, into which

the disc is inserted, an oblong 'button', and an indicator light.

When inserting a disc, it should be pushed straight with the side carrying the files you wish to access upwards. It should be pushed in until it drops down with a click, and the button pops out.

The indicator light glows dimly all the time the computer is switched on. It becomes brighter when the disc is actually being read from or written to.

To remove a disc, you press the button, and allow the disc to spring out of the drive an inch or so. It can then be easily pulled right out.

You will find that the motor starts a little before the indicator light glows bright. This is to allow the disc to reach normal operating speed. The motor also runs for a short time after the reading or writing has finished (you can easily hear when the motor is running – there is no visual indication). However, you do not have to wait for the motor to stop before removing the disc.

You should not, however, ever remove a disc while the indicator light is glowing bright. If you do this, either a read or a write failure is bound to occur, and you could corrupt files other than the one you were accessing on the disc.

The Second Drive

A single disc drive is fine for loading programs and for simple file reading and saving, but it becomes a bit of a nuisance when trying to make backup copies of files or programs. Special programs are included with the supplied CP/M 2.2 and CP/M Plus software to allow for disc copying on a single drive, but these can involve a lot of disc swapping and therefore need constant attention when in use.

When working under CP/M Plus, the one disc drive is treated as if it were actually two drives, A and B, and there is an indication at the bottom of the screen as to which it is at any time. This simplifies file copying, but still requires a lot of disc swapping. Many CP/M programs use overlays. That is, not all of the program is loaded into memory at once. Parts of the program are only loaded when needed, and 'overlay' other parts of the program, thus allowing less memory to be used than if the whole program was loaded.

Programs such as wordprocessors and databases may also only

load a part of the file on which they are working into memory at any one time, the rest being 'buffered' onto the disc. This allows much larger files to be handled than could be the case if it all had to be in memory at once.

It follows that when working with CP/M you will often need to keep a program disc in drive A and a data disc in drive B. This is possible with the single (physical) drive under CP/M Plus, as it is treated as two (logical) drives, and you are automatically prompted when to swap discs. However, this is not very convenient for extensive use. Serious work under CP/M 2.2 is not really possible with only one drive.

It follows that if you want to do serious work with CP/M programs, it is well worth getting the second disc drive. This drive has its own power supply built-in, and connects to the computer by a length of ribbon cable, which is not supplied with it (at time of writing).

When fitted, the second drive becomes drive B, the one in the computer being drive A. Once the second drive is fitted. the drive in the computer *cannot* be treated as two logical drives.

The second drive for the CPC computers uses exactly the same disc formats as the drive in the computer, so discs can be swapped freely between the two drives and read successfully. This is not so with the second drive for the PCW 8256 or 8512 which uses a completely different format to the first drive, and indeed different discs.

The second disc drive has two indicator lights on it, one green and one red. The green light indicates that the disc drive is connected to the mains and turned on. The red indicator light glows all the time the computer is turned on, except when the drive in the computer is running. The red indicator on the second drive does not, therefore, indicate when the disc in the second drive is actually being accessed. It is not, therefore, a good idea to remove a disc from the second drive while the motor is running.

You must always turn the second disc drive on before turning on the computer. During the computer's start up sequence, it checks to see what peripherals are connected. If the second disc drive is not switched on during this sequence, the computer cannot detect its presence, and so you will not be able to use the second drive.

Chapter 2

OPERATING SYSTEMS

All computers need to have some system built-in to do such jobs as reading in files from discs and printing characters on the screen or on the printer. In the simplest home computers these routines are a part of the BASIC language interpreter. On more sophisticated computers, these routines are kept separate. This makes it possible for alternative languages to use the routines, saving programmers the trouble of having to write their own. The routines can also be used directly by machine code applications programs.

A computer may have an operating system which is unique to it, usually termed a MOS (for Machine Operating System), also called a 'native environment'. Alternatively, the machine may use an operating system common to several computers. The advantage of such a system is that it enables several makes and types of computer to use the same programs written to run under that particular operating system. The operating system is written for each computer to use the particular hardware features that computer provides, but the operating system makes the computers all look the same to the programs. (This is the ideal situation, in fact some modification to programs is often necessary for different machines).

All operating systems have the basic routines for input and output, and these are called the BIOS, (basic input-output system, for keyboard and screen, sometimes collectively termed the 'terminal' or 'console'), and the routines for disc access, called the BDOS (basic disc operating system). Note that the 'basic' in these terms has nothing to do with the BASIC computer language.

In addition, a full operating system will provide facilities for such things as disc formating, file copying, and switching input/output between peripherals. These facilities will be available by means of direct commands from the user. The part of the operating system which interprets these commands, which the operator types in, is called the CCP, or 'console command processor'.

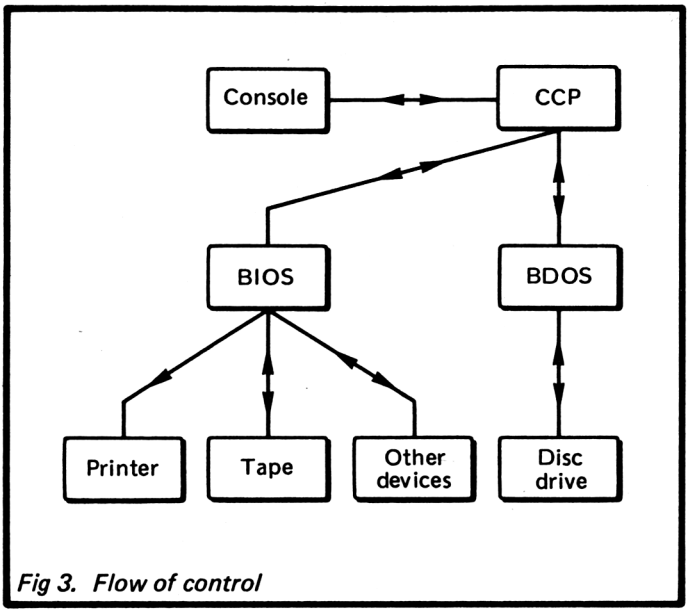


Fig 3. Flow of control

The Amstrad 6128 has both a native environment, called AMSDOS, and a standard operating system. The standard operating system is CP/M, which is probably the best known and widest used operating system of all.

AMSDOS is not, in fact, a complete operating system as it does not, for example, have a disc formatting facility included. It can, however, be used from BASIC programs for disc operations. There are also a number of popular applications for the Amstrad computers which run under AMSDOS rather than CP/M, and they include TASWORD 6128 on which this book was written.

In AMSDOS, the commands are in the form of RSXs. This stands for Resident System Expansions, and they are used just like BASIC keywords, either from command mode, or within programs. The only difference is that each RSX command must be preceded with a vertical bar (shifted @) to distinguish it from a regular BASIC keyword. These RSXs are contained in ROM

within the computer and are available as soon as the BASIC 'Ready' prompt appears. In addition, the cassette filing commands within BASIC from the Amstrad CPC 464 are redirected to work with the disc drive on the CPC 664 and 6128. These include LOAD, SAVE, OPENIN, OPENOUT, etc.

CP/M provides all the facilities missing from AMSDOS, including disc formatting, disc copying, and file copying. CP/M also includes a number of utility programs, including a simple text editor, and an 8080 assembler. However, CP/M facilities are not available to programs running under the native Locomotive BASIC on the Amstrad. There are a great number of programs available to run under CP/M, including powerful word processors, data-bases and spreadsheets, though many of these need a second disc drive in order to work effectively (because they expect to have the program disc in one drive and a disc for data storage in the other). There are also special applications programs, such as the DR DRAW graphics program which was used to generate some of the graphics in this book.

The CP/M software is provided on disc, and it has to be loaded into the computer before the CP/M commands can be used (though in fact the CP/M BIOS is in ROM). CP/M has to be loaded from special system format discs. The computer is ready to accept CP/M commands when the 'A>' prompt appears on the screen. Some of the commands are always present in the computer and are directly available. Others require extensive code and are only loaded in when the command to use them is given, to avoid taking up memory space unnecessarily. In order to use these commands, there must be a system disc with the appropriate program on in the disc drive.

The version of CP/M supplied with the Amstrad 6128 is called CP/M Plus (or version 3). This version has been written to use memory bank switching, and can therefore make use of the full 128k of memory in this machine. With CP/M Plus, all of the operating system can be held in RAM, so all the console commands can be used without needing to have a system disc in the drive. It is only the more extensive transient programs which have to be loaded in for use. The earlier version of CP/M, version 2.2, is also supplied for compatibility with the models 664 and 464 with DD1.

Both CP/M and AMSDOS use the same disc formats for files, which means that, to some extent, you can transfer data between CP/M programs and AMSDOS programs. This is dependent, of course, on the two programs using the same sort of data. For example, you could read a file containing only ASCII characters produced on one word processor into another word processor using the other operating system. Of course, there would be problems if the file contained special characters (for such things as printer controls), but this would also apply to two word processors using the same O.S.

AMSDOS Commands

In this section we will look at each of the AMSDOS commands in turn, both the external commands (RSXs) and the re-directed BASIC commands. The former can be identified by the fact that they are preceeded by the vertical bar (|). Special syntax is sometimes necessary when using these commands.

|A

This command will set drive A as the default drive. For this command to work, there must be a disc in drive A, and it must be correctly formatted.

Note that when AMSDOS starts, drive A is set as the default drive anyway.

|B

This command will set drive B as the default drive. For this command to work, there must be a second drive fitted (AMSDOS does not map two logical drives on to the one physical drive), and it must contain a correctly-formatted disc.

CAT

This command causes the computer to read the disc directory and display a catalogue of files on the screen. The files are sorted into alphabetical order, and the size of each file is also shown, together with the free space remaining on the disc. If you have two drives fitted, CAT will operate on the currently selected drive.

CAT displays the currently selected drive, and also the current 'user number'. The user number is a way of partitioning up the directory, and is described under the '|USER' command.

CHAIN

This command loads and runs a named program from disc. When operating from tape, the filename may be omitted, in which case the next program found on the tape will be loaded and run, but when using discs, a filename is essential.

A line number may also be specified. When it is, the program will run from this line.

Examples:

CHAIN "PART2.BAS"
CHAIN "PART3.BAS",220

CHAIN MERGE

This command loads a program and merges it with the current program in memory, running the combined program either from the beginning or from a specified line. You may also specify a range of lines in the original program to be deleted.

In effect, this command allows you to use overlays in BASIC programs, retaining the variables, including arrays. However, any user defined functions (DEF FN) are forgotten, ON ERROR GOTO is turned off, and any open files are abandoned and any buffered input is lost (i.e. characters typed ahead at the keyboard).

An implicit RESTORE is performed, and you should not use CHAIN MERGE within a FOR or WHILE loop, or within a sub-routine, as the stack of return addresses is cleared.

Examples:

CHAIN MERGE "LOADER.OVL",220
CHAIN MERGE "OVERLAY3.BAS",190,DELETE 2000-2300

CLOSEIN

Closes a file on disc from which you have been reading, i.e, an input file. This frees the channel to enable you to open another file for reading.

All files should be properly closed when you have finished with them, though forgetting to close an input file is not as bad as forgetting to close an output file.

CLOSEOUT

Closes a file on disc to which you have been writing, i.e, an output file.

Using this command ensures that all the data is written to the disc, and that the disc directory is properly updated. If you forget to close an output file, some data will almost certainly be lost.

|CPM

This command leaves BASIC and AMSDOS and loads ("cold boots") CP/M from disc. A suitable system disc must be in drive A, otherwise the computer will return to BASIC with an error message.

By suitably configuring the system disc, this command may be used to boot CP/M, and then load and run a program directly without further commands from the keyboard.

When this command is issued, any BASIC program is lost, as are any open files.

A CP/M utility, AMSDOS.COM, is provided to give a clean return to AMSDOS and BASIC. Alternatively, the computer may be reset by pressing SHIFT, CONTROL and ESC simultaneously.

|DIR

This command produces a directory of the files on the disc. Unlike the BASIC CAT, the files are not sorted into alphabetical order, and the sizes of the files are not shown. However, the amount of free space remaining on the disc is displayed, as is the current drive and user number.

|DISC.IN

This command directs the filing system firmware to use the disc system rather than the cassette system for input. The default is to use the disc system, so there is no point in using this command unless you have previously switched to using the cassette system.

The ability to use the disc system for input and the cassette system for output simultaneously provides a means for transferring some types of file from disc to tape, perhaps for archival storage purposes.

|DISC.OUT

This command directs the filing system firmware to use the disc system rather than the cassette system for output. Again the

default is to use the disc system, so this command is only of use if you have been using the cassette system.

By using the tape system for input and the disc system for output, you can transfer some types of file from tape to disc, allowing faster access in future, this is useful if you have added a disc drive to a CPC 464, or upgraded from that model to a 6128.

|DISC

This command combines the effects of the previous two. Once more, as the default is to use the disc drive system, this command is only of use to switch back to the disc system after using the cassette.

|DRIVE

This command changes the currently selected drive. In effect, it provides the same function as |A and |B, but it allows the drive identifier to be a string variable, which can be useful within programs.

This command requires a single string parameter which must be a single letter in the range A to P, upper or lower case. Only two disc drives may be fitted normally, A and B, but letters C to P are for future expansion. A third-party silicon disc is already available, and this can be used as drive C.

When passing a string variable to an external command, it must be done by address. This is done by preceeding the variable name with the @ symbol.

Examples:

|DRIVE,@drive$

|DRIVE,"A"

Note the comma between the command and the parameter. This is normal syntax with external commands.

EOF

This is a function which tests for the end of a file. It returns −1 if the end of the file has been reached (true) and 0 if it has not (false). It is used when reading in files, to avoid reading past the end of the file (which would cause an error).

|ERA

This command erases files from a disc. The name of the file after

the command and may contain wildcards. That is to say, "?" may be used to represent any single character in the filename, and "*" may be used to represent any sequence of characters.

When wildcards are used, if more than one filename on the disc matches the specified filename, all matching files are erased. Care should be taken here.

In fact, it is only the directory entries which are erased. The files remain intact on the disc until they are overwritten by other files. However, they cannot easily be recovered.

Examples:-

|ERA,"PROG1.BAS" (erases one file)
|ERA,"*.BAS" (erases all files of type .BAS)
|ERA,"MYTEXT?.*" (erases MYTEXT1.DOC, MYTEXT2.DOC, MYTEXT1, BAK, and possible others).

INPUT#9

This command reads data from a file on disc. It is normally used to read files which have been produced from BASIC using WRITE#9.

The input is placed into variables, which must be of the correct type for the data. You can read numbers into string variables (they will be treated as strings of numerals) but you cannot read strings into numeric variables (an error will occur).

You may read data into more than one variable in a single INPUT statement.

LINE INPUT#9

Similar to INPUT#9, but more useful with files which have been created with PRINT#9 rather than WRITE#9. Creating files from BASIC is fully covered in Chapter 4.

MERGE

Merges a BASIC program from disc with the program currently in memory. If the program on disc has line numbers which coincide with line numbers in the program in memory, the lines from the disc overwrite the ones in memory.

When you use MERGE, all variables and arrays are lost, user functions are forgotten, ON ERROR GOTO is turned off, all

open files are abandoned and any buffered output is lost DEFINT, DEFSTR and DEFREAL settings are cleared, and an implicit RESTORE is performed.

MERGE is mostly used for building up long programs by adding 'library' subroutines from disc. For those who write their own programs, it can be worthwhile building up a collection of general-purpose routines for this purpose.

You cannot MERGE a protected BASIC program.

OPENIN

Opens a file on disc for input. The file can be read by INPUT#9 or LINE INPUT#9.

OPENIN must be followed by the name of the file to be read. If the filename is immediately preceeded by an exclamation mark, this has the effect of suppressing the usual screen messages when the file is being read. The excalmation mark does not form part of the filename for directory search purposes.

OPENOUT

Opens a file on disc for output. The file can then be written to with WRITE#9 or PRINT#9. OPENOUT must be followed by the name for the file, which for disc operations, must conform to CP/M filename requirements.

Again, if the first character of the filename is an exclamation mark, the screen messages during file writing operations are suppressed. The exclamation mark does not form part of the filename as used in the directory.

PRINT#9

Writes a variable or list of variables to a disc file. To use this command, a file must be open on the disc (i.e. you must have previously used OPENOUT).

This command is too complex for a simple but useful description. Chapter 4 covers the subject of creating and using files from BASIC in detail.

|REN

Rename. This changes the name of a file currently on disc. The new name must not be the same as a file already on the disc.

This command requires two parameters. The new name for the file is given first, followed by the old name. Wild cards cannot be used.

Examples:-

|REN,"DATA1.DOC","DATA.TMP"
|REN,@newname$,oldname$

RUN

Loads and runs a program from disc. The program is run from the first line.

Any existing program is lost, as are any open files, buffered output, DEFINT, DEFSTR or DEFREAL settings, and any user defined functions. An implicit RESTORE is performed.

Unlike cassette operations, where the use of a filename is optional (if omitted the computer loads and runs the next program encountered on tape), a filename must always be specified for disc operations.

This command is the equivalent of LOAD followed by RUN, except where protected BASIC programs are concerned. A protected BASIC program can only be loaded and run with RUN followed by the filename.

SAVE

Saves a BASIC or machine code program or an area of memory to disc. This command can take up to three parameters.

The first parameter must always be specified, and is the name for the file. It must conform to CP/M filename conventions. If it is the same as a file already on the disc, the new file will be saved with the specified name, and the old file will be renamed with the specifier .BAK, thus giving an automatic one-stage backup.

The second parameter is optional and specifies the type of file to be saved. If no type is specified, the file saved is a BASIC program in unprotected internal format. The other type specifiers are A, which saves a BASIC program in ASCII format (in which form it can be loaded into most word processors for editing), P, which saves a BASIC program in protected form, so that it cannot be copied, listed or edited, and B, which saves a binary file, which may be a machine code program or area of memory.

The third parameter (in fact, third and fourth) are essential

when saving a binary file, but are illegal otherwise. They are the start address and the length (in bytes) of the area of memory to be saved. Additionally, an entry point address for a machine code program may be specified. If this is not specified, the entry point will be assumed to be the same as the start address.

SAVE cannot be used to save only a part of a BASIC program. That is, you cannot specify a line number range. It is, however, possible to save part of a program in ASCII format by using LIST#9. This can subsequently be LOADed or CHAIN MERGEd.

|TAPE.IN

Sets the computer to accept input from a cassette recorder instead of the disc drive. By using this, BASIC programs on tape can be loaded from tape and then saved to disc for subsequent easier access. Data files created with WRITE#9 can also be transfered to disc by means of a short BASIC program. This may be of interest to those who have upgraded from a CPC 464.

Note that protected BASIC programs cannot be transferred in this way, and there may also be problems with some commercial software which overwrites the area of memory used by the disc system.

|TAPE.OUT

Sets the computer to send output to a cassette recorder instead of the disc drive. This can be useful to transfer data files created with WRITE#9 from disc to cassette for archival storage. Cassettes are less expensive than discs for this purpose. Programs can also be transferred (subject to the limitations explained in the previous command) but there is normally little point in this.

|TAPE

This combines the effects of the two previous commands, setting the computer to use the tape cassette interface for all filing operations.

|USER

This command changes the current user number. User numbers can be used to 'partition' the disc drive directory.

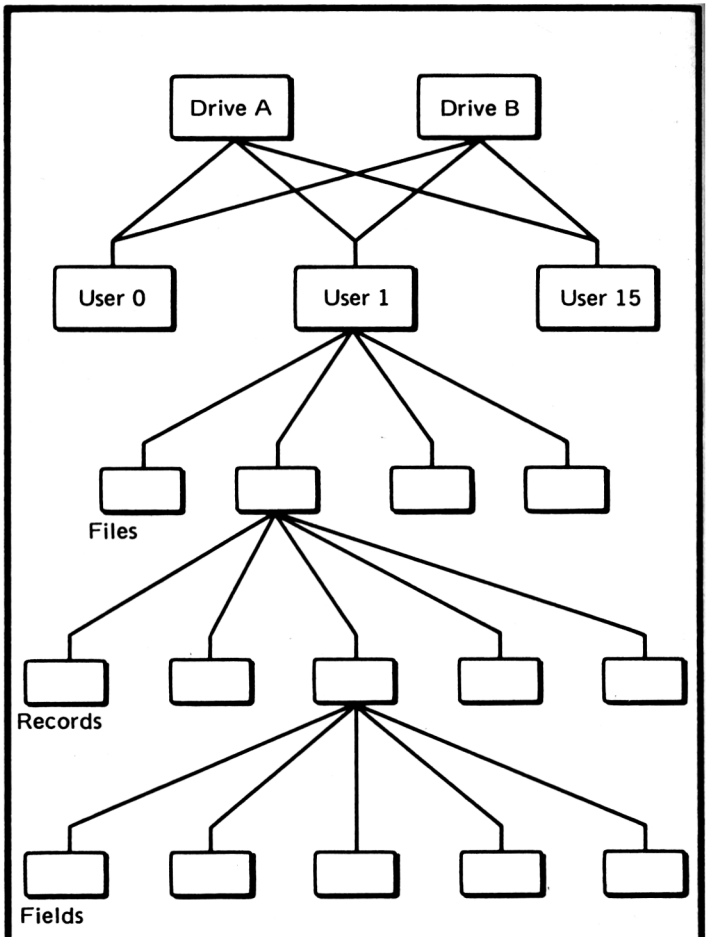


Fig 4. Heirarchical filing structure

User numbers derive from a version of CP/M called MP/M. This was a multi-user system, which meant that more than one person could use the computer at once, each with their own terminal. The user numbers were used to give each user their own area of the filing system, not accessible to other users. In fact, this operating system did not achieve any significant success.

The user numbers have, however, been retained on CP/M Plus and AMSDOS also uses them for reasons of compatibility.

When files for several user numbers are present on a disc, only files for the currently selected user number can be accessed (in fact this is not strictly true as we shall see, but it will do for now). Only these files will be displayed by CAT and |DIR. This can be a useful way of avoiding the problem of having so many filenames on the screen that you have difficulty finding the one you want. By using different user numbers for different types of file (i.e. programs, data files, text files), you can display only the type of file you want.

However, note that most commercial applications programs running under AMSDOS do not provide for switching user numbers.

It should be stressed that the user numbers do not in any way represent a physical dividing up of the disc space. They are, in effect, an addition to the filename which is used by the operating system to control file access.

WRITE#9

This command sends a 'write item' or list of write items to disc. It is similar to PRINT#9, except that items are enclosed in inverted commas and separated by commas. It is used mainly to create data files from BASIC which can be read back simply by INPUT#9. Chapter 4 covers this topic in some detail.

CP/M Commands

This section describes each of the most important CP/M O.S. commands in alphabetical order. The CP/M programs for assembly language programmers are beyond the scope of this book and are not included. Some other CP/M programs such as SUBMIT and SETKEYS are also not covered here, but are included in later chapters.

In the version of CP/M supplied with the Amstrad computers, some of the programs are standard CP/M, others are specific to the Amstrad implementation. For example, SUBMIT is standard CP/M whereas SETKEYS is specific to the Amstrad computers. All the commands in this chapter are standard CP/M.

In some cases, there is a short in-built form of a command, and a longer transient form with more facilities which must be loaded from disc when necessary. In these cases, both forms are described here.

DIR

This command produces a directory of a disc. It displays the current drive and user number, and a list of filenames with extensions. It does not give the sizes of files, or show the amount of space remaining on a disc. Any files with the SYS attribute set are not listed.

There is also a transient version of DIR which allows a number of extensions, giving fuller information about files, and other facilities. This version will be loaded and used whenever an extension to DIR is specified.

As usual in CP/M, the extension options are given after the command, enclosed in square brackets (though in some cases the closing bracket can be omitted, and abbreviations of the extension used).

In order to use the transient version of DIR, a disc containing the transient program must be in the current drive when the command is executed. It will normally be this disc for which the directory will be produced. In order to produce a directory with optional extensions from a disc which does not have the DIR program on it, you must specify a different drive in the command. For example, assuming that drive A is the current drive, and the disc currently in drive A has DIR on it, you could type, for example:-

DIR B:[FULL]

The DIR program will load from the disc currently in the drive A. If you have a second drive fitted, the directory will be produced from the disc in it. If you have only one drive, you will be prompted to 'Insert the disc for B then press any key', so that you

can change to the disc for which you require the directory.

The available options are as follows:-

ATT – displays the file attributes, which are whether a file is DIR or SYS, whether it is read and/or write protected, etc.

DATE – displays the date/time stamps on files if this facility (CP/M Plus only) has been used.

DIR – displays only the files with the DIR attribute.

DRIVE – selects the drive or drives. This option can take several forms.

DRIVE=ALL – polls and displays all the currently valid drives.

DRIVE=(A,B,...P) – displays the selected specified drives. Note however that the Amstrad computers can normally only be fitted with a maximum of two disc drives, A and B.

DRIVE=d – displays the selected drive.

EXCLUDE – displays only the files which *do not* match the specified filename, which may of course include wildcards. For example, if you have a TASWORD disc with the program files on it, and also a number of documents, you could list the document files only by using

DIR [EXCLUDE] TAS*.*

All the TASWORD program files begin with TAS, so this will ensure that these files are not displayed. This does assume, of course, that you have not begun any of the document filenames with TAS! (TASWORD is not a CP/M program but you may still want to use CP/M utilities for making backup copies of files).

FULL – gives a full directory, with filename, number of 128- byte records in the file, and all file attributes. If there is a directory label on the disc, it also shows the password protection mode and date/time stamps. The display is alphabetically sorted.

MESSAGE – displays the drives and user numbers currently being searched by DIR. Useful in conjunction with DRIVE option.

NOSORT – can be used in conjunction with FULL, where it inhibits the alphabetical sorting.

RO – displays read only files only.

RW – displays read/write files only.

SIZE – displays file sizes in Kbytes.

SYS – displays files with the SYS attribute only.
USER=ALL – displays files for all user numbers.
USER=n – displays files for selected user number only.
USER=(0,1,.....15) – displays files for selected user numbers.

It is possible to combine options by placing them within the brackets, separated by commas, for example

DIR [SIZE,USER=ALL]

When a filename is specified, it is placed after the option, for example

DIR [USER=ALL] B:*.BAS

This example lists all files on drive B having filenames with the extension .BAS.

DIRS

Similar to the built-in DIR, DIRS displays only the SYS files. There is no longer transient form of DIRS, but the facilities of the transient DIR include the ability to display SYS files.

DEVICE

This command displays the current logical device assignments and physical device names. In other words it tells you what peripherals, attached to the computer, are recognised and can be used from CP/M. It is also used to assign logical devices to physical devices attached to the computer.

DEVICE also sets the protocols and baud rates for communications, and can be used to set the current screen size, or display the size currently set.

DEVICE on its own displays the physical devices and their assignments. DEVICE NAMES lists the physical devices and gives information about them. DEVICE VALUES displays the current logical device assignments.

ERA(se)

This command erases files. Wildcards can be used, and if more than one file on the disc matches the specified filename, all the matching files will be erased. ERASE asks for confirmation before erasing each file.

If you type ERASE without a filename, a transient program loads, but the only difference in this, (in CP/M Plus), is that it prompts for a filename specification. In earlier versions of CP/M, the inbuilt version of ERASE does not allow wildcards, and you must add and extender [CONFIRM] (which may be abbreviated to [C]) to prompt for confirmation on each file.

PIP

This stands for 'Peripheral Interchange Program', and it is used mostly for selective file copying. It can also be used for sending information from one device to another, for instance from a disc to an RS232 interface. It also has many other uses, and can even be used as a very elementary editor (i.e. sending data from the keyboard to a disc file).

When copying files, wildcards may be used in the filename specification. When more than one file on the disc to be copied from matches the filename specification, all the matching files will be copied. PIP does not normally ask for confirmation before copying files (but see extensions below).

PIP is a transient program which must be loaded from a disc before use. When PIP has loaded, you will see a '*' prompt instead of the usual 'A>' or 'B>' prompts.

The general form of a PIP command is

A:filename=B:filename

This will copy a file from drive B to drive A. Note that the file you wish to copy from is specified second. This can be confusing at first, as it seems to be the wrong way round. You just have to get used to it.

When you want to copy a file from one disc to another, without changing the filename, you do not need to specify the filename in the destination specification, thus

B:=A:mytext.doc

This will make a copy on the drive B disc of the file 'mytext.doc' on the drive A disc. (If you have only a single drive, you will of course be prompted to change the disc as necessary.)

B:=A:*.*

will copy all the files on disc A to disc B. Note that I said 'all files' not 'the entire disc'. PIP will not copy the CP/M system files. To copy an entire disc, use DISCKIT3.

The main options of PIP are as follows. These extensions should be placed after the source file specification, in the usual square brackets.

A
Archive. This will copy only files which have been changed since the last time they were copied.

C
Confirm. Prompts for confirmation before copying each file.

E
Echo. Echoes the file being transferred to the console (odd things may happen if the file is not straight ASCII text).

K
Kill display of file specification on console.

L
Translates any upper case letters in the file to their lower case equivalents. Numerals, punctuation marks, and lower case letters are unaltered.

R
Read and copy SYS files.

U
Translates any lower case letters to their upper case equivalents (see L).

V
Verify. Checks to ensure that data has been copied correctly. This extension can be very important, as there is a bug in PIP which sometimes results in long binary files (which include program files and data files from programs such as DR DRAW) being corrupted. It is recommended that you always use the V extension with such files.

RENAME

This command lets you change filenames in the directory. Wildcards are permitted in both the old and the new filenames. When wildcards are used in the new name, the characters in the part of the new name represented by the wild card will be the same as the characters in the equivalent positions in the old name. However, when using wildcards, you should be careful not to arrive at a situation where you would be forming two identical filenames. The command takes the general form

RENAME newname=oldname

Like PIP, this seems somehow to be the wrong way round! There are no optional extensions.

SET

This command is used to initialise certain CP/M file facilities, such as password protection, time/date stamping of files, and labelling of discs. It can also be used to set the read-only, read-write, DIR and SYS attributes on files.

The following modes of password protection may be set.

READ The password is required to read, copy, write to, delete or rename the file. This is the maximum degree of protection.

WRITE The password is required to write to, delete or rename the file.

DELETE The password is required to delete or rename the file only.

NONE No protection. Can be used to delete a password.

If time/date stamping has been initialised on a disc, a file will automatically have the time and date recorded with it each time it is updated. For this facility to be of any use, the internal clock must be set each time CP/M is 'cold booted'. When password protection, time/date stamping or disc labelling has been initialised, the disc should not subsequently be written to by CP/M 2.2 or AMSDOS, as these operating systems do not support these facilities.

SET is also used to alter the file 'attributes'. These attributes are flags recorded with the file which control how it will be treated by, for example, DIR and PIP. The attributes are as follows:-

RO Sets the file to read-only. This protects the file from attempts to modify it. Any attempt to write to a read only file will produce an error message. The file can be copied, however, copies will not automatically be read-only.

RW Sets file to read-write. The file may be read and modified.

SYS A file with the SYS attribute will not be listed by DIR, but only by DIRS or the transient version of DIR with appropriate extensions.

DIR A file with the DIR attribute set will be listed by DIR, but not by DIRS.

ARCHIVE This attribute is used to indicate whether a file has been backed up for archival storage. It can be set either by PIP when copying or by SET. It can be reset by SET, and is automatically reset if the file is modified. By using this attribute, PIP can be used to copy only those files which have been modified since the last backup operation. See PIP.

TYPE This command displays an ASCII (text) file on the console screen. If the file contains things other than ASCII characters, strange things may happen. There are two options.

PAGE This stops output after each screenful and awaits a keypress before continuing (a prompt is produced). This is the default.

NOPAGE This gives continuous output.

Chapter 3

RULES AND REGULATIONS

When working with files on disc, the use of filenames is obligatory. These filenames have to conform to rules set by the CP/M specifications. For reasons of compatibility, AMSDOS uses the same rules.

In general, a filename consists of two parts, the name, which may be up to eight characters long, and the extension, which may be up to three characters long. The two parts are separated by a full stop.

The first part is compulsory, and you will always be required to specify a filename when opening a file. Note that this includes saving programs in BASIC or other languages. Eight characters is the maximum length for the first part, but it can be shorter (just a single character if you like). If any longer, the extra characters will be discarded (or in some cases may produce an error). If you have more than 8 characters in the first part of the name and also supply an extension, the extension will be discarded along with the extra letter(s), and in some cases, the program may add its own extension.

The most important thing is to choose a filename which will remind you of what the file contains. Always try to think of sensible filenames. The first part of the filename should be used to distinguish individual files. The real purpose of the extension is to group similar files, so that for example, all data files belonging to a database program would have different identifying filenames, but the same extension grouping them together as belonging to that program.

The extensions to filenames are optional. If you do not give one, in many cases the operating system will add one of its own. For example, BASIC programs will be automatically saved with the extension .BAS.

Both the first part and the extension of the filename may contain letters and numbers, and they may start with a number if required. They may also contain symbols such as $ and &, but not commas, the full stop, the exclamation mark, the asterisk or the

question mark. As a general rule, however, it is best just to use the letters or numbers. Neither part of the filename may contain spaces.

Although in general the extension does not force the operating system to use a file in a specific way, some extensions are used for specific purposes, and these extensions should be regarded as 'reserved', and not used otherwise. For example, CP/M command programs use the extension .COM, and this should not be used for other files. A command program must have this extension in order to be loaded and run under CP/M. Similarly, it is not a good idea to use the .BAS extension other than for BASIC programs, or the .BIN extension other than for machine code programs (or files of saved memory).

Some programs also use extensions to identify their files, and you should be careful about using these extensions for other files you may create. For example, DR LOGO uses the extension .PIC for saved screen displays, and DR DRAW uses .PIX for picture files. If you were to use these extensions for other types of file, and subsequently try to load them into these programs, you would probably cause a crash.

Many programs use the extension .TMP for a temporary file which is used by the program while it is running. Such files are normally erased automatically when the program ends. If you find a file with this extension in the directory it usually means that a program crash has occurred at some time. You can build up a succession of .TMP files with some programs if you end them by resetting the computer, or just switching off, rather than using the proper exit procedure. You should avoid doing this, as of course you are unnecessarily using up disc space.

When CP/M first opens a file, it gives it the temporary extension .$$$. This is changed to the final extension when the file is closed. Again, if you find a .$$$ file in the directory, it is probably the aftermath of a program crash. If you have a program crash with a program which uses temporary files on disc, it is worth checking whether there is a.TMP or .$$$ file on the disc afterwards. Sometimes these files can be renamed with a proper extension, and some or all of the data recovered.

Some programs do not use the extension at all, TASWORD 6128 is a program of this type. In these cases you can either ignore

extensions and save yourself some typing, or use your own extensions to identify types of file. For example, I use the extension .MS for book manuscript files, .LTR for letter files, and .DOC for other documents.

With programs which automatically add their own extensions, and which require these extensions to be present in order to load a file, you do not normally need to type the extension, either when saving or loading data. In fact, if you do type an extension it will normally be ignored.

With a program like TASWORD, which does not use extensions for its own purposes, you will normally need to type the entire filename including extension if used.

When saving a file, if the filename you specify is the same as one already in the directory, the existing file is not overwritten. Instead, it is renamed with the extension.BAK, thus giving an automatic one-stage backup. If you do a second save using the same filename, the .BAK file is erased, the second version becomes the .BAK version, and the third version is saved with the proper extension (and so on with subsequent saves).

No warning is given when you are about to save a file with an existing filename. Take care not to use a name you have already used for a *totally different* file, as you could lose it this way. Whenever possible, and it usually is, check the disc directory before saving.

Wildcards

Wildcards have already been mentioned, but they are worth looking at in more detail.

The idea of wildcards is that they enable a file to be referred to without typing, and in some cases without knowing, the full filename. They also, in some instances, allow reference to more than one file at a time. For example, they allow PIP to copy a number of files with a single command.

There are two wildcards, the asterisk (*) which can replace any number of characters, and the question mark, which can replace any single character. Both of these may be used in either the first part or the extension to the filename. It was mentioned earlier that the extension should be used to group files together. This is useful when using wildcards. For example, if you use the

extension .DAT for all your database files, you could then copy all these in one go by using PIP B:=A:*.DAT. This is an example of using the asterisk to match any filename up to the maximum 8 characters.

You may also have a number of files which belong together, and have the same first part to the filename, but different extensions. For example, you might have a program which is part BASIC, called MYPROG.BAS, part machine code, called MYPROG.BIN, with an overlay part MYPROG.OVL, and a data file MYPROG.DAT. You could copy all these with PIP using PIP B:=A:MYPROG.*.

The question mark wildcard is more useful when you want to be more selective. For example, my naming system for book text files (I keep chapters as separate files) is to use filenames of the form CHAPnii.MS, where n is the chapter number, and ii represents 2 letters which identify the book. For this book they are AD (no prizes for guessing from whence they derive). If I wanted to copy all the files for this book, but not for any other book, and not any .BAK versions, I could use PIP B:=A:CHAP?AD.MS. In this example, the question mark can match with any chapter number. With ten or more chapters, the chapters with 2 digits would not be matched, however, and would need a separate command. The asterisk could be used, but would also match with, for example CHAP1CAD.MS, which might cause problems.

Wildcards are extremely useful, but should always be used with care and forethought, especially if you are erasing files.

Wildcards cannot be used when specifying filenames for use by BASIC when loading, saving, opening and closing files, that is, with the keywords SAVE, LOAD, CHAIN, MERGE, CHAIN MERGE, PRINT#9, WRITE#9, OPENIN, OPENOUT, INPUT#9 etc.

Something which causes a lot of people difficulty is the use of wildcards in the destination filename when copying or renaming files. The rule is that, when wildcards are used in this way, the characters they represent will be the same in the destination filename as they are in the source filename.

As a practical example, suppose you have been using DR LOGO, and have saved a number of picture screens. In several cases, you have saved two versions using the same filename, so you have .PIC and .BAK files with the same filename. You now

want to refer back to the earlier version, and in order to load them you must copy them to another disc changing .BAK to .PIC. You can do this without copying any of the .PIC files on the original disc with

PIP B:*.PIC=A:*.BAK

Remember, it is the destination filename which is given first. This command will copy any file on disc A with the extension BAK to a file on disc B with the same filename but with the extension .PIC. It will not copy files on disc A with the extension .PIC, or indeed with any extension other than .BAK.

You can also use wildcards to rename files selectively changing parts of the name only. For instance, you may have a number of files for a business with filenames like SALESCUR.FIG, CUSTCUR.DAT, SUPPLCUR.DAT, where the CUR in the name indicates current files. At the end of a year, you may wish to defer these files and start new current files with these names. You could rename these files with a date (year) instead of CUR with

RENAME B:*85.*=A:*CUR.*

The files above will become SALES85.FIG, CUST85.DAT and SUPPL85.DAT.

Filespec

In the last chapter it was said, when discussing user numbers, that only files in the current user area could be accessed. It was also said that this was not strictly true. In fact, it is possible to specify a user number other than the current one, and also a drive other than the default drive, when specifying a filename. You can specify either of these, or both. A filename with a user number and/or drive specifier is referred to as a 'filespec'.

You can give a full filespec for CP/M console commands, but some programs (both under CP/M and AMSDOS) may impose restrictions. You cannot specify a drive or user number with Amstrad BASIC or AMSDOS external commands, with the exception of |REN.

In fact, we have already seen how the drive may be specified, in the PIP and RENAME examples above. The drive specifier is given ahead of the filename, separated from it by a colon, as in this example.

ERA B:ADDR.TMP

Specifying the drive in a filespec in this way does not change the default drive. Of course, under AMSDOS and CP/M 2.2 you can only specify a drive other than drive A if you have a second disc drive fitted (or an add-on silicon disc).

The user number can be specified in much the same way, by giving it immediately after the drive specifier, as in this example.

```
ERA B1:ADDR.TMP
```

To copy a file from one user area to another, you have to specify the user area after the destination filename like this.

```
PIP LETTERS.DOC [G2]=CRSP.DOC
```

This will copy the file CRSP.DOC from the default drive and current user number to the default drive and user number 2. Note that the G in the square brackets is *not* a drive specifier. It simply indicates that the number which follows is the user area into which the file is to be copied. If you need to specify a different drive, you do this ahead of the filename as usual, for example

```
PIP B:LETTERS.DOC[G4]=A:CRSP.DOC[G2]
```

This also shows how PIP requires the source user number to be specified in the same way as the destination user number. This example copies the file CRSP.DOC from user area 2 on drive A to the file LETTERS.DOC in user area 4 on drive B.

Both these methods will result in two copies of the data being produced. If it is required simply to transfer a file from one user number to another on the same disc, this can be done with the AMSDOS |REN command, as in this example.

```
|REN,"4:CRSP.DOC"="2:CRSP.DOC"
```

Remember that filenames must be enclosed in inverted commas in AMSDOS commands.

An important point to remember is that when you change drive, either temporarily in a filespec or by changing the default drive, the user number remains the same. Therefore, if you are in user area 4 on drive A, and you change to drive B, you will be in user area 4 on drive B. Equally, changing user area does not change the drive.

Chapter 4

FILING FROM BASIC

Though you can buy many database programs for Amstrad computers, writing your own programs for filing purposes is not difficult, and in some cases it can take no longer to write a simple BASIC program for a task than it takes to configure some of the more complex database programs.

There are three stages in creating a file on disc. Firstly, you have to open the file. At this stage, the computer will check that there is a formatted disc in the drive, and it will create an entry for the file in the directory. After this, you must not change the disc in the drive until the file has been closed. It also reserves some memory space for use by the filing system.

The second stage is to write the data to the disc. This is done with the BASIC keywords PRINT and WRITE. These normally direct output to the screen, but output can be directed to the disc by using the channel identifier #9, i.e. PRINT#9 or WRITE#9. The keyword WRITE is generally the most useful for data files.

The program output is not sent to the disc as soon as it is produced. Instead it is stored in memory until there is enough to fill a block on the disc, and then the whole block is written out. This is called 'buffering', and the area of memory used to temporarily store the data is the 'buffer'.

When the program finishes producing output, it may well be that the buffer is not full, and the contents will therefore not be automatically written to disc. This brings us to the third stage. Closing the file ensures that all the remaining data in the buffer is stored on the disc, that the directory entry is complete, and that a special marker is written onto the disc to indicate the end of the file. This marker can be used when reading the file back. Failing to close a file opened for writing is a serious error, as it will almost certainly mean that some data will not be recorded on the disc. The end of file marker will be missing, and the directory entry may be incorrect, and these two things can mean that the file cannot be read.

It follows from this that you should never remove a disc from

a drive while it has open files on it. If you change a disc while there are open files on that drive, they will be abandoned by the operating system, and some data loss will almost certainly occur.

Having created your file you will want to read it back. Again there are three stages to this.

First step again is to open the file. At this stage the computer checks that the file is on the disc in the default or specified drive, and reserves some buffer space in memory. In fact, the first block of the file is read into the buffer.

The data is read by the BASIC program with INPUT#9 or LINE INPUT#9 statements. These read the data from the file in much the way that INPUT reads it from the keyboard. As the buffer is emptied by the program, further blocks are read into it from the disc. If you try to go on reading from the file after you have reached the end of it, an error will occur. This can be avoided by testing for the end of file marker with the function EOF.

When you have finished with the file, it must again be closed. This releases the buffer space. Closing a file opened for reading does not have any effect on the file on the disc. In fact, all the read operations leave the disc file unaltered. It follows from this that failing to close a file opened for reading is not as serious as failing to close a file opened for writing. However, it does mean that the buffer memory space is still reserved, and you cannot open another file until the first one is closed.

Under AMSDOS, you can only have one file open for writing at a time, as there is only the one channel for the disc drive, channel nine. You can also only have one file open for reading at a time. However, you can have one file open for reading and another open for writing simultaneously. This allows files to be modified using a single program in a read/modify/write cycle. You have to use two filenames for such an operation. You cannot have the same file open for reading and writing.

Listings 1 and 2 are simple programs to create, and then read back, a file on disc. The file in this case is a simple cash account record. This allows you to record the date of each transaction, what the transaction is, whether it is debit or credit, and the amount. This introduces the concept of fields and records, which is how data in a database is structured.

In this example, the date, the item, the debit/credit, and the

```
10 REM listing 1
20 ON BREAK GOSUB 120
30 OPENOUT "account.dat"
40 CLS
50 WHILE 1
60 INPUT "Date ";date$
70 INPUT "Item ";item$
80 INPUT "debit/credit ";dc$
90 INPUT "Amount £",amount
100 WRITE#9,date$,item$,LEFT$(dc
$,1),amount
110 WEND
120 CLOSEOUT
130 END
```

```
10 REM listing 2
20 credit=0:debit=0
30 MODE 2
40 OPENIN "account.dat"
50 WHILE NOT EOF
60 INPUT#9,date$,item$,dc$,amoun
t
70 PRINT TAB(1)date$;TAB(12)item
$;TAB(60)dc$;TAB(65)USING "££###
###.##";amount
80 IF LOWER$(dc$)="c" THEN credi
t=credit+amount
90 IF LOWER$(dc$)="d" THEN debit
=debit+amount
100 WEND
110 PRINT "Credit total= ";USING
"££#######.##";credit
120 PRINT "Debit total=  ";USING
"££#######.##";debit
130 PRINT "Balance       ";USING
"££#######.##";credit-debit
140 END
```

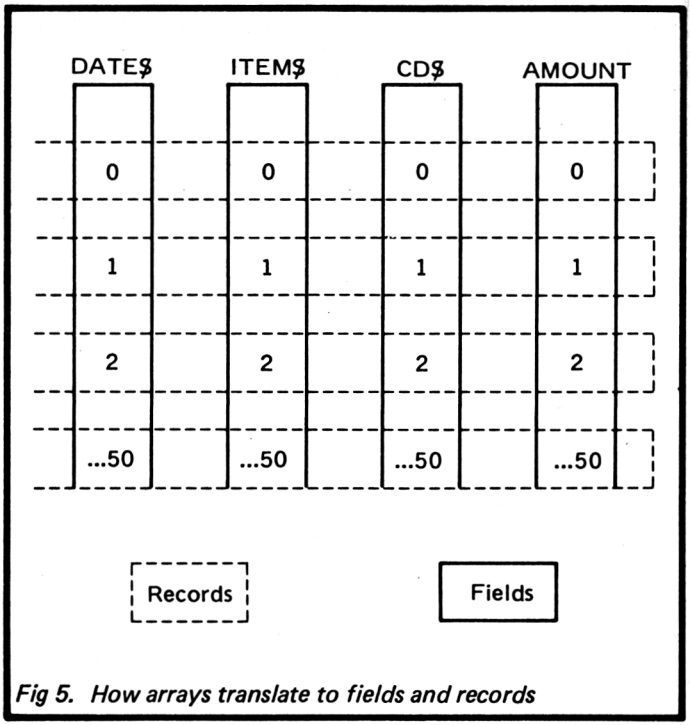


Fig 5. How arrays translate to fields and records

amount are the fields. Each collection of date, item, debit/credit, and amount is a record. Every record in a database has the same fields. When writing a database program, you must make sure you include all the fields you will require (taking into account future requirements) as it is usually very difficult to add fields once a file has been started.

In listing 1, the file is opened, and the program then goes into a loop, which allows you to type in the entries one after another. Note that the "OPENOUT" statement in line 30 is the only place where the filename is used in the program. In this simple program, the filename is given as a string literal. In practical programs, it would be more usual to obtain a filename as a string variable from an input statement.

The disc drive may run from time to time as the buffer is filled, though you have to type in quite a lot of data for this to happen, as the buffer size used is 2k (over 2000 characters). When you have made all your entries, you exit from the loop by pressing the escape key twice. The file is then closed. The disc drive will run at this stage, as the remaining data is written to the disc. Use CAT after the program has finished to check that the file is indeed on the disc.

Listing 2 can then be used to read back the file. The 80 column mode is selected to allow a neat display. Once again, the filename is included in line 40 as a string literal. Note the use of WHILE NOT EOF in line 50 to create the loop to read the entries back. This is done with the INPUT statement.

The entries are printed on the screen as they are read in. The use of TABs in the print statement allows them to be displayed neatly in columns. Each record is printed on one line, the columns of the display being the fields. The PRINT USING feature of Locomotive BASIC is also used in line 70 to give a neat display in monetary form. Also, the program calculates the debit and credit totals, and the balance, and prints these on the screen after the last entry. The credit/debit field is used to determine what each entry is (lines 80 and 90).

These programs are, of course, of little practical use. For a start, it is difficult to correct a mistake, and once closed, a file cannot be extended. For any kind of database program to be of real use, it must be possible to edit the data.

With BASIC programs running under AMSDOS, it is not possible to edit data actually on the disc. To do this, you need something called 'random access filing', where records on the disc may be read and written individually in any order. With random access filing you can open files for reading and writing simultaneously. AMSDOS has only serial access, which means that you must start reading the file from the first record, and read them in serial order until you come to the end of the file. To start again, you must close the file and then re-open it. A file can be open only for reading or for writing, not both at the same time.

In order therefore to edit a file, the whole file must be read into memory. It can then be altered in memory, and then the whole file written out to disc again, replacing the original.

When editing the file, it would normally be held in memory in BASIC arrays. The form of the array (i.e. the number of dimensions and number of elements in each dimension) will echo the structure of the file (i.e. the records). BASIC arrays are a random access form of storage, so editing of the data, and method of data associated processes such as searching and sorting, are straightforward.

When writing a program to edit data in an array, it makes sense to include the initial data entry and file creation in this program also, so that one program can include all the necessary processes in database management. Listing 3 is an example of such a program, doing the same job as listings 1 and 2, plus editing and other facilities. This program shows the general form of BASIC database programs, with a menu system for selecting the required operations, and subroutines to perform each of the tasks.

```
10 REM Listing 3
20 MODE 1
30 DIM date$(50),item$(50),cd$(5
0),amount(50)
40 CLS:f$="≠≠ \           \  \

     \  \ \  ££≠≠≠≠≠≠.≠≠"
50 LOCATE 14,4:PRINT "ACCOUNT FI
LER"
60 LOCATE 10,8:PRINT "1 - Enter
Account data"
70 LOCATE 10,10:PRINT"2 - Load f
ile"
80 LOCATE 10,12:PRINT "3 - Save
file"
90 LOCATE 10,14:PRINT "4 - Calcu
late totals"
100 LOCATE 10,16:PRINT "5 - Disp
lay entries"
110 LOCATE 10,18:PRINT "6 - Alte
r/amend an entry"
```

```
120 LOCATE 10,20:PRINT "Press nu
mber key"
130 choice$=INKEY$:choice=VAL(ch
oice$)
140 ON choice GOSUB 190,340,470,
590,740,850
150 IF choice$="" THEN 130 ELSE
40
160 END
170 '
180 '
190 CLS' entry routine
200 IF n<>0 THEN GOSUB 1200
210 INPUT "Date (RETURN to exit)
";date$
220 IF date$="" THEN RETURN:REM
Exit routine on null entry
230 INPUT "Item";item$
240 IF item$="" THEN 230
250 INPUT "Credit/Debit";cd$
260 cd$=LOWER$(cd$):IF cd$<>"c"
AND cd$<>"d" THEN 250
270 INPUT "Amount";amount
280 PRINT
290 date$(n)=date$:item$(n)=item
$:cd$(n)=cd$:amount(n)=amount
300 n=n+1:IF n>50 THEN PRINT "ME
MORY FULL!" GOSUB 1140:RETURN
310 GOTO 210
320 '
330 '
340 CLS:CAT: ' file recovery rou
tine
350 LOCATE 1,20:INPUT "Please en
```

```
ter filename";name$
360 IF name$="" THEN RETURN
370 OPENIN name$
380 n=0
390 WHILE NOT EOF
400 INPUT#9,date$(n),item$(n),cd
$(n),amount(n)
410 n=n+1
420 WEND
430 CLOSEIN
440 RETURN
450 '
460 '
470 CLS:CAT 'filing routine
480 IF n=0 THEN GOSUB 1310:RETUR
N
490 LOCATE 1,20:INPUT "Please en
ter filename";name$
500 IF name$="" THEN RETURN
510 OPENOUT name$
520 FOR entries=0 TO n-1
530 WRITE#9,date$(entries),item$
(entries),cd$(entries),amount(en
tries)
540 NEXT entries
550 CLOSEOUT
560 RETURN
570 '
580 '
590 LOCATE 10,20:PRINT SPACE$(20
):LOCATE 10,20:PRINT "Calculatin
g..."
600 IF n=0 THEN GOSUB 1310:RETUR
N
```

```
610 credam=0:debam=0
620 FOR entries=0 TO n
630 IF cd$(entries)="c" THEN cre
dam=credam+amount(entries)
640 IF cd$(entries)="d" THEN deb
am=debam+amount(entries)
650 NEXT entries
660 LOCATE 10,20:PRINT SPACE$(20
)
670 LOCATE 5,20:PRINT "Debit ";U
SING "ff######.##";debam
680 LOCATE 5,21:PRINT "Credit";U
SING "ff######.##";credam
690 LOCATE 5,22:PRINT "Total ";U
SING "ff######.##";credam-debam
700 GOSUB 1140
710 RETURN
720 '
730 '
740 MODE 2'display entries routi
ne
750 IF n=0 THEN GOSUB 1310:MODE
1:RETURN
760 FOR d=0 TO n-1
770 PRINT USING f$;d+1;date$(d);
item$(d);cd$(d);amount(d)
780 IF (d/20)=INT(d/20) AND d<>0
 THEN GOSUB 1140:CLS
790 REM last line displays in gr
oups of 20
800 NEXT d
810 GOSUB 1140:MODE 1
820 RETURN
830 '
840 '
```

```
850 MODE 2 'alter/amend routine
860 IF n=0 THEN GOSUB 1310:MODE
1:RETURN
870 LOCATE 5,2
880 INPUT "Enter no. of item to
alter";in:in=in-1
890 IF in<0 THEN MODE 1:RETURN
900 IF in>n-1 THEN 850
910 LOCATE 1,5
920 PRINT USING f$;in+1;date$(in
);item$(in);cd$(in);amount(in)
930 LOCATE 5,7
940 INPUT "New date";nd$
950 IF nd$<>"" THEN date$(in)=nd
$
960 LOCATE 5,9
970 INPUT "New item";ni$
980 IF ni$<>"" THEN item$(in)=ni
$
990 LOCATE 5,11
1000 INPUT "New C/D";nc$:nc$=LOW
ER$(nc$)
1010 IF nc$<>"" THEN cd$(in)=nc$
1020 IF cd$(in)<>"c" AND cd$(in)
<>"d" THEN 1000
1030 LOCATE 5,13
1040 INPUT "New amount";na$
1050 IF na$<>"" THEN amount(in)=
VAL(na$)
1060 CLS:LOCATE 1,5
1070 PRINT USING f$;in;date$(in)
;item$(in);cd$(in);amount(in)
1080 LOCATE 5,7
1090 PRINT "Another alteration (
Y/N)?"
```

```
1100 k$=INKEY$:IF k$="" THEN 110
0
1110 IF UPPER$(k$)="Y" THEN 850
ELSE MODE 1:RETURN
1120 '
1130 '
1140 LOCATE 5,24:PRINT "PRESS AN
Y KEY"
1150 r$=INKEY$
1160 IF r$="" THEN 1150
1170 RETURN
1180 '
1190 '
1200 LOCATE 5,2:PRINT "Add to fi
le or"
1210 LOCATE 5,4:PRINT "start New
 file?"
1220 LOCATE 5,6:PRINT "A/N"
1230 k$=INKEY$:IF k$="" THEN 123
0
1240 IF UPPER$(k$)="N" THEN n=0
1250 LOCATE 5,8:PRINT k$
1260 GOSUB 1140
1270 CLS
1280 RETURN
1290 '
1300 '
1310 CLS 'no entries routine
1320 LOCATE 5,2:PRINT "NO ENTRIE
S IN MEMORY"
1330 GOSUB 1140
1340 RETURN
```

In this program, four separate arrays are used to hold the data, each array storing the contents of a field. The elements of the arrays represent the records. The arrays are declared in line 30, and have 51 elements (remembering that Locomotive BASIC arrays count elements from zero). This allows 51 records. (This is a purely arbitrary limit and could be considerably increased if required).

Line 40 initialises a string (f$), which is used as a formal template for PRINT USING statements. It will be very hard to type this in correctly just from the listing, so the items contained between the inverted commas are listed here.

2 hashes
1 space
backslash
10 spaces
backslash
2 spaces
backslash
40 spaces
backslash
2 spaces
backslash
1 space
backslash
2 spaces
2 pound signs
6 hashes
decimal point
2 hashes

This string is used when displaying records on the screen, and gives a neat columnar display.

Lines 50 to 120 print the menu display on the screen.

Lines 130 to 150 select the subroutines according to the key pressed.

Lines 190 to 310 are the input routine. This is quite straightforward, but note how the inputs are placed first into simple variables, and then placed in the arrays in one go in line 260. This

means the arrays are never 'out of line' if, for example, you break into the program with the escape key. A small point but sometimes helpful. The variable 'n' is used as the entry counter – a vital variable in any database program. In this example it points to the array element where the next entry will be stored, not where the last one was placed. It is important to remember this when writing the part of the program which saves the data to tape.

There is some error trapping in this routine. Empty entries are not allowed (lines 240, 260) and line 260 also converts all entries in the credit/debit field to lower case. This simplifies later operations.

Note that this routine is exited by making a null entry in the first (date) field. This is usually the cleanest way to leave an entry routine.

The next routine in order is the one to recover a file from disc, lines 340 to 440. This is similar to listing 2 in that it uses a WHILE...WEND loop, testing for the end of file, but it places the items into the arrays as they are read, rather than printing them on the screen.

Line 340 clears the screen, and then produces a catalogue of the disc on the screen. Line 350 then prompts for a filename. In this program the filename is entered into a string variable. If you just press RETURN or ENTER here, the routine is abandoned (line 360). It is always a good idea to provide an escape route like this.

Line 380 resets the variable n to zero. This means that a load operation will cause any file already in memory to be cleared. If you want the option of being able to add one file to another, you could change line 380 to GOSUB 1200. However note that there is no error trapping in this routine for trying to load in more entries than there is space for in the arrays.

In fact, this program can load in the files created by listing 1, as the file structure used by the two programs is identical. If you have tried listings 1 and 2, this may be useful to you. However, note again that if you have created a file with more than 51 records, an error will occur on loading, unless you increase the size of the arrays in line 30, and modify line 300 accordingly.

Lines 470 to 560 are the routine to save a file to disc. The first part of this (470 – 510) is almost identical to the start of the load routine. Line 480, however, is an extra to prevent saving of an empty file. A FOR...NEXT loop is used to save the entries, the terminating value for the loop being n−1, to avoid saving an empty record.

The remainder of the program consists of routines to calculate the credit and debit totals, and balance, and display them, and also to display all the entries (in blocks of 20), to allow entries to be altered, plus three service routines used by several of the other routines. This is all straightforward programming.

As you can see from this listing, the routines to save data on tape and reload it are often the easiest parts of a database program to write.

Disc Error Handling

This program contains no error handling. This means that if an error occurs an error message will be printed and the computer will return to command mode. This is generally unsatisfactory for database programs as any data entered could easily be lost. (In fact you can re-enter this program cleanly by using GOTO 40.)

Run-time errors in a program of this sort can usually be avoided by using error trapping lines of the sort already mentioned above, and this is usually easier than using ON ERROR GOTO. However, all errors which can arise from disc operations cannot be prevented in this way. For example, there is no way the program can check for a disc not being present in the drive. Some error handling is therefore desirable.

Detecting disc errors has in general to be done in two stages. Firstly, all disc errors produce error number 32. That is to say, the function ERR will return 32 if any disc error has occured. A second function, DERR, allows the particular disc error to be identified. When a disc error occurs, the operating system always produces an error message on the screen, even when ON ERROR GOTO is in force.

Some disc errors will produce the default operating system response "Ignore Retry or Cancel?". If you retry, and the fault is cleared, program execution will continue. If you choose to cancel, and there is no ON ERROR GOTO in the program, you will be dumped into command mode. If there is an ON ERROR GOTO, the program execution will be passed to this. Ignoring it will often just lead to further errors at a later stage in the program.

Other errors will just send you straight to command mode if no ON ERROR GOTO is present.

An example of the first type of error is a disc being missing. An

example of the second type is if the file you try to open (to read) is not present on the disc.

LISTING 4

```
15 ON ERROR GOTO 1360
1350 '
1360 '
1370 IF ERR=25 THEN PRINT "Wrong
 file type.":GOSUB 1140
1380 IF ERR=32 AND DERR=146 THEN
 GOTO 1430
1390 IF ERR=32 AND DERR=144 THEN
 GOTO 1500
1400 RESUME 40
1410 '
1420 '
1430 PRINT "Retry or Cancel"
1440 res$=UPPER$(INKEY$)
1450 IF res$<>"R" AND res$<>"C"
THEN 1440
1460 IF res$="R" THEN RESUME
1470 IF res$="C" THEN RESUME 40
1480 '
1490 '
1500 LOCATE 21,20:PRINT SPACE$(2
0)
1510 IF ERL=370 THEN RESUME 350
1520 IF ERL=510 THEN RESUME 490
1530 RESUME 40
```

Listing 4 is the error handling which can be added to listing 3 to deal with these errors, and also with illegal filenames.

Line 1370 deals with the problem of trying to open the wrong type of file for read operations (for example, a BASIC program file). This is not a disc error, as it can also happen with cassette files.

Line 1380 is used to detect the 'file not found' error. When this happens, lines 1430-1450 give you the options of cancelling or retrying. Cancelling takes you back to the menu (RESUME 40 in line 1470). If you choose to retry, line 1460 resumes execution by retrying the line where the error occurred. It is only sensible to retry if you change the disc in the drive for the correct one.

Detecting a bad filename is a little more complex, as an illegal filename and a disc being missing both return disc error (DERR function) no.144. A disc missing needs no programming as the operating system copes as previously mentioned. A bad filename does, however, require action.

The best way to separate a missing disc from a bad filename is by using the function ERL, which gives the line where the error occurred. In this program, if the error occurs at lines 370 or 510, it can be assumed to be a bad filename, as a disc missing would have been detected at lines 340 or 470. Lines 1510 and 1520 take you back to the lines in the program where the filename is entered. Line 1500 simply blots out the old (erroneous) filename.

When the disc missing error occurs, and you choose to cancel, this part of the program will still be executed, so line 1530 is necessary to take execution back to the menu stage.

This error handling routine cannot cope with the situation if the disc is removed between lines 340 and 370, or between lines 470 and 510, however. If there is a way of doing this, I have yet to find it!

A further point about lines 1380 and 1390. You must always test both for the ERR value 32 and the DERR value. This is because the disc error number will not be changed unless and until another disc error occurs. If you just test for the DERR value, you could keep having to deal with an old disc error, every time an error of any sort occurs.

It should also be pointed out that there seem to be some bugs in the Locomotive BASIC error handling. In particular, RESUME may not work correctly if it is used in an IF...THEN RESUME...ELSE RESUME... type of construction, or indeed if it is used in any statement but the first in a multi-statement line.

Other File Uses

As well as being used to hold records of the type demonstrated in this chapter, data files can be used to hold other types of

information which may be required by programs. For example, many games use a lot of numeric data for such things as graphics displays and user-defined characters.

Information of this sort is usually held in DATA statements and used once only, usually soon after the program is run. Such data can easily be held in a file on disc instead, and used directly from this.

This is also true of the text required by adventure games, which usually has to be read from DATA statements into arrays. Clearly, this could also be stored on disc instead.

By storing the data separately from the program in this way, more space is available for the program proper, allowing it to be longer and more complex.

WRITE and PRINT

Both WRITE#9 and PRINT#9 can be used to send data from a program to a disc file. The difference between these two are important.

PRINT sends data to the disc in virtually the same form as it sends to the screen. This means that strings are printed without enclosing quotation marks, if commas are used as separators in print lists a number of spaces will be placed between the print items, to cause printing in columns, and unless the print list is terminated by a semicolon, it will be terminated with a carriage return.

Data sent to a disc will also take the form of a print template if the PRINT USING form of statement is used.

Sending data to a disc in this way can cause problems when reading back with the INPUT statement. For example, INPUT will regard a comma in an input as a terminator when reading in from disc just as it will with keyboard input. If your data is an address file, for instance, you may well have commas in the middle of strings, as in "123, High Street".

You may know that, when typing inputs from the keyboard, you can include commas if you enclose the input in quotation marks. The same applies to input from discs.

WRITE sends data to the disc in this form: strings are enclosed in double quotes, items are separated by commas, no extra spaces are written to the disc, the last item in each WRITE list is followed by a comma, not a carriage return.

Data sent in this form is ideal for reading back with the ordinary INPUT statement, and is therefore normally the most useful for creating data files.

Printing to disc does, however, have its uses. In particular, it creates files in a form which can be read into most word processors. (Files created by WRITE can be read in, but the data will be unformatted, and the quotation marks and comma separators will also be read in). The ability to use PRINT USING to create such files can be very useful, as it allows the data to be formatted in vertical columns. This can be difficult to do with the word processor itself.

If it is required to read a file created by PRINT back into a BASIC program, this is best done with the LINE INPUT statement. Unlike the ordinary INPUT, this does not regard the comma as a terminator. It reads everything, including commas and quotation marks as input, until a carriage return is received.

It must be remembered, however, that there is a maximum length for a string of 255 characters. If you send data to a disc with PRINT and terminate each print list with a semicolon, no carriage returns will be sent, and you may well generate a file a lot longer than 255 characters. If this happens and LINE INPUT is used to read the file, input will automatically be terminated after each 255 characters. This could result in any structure of the data being distorted.

Chapter 5

PROGRAM DEVELOPMENT

As well as being a fast method of saving and loading programs, a disc drive can also be a help when writing your own programs. There are a lot of things which must be done in programs which will be found in many programs. The sections of code to do these tasks do not have to be written specially for each program. Rather, it is possible to build up a 'library' of program sections (often in the form of subroutines) on disc, which can be added into a program as required.

Building up a library in this way is possible when using cassettes, but finding the right cassette and the right place on the tape is time-consuming, which rather takes the gilt off the gingerbread. With discs, the process is much simpler, and the task of building up the library really worthwhile.

The command to load a part of a program into memory, merging it with the program already there, is MERGE. This is followed by the filename of the program segment, in inverted commas of course.

When you merge in a section of program, if any lines in the new segment have the same line numbers as lines in the existing program, those lines in the existing program will be overwritten. When adding library routines, you will not normally want this to happen. The best answer to this is to save all the library routines with high line numbers, and immediately after MERGEing use the renumber command to change the numbers to ones in sequence with the rest of the program.

The Locomotive BASIC used in the Amstrad computers (except the PCW series) allows line numbers up to 65535. All library line numbers can conveniently be started at 60000. This allows plenty of scope for even the longest routines, whilst also allowing plenty of scope for long programs in memory.

When writing long programs, it is the practice of many programmers to start the subroutines with lines numbered at intervals of 1000. That is, the first line of the first subroutine will be at 1000, the first line of the second subroutine to be written will be 2000,

and so on. This makes it easier to remember where a routine begins if it has to be listed on the screen for editing.

The same thing can be done when merging in subroutines, as the RENUM command allows the new and old line numbers to be specified (as well as the increment). For example, suppose you have just merged in a section of program, with line numbers starting at 60000 as suggested, and you want to renumber it so that the line numbers start at 9000. The renumber command takes the form

RENUM new line number, old line number, increment

so in this example you would type

RENUM 9000,60000

The increment may be omitted, and normally will be, as the default of 10 is ideal for most programming.

The main thing to remember about the renumbering command is that it does not allow the order of lines within a program to be altered. This means, in the example above, that there must be no lines in the program with numbers between 9000 and 60000 (inclusive of 9000). If there are, the renumbering command will fail.

There are some 'side-effects' of the MERGE command which you should be aware of, though they are normally not significant when building up programs from library routines. Firstly, all variables and arrays are forgotten, just as happens if you type in a new line from the keyboard. All user functions are also forgotten, and any DEFINT, DEFREAL and DEFSTR settings are reset.

Any open files are abandoned, and any buffer contents is lost, and the ON ERROR GOTO function is turned off. An implicit RESTORE is performed. BASIC will always return to direct mode after performing a MERGE, even if the command is in a program line, and the stack of subroutine/loop return addresses is reset.

You cannot merge a protected BASIC program.

Variables and Passing Parameters

If you intend to make extensive use of library routines to build up programs, you need to have a consistent policy for variable

names, to avoid having to do a lot of editing of the routines after merging. If you are careful to choose meaningful variable names, and always use the same names for the same purpose, you should have to do very little editing.

An associated problem is that of passing parameters to subroutines. What does this mean? Well, if for example, you have a library routine to draw a circle, you will need to tell the subroutine where on the screen the circle should be drawn, and how big it should be (otherwise you would need separate routines for each size and position of circle). These are the parameters. In Locomotive BASIC, the only way of passing parameters between the calling program and subroutines is by way of global variables.

In order to draw a circle, you would have to set variables to the co-ordinates of the required position on the screen, and to the required radius, and then call the subroutine. The subroutine would then use these variables in drawing the circle.

Any type of variable can be used in this way, including elements of arrays, and string variables where appropriate.

When a variable is to be used to pass parameters to a subroutine, it is as well to reserve it for this purpose, and not to use it for anything else. It is not necessary to have separate variables for each subroutine however, and indeed this could be difficult when using library routines. It is a better idea to have reserved variable names which you use for particular purposes in all routines which need them and in all programs.

Locomotive BASIC allows long variable names, and it is normally a good idea to make full use of this to give descriptive names to variables. This can make programs much easier to follow. However, with the variables used for parameter passing, it can be a good idea to use more cryptic one or two letter names. As these names will be used in the same way in many programs, they will be easily remembered, and the contrast between the short names and the long names will make the parameter-passing variables easy to spot.

As examples of this, you could use TX% and TY% for text co-ordinates (as used in LOCATE, for example), GX% and GY% for graphics co-ordinates, R% for radii, and so on.

The remainder of this section is given over to a number of examples of library subroutines which do useful common jobs, and

which you could use as the start of your own library, and incorporate into your own programs.

LISTING 5

```
60000 curx%=POS(≠0):cury%=VPOS(≠
0)
60010 LOCATE 5,24
60020 PRINT "Press any key to co
ntinue"
60030 r$=INKEY$:IF r$="" THEN 60
030
60040 LOCATE 5,24
60050 PRINT SPACE$(25)
60060 LOCATE curx%,cury%
60070 RETURN
```

Listing 5 – Press Any Key
In many programs you will want a routine which will hold up program execution until the user presses a key to continue. It is a good idea to have such a routine in your library. This example is a 'clean' routine, in that once it has finished executing it leaves everything as it found it. Line 60000 stores the position of the cursor at the time the routine is called, and line 60060 restores it just before the routine ends. Line 60050 erases the "Press any key..." message from the screen. It is a good idea to write the simpler library routines in this way. It is not usually possible with more complex operations, however. This routine requires no parameters.

Listing 6 – String Editor
This routine prints a string on the screen and then allows it to be edited. The string must be on one line only, i.e. it must not 'wrap' onto the next line. The cursor key may be moved along the line, characters in the string may be overwritten, and the delete and clear keys may be used in the normal way. There is no provision for inserting characters.

LISTING 6

```
60000 LOCATE x%,y%
60010 PRINT in$
60020 curcol=x%:LOCATE curcol,y%
60030 limit=x%+length:start=x%
60040 k$=""
60050 WHILE k$<>CHR$(13)
60060 CURSOR 1,1
60070 k$=INKEY$:IF k$="" THEN 60
070
60080 code=ASC(k$)
60090 IF code=242 THEN curcol=cu
rcol-1-(curcol=x%)
60100 IF code=243 THEN curcol=cu
rcol+1+(curcol=limit)
60110 IF code=127 THEN curcol=cu
rcol-1-(curcol=x%):LOCATE curcol
,y%:PRINT" ";
60120 IF code=16 THEN PRINT " ";
60130 IF code>31 AND code<127 TH
EN PRINT k$;:curcol=curcol+1+(cu
rcol=limit)
60140 IF curcol>limit THEN curco
l=curcol-1
60150 LOCATE curcol,y%
60160 WEND
60165 CURSOR 0,0
60170 ret$=""
60180 WHILE start<limit
60190 LOCATE start,y%
60200 ret$=ret$+COPYCHR$(≠0)
60210 start=start+1
60220 WEND
60230 RETURN
```

This routine needs several parameters. Firstly, x% and y% are the co-ordinates on the screen where the string (first character) will be printed. The variable 'length' is the maximum length of the string. The cursor can only be moved by this number of characters, and only this number of characters will be returned in the edited string. The string to be edited must be assigned to in$, and ret$ is the edited string which is returned by the routine.

The main 'works' of this routine are really in two parts. Lines 60050 to 60160 allow the string to be edited. In fact, they only allow the printing on the screen to be changed – this routine leaves in$ unchanged. The second part, lines 60180 to 60220, read the edited string on the screen into ret$, using the function COPYCHR$.

Note lines 60060 and 60165, which turn the cursor on during the editing, and off during the reading.

You would call this routine with a line like

```
x%=15:y%=12:in$=name$:length= 20:gosub 60000
```

and follow it with a line like

```
name$=ret$
```

LISTING 7

```
60000 DEG
60010 FOR p=0 TO 360 STEP 100/d
60020 f=m*d*COS(p)
60030 x1=x+f*SIN(e)+d*SIN(p+e)
60040 y1=y+f*COS(e)+d*COS(p+e)
60050 IF p=0 THEN MOVE x1,y1
60060 DRAW x1,y1,c
60070 NEXT p
60080 RETURN
```

Listing 7 – Circle Drawing

This is an advanced circle drawing subroutine, which can not only draw circles, but also ellipses with a given degree of stretch, and

at any angle. It follows that several parameters are needed. These are d, which is actually the radius of the circle (or minor radius of the ellipse), m, which is the modifier which controls the degree of ellipticality, x and y, the co-ordinates of the centre of the circle/ellipse, e, which is the angle of the ellipse, and c, which controls the logical colour of the line.

If m is zero, a circle will be drawn. If m is 1, the major radius will be twice the minor radius, if it is 2, four times.

I have included a line in this routine to make it work in degrees. If you prefer radians, omit line 60000. As it stands, if e is zero, the major radius of the ellipse will be vertical, if 90, horizontal.

You would call this routine with a line like

```
d=200:x=320:y200:m=0.7:e=45:c=1:gosub 60000
```

LISTING 8

```
60000 READ x,y
60010 MOVE x,y
60020 WHILE x>0
60030 DRAW x,y
60040 READ x,y
60050 WEND
60060 RETURN
```

Listing 8 – Line Drawing

This routine demonstrates how DATA statements can be used to pass parameters to subroutines when a lot of data is necessary. The idea of this routine is that it will draw any outline shape on the screen from co-ordinates stored in DATA statements. The list of co-ordinates is terminated by a negative value for the x co-ordinate (a dummy y co-ordinate is also necessary or an OUT OF DATA error could occur). The routine is quite simple, but note that lines 60030 and 60040 are *not* in the wrong order. This is necessary for the WHILE condition to end the loop without line being drawn to the terminating co-ordinates in the DATA statements.

Before calling this routine, you would normally use a

RESTORE to set the data pointer to the line where the co-ordinates for the figure to be drawn begin.

Program Overlays

As we saw in Chapter 4, there sometimes has to be a compromise between the complexity of a program and the amount of data it is capable of handling. Some desirable features may have to be left out in order to allow bigger arrays to be dimensioned.

However, it is a fact that at any one time only a small part of the program in memory is actually being used. The rest is occupying memory space unnecessarily. Indeed on some runs, parts of the program may not be used at all. For example, with a database program, if you are creating a new file you will not use the routines to load in an old file. If parts of the program were only loaded into memory when needed, programs could be made more sophisticated, whilst at the same time able to handle even bigger data files.

This is possible with Locomotive BASIC by using CHAIN MERGE. This is similar to MERGE in that it allows program files to be merged with the program in memory, but it is different in that it does not clear the variables (including arrays), and the program does not return to direct mode after a CHAIN MERGE. Instead, it will be run either from the beginning or from a line number in the CHAIN MERGE command.

CHAIN MERGE does have some of the side effects of MERGE. It clears the stack of return addresses, so you cannot CHAIN MERGE from within a subroutine or a loop. All user functions are forgotten, all open files are abandoned, ON ERROR GOTO is turned off and an implicit RESTORE is performed. You cannot CHAIN MERGE a protected program.

Loading in parts of programs as they are needed like this is called using 'overlays', as the parts of the program overlay unwanted parts as they are loaded.

This type of programming can be thought of as an extension of the use of subroutines. With subroutines, you have a main program which calls the subroutines as desired. With overlays, you have a main program which loads in and runs the overlays as required. However, the two techniques are not mutually exclusive, and the main program in an overlay system may well

include a number of subroutines which can be called either by the main program or by the overlays.

In database programs using overlays, the main program may well take the form of a 'menu', as in subroutine based programs. The user selects options from this menu, and this causes appropriate parts of the program to be loaded and run. In the ultimate form of this, the menu section may be written as a separate program which is completely overwriten by each of the overlays, and then loaded back when each overlay program finishes. In this case, it is possible to use CHAIN rather than CHAIN MERGE.

Whenever overlays are used, you do need to allow for the fact that some overlays will be bigger than others. If you have large arrays dimensioned, you may have the memory very nearly full. If you then try to load an overlay which increases the overall size of the program, you may well run out of memory. This should not cause the arrays to be lost, but will prevent the overlay from being completely loaded. If this should happen, the program in memory may be a corrupted mixture of old and new, and it may be difficult to save the data, even though it should be intact in memory.

The best solution to this is to make sure that the sections of program initially loaded are larger than any subsequent overlays. There is provision in the CHAIN MERGE command to specify a range of lines which are to be deleted when the new section is loaded, and this can be used to ensure that all unwanted lines are cleared from memory.

In practical programs, it is often best to use a mixture of CHAINING programs to do specific jobs, and CHAIN MERGEing overlays into some of them to increase versatility without increasing overall size. Listings 9 to 14 are a set of programs and overlays which work together. They form a card-index type database program. This is meant to be a demonstration of the use of overlays rather than a fully finished practical program, but is nevertheless quite usable.

There are four main programs, and one of them uses overlays. The first program, listing 9, is the first to be run. It offers the options of either starting a new file or reloading an old one. If the option to start a new file is chosen, it first asks how many fields are required, dimensions the array for storage, and then prompts for a name for each field (lines 200-340). These names are stored in the zero elements of the array.

```
10 REM listing 9
20 REM Overlay Demo
30 REM Startup program
40 REM save as "DATABASE.BAS"
50 MODE 1
60 LOCATE 16,5
70 PRINT "DATABASE"
80 LOCATE 10,10
90 PRINT "(S)tart new file or"
100 LOCATE 10,12
110 PRINT "(L)oad an existing fi
le"
120 LOCATE 10,16
130 PRINT "press letter key"
140 LOCATE 10,18:CURSOR 1,1
150 k$=INKEY$:IF k$="" THEN 150
160 k$=LOWER$(k$):PRINT UPPER$(k
$)
170 IF k$="l" THEN CHAIN "LOADER
.BAS"
180 IF k$<>"s" THEN 140
190 REM start of file creation
200 CLS
210 LOCATE 2,5
220 PRINT "Please enter no. of f
ields required"
230 LOCATE 2,7
240 PRINT "(maximum 5)
250 LOCATE 2,9
260 PRINT " "
270 LOCATE 2,9
280 INPUT fields%
290 IF fields%<1 OR fields%>5 TH
EN 250
300 DIM store$(fields%,100)
310 FOR names=1 TO fields%
```

```
320 LOCATE 2,10+names
330 PRINT "Name for field ";name
s;:INPUT store$(names,0)
340 NEXT names
350 FOR field=1 TO fields%
360 FOR element=1 TO 100
370 store$(field,element)=STRING
$(60,32)
380 NEXT element
390 NEXT field
400 LOCATE 2,24
410 PRINT "Loading main program"
420 CHAIN "DATAMAIN.BAS"
430 END
```

This program uses a fixed field length of 60 characters, and lines 350 to 390 initialise the array elements by filling them with spaces. It is a good idea to fill arrays like this when using overlay techniques, because if you are going to run out of memory space, it happens straight away and not when you have typed in a lot of data. Initialising arrays also helps to reduce the amount of 'garbage' produced when editing or sorting the data in the array. Line 420 chains the next part of the program, listing 10.

If, at line 130, you opt to reload an existing file, listing 14 is chained. This reads in the required number of fields from the file, dimensions and initialises the array like listing 9, and then reads in the field names and data. This program then also chains listing 10. Note, however, an important difference between line 420, listing 9, and line 280, listing 14. In listing 9, listing 10 is run from the beginning. In listing 14, it is run from line 60. This is to avoid having the record counting variable (n) reset to 1 in line 50 of listing 10. (Listing 14 reads the value of n in from the disc file).

Other aspects of listing 14 will be discussed when we come to the saving program.

Listing 10 is the main progam, and is the part which uses the overlays.

```
10 REM Listing 10
20 REM database main program
30 REM save as "DATAMAIN.BAS"
40 MODE 2
50 n=1:current=1
60 CHAIN MERGE "screen.ovl",70
70 GOSUB 1000
80 CHAIN MERGE "add.ovl",90,DELE
TE 1000-1140
90 WINDOW#1,40,80,5,15
100 LOCATE 2,8:CURSOR 1,1
110 k$=INKEY$:IF k$="" THEN 110
120 k$=LOWER$(k$):CURSOR 0,0
130 IF k$="a" THEN GOSUB 1000
140 IF k$="b" THEN current=curre
nt-1:GOSUB 2000
150 IF k$="f" THEN current=1:GOS
UB 2000
160 IF k$="l" THEN current=n-1:G
OSUB 2000
170 IF k$="n" THEN current=curre
nt+1:GOSUB 2000
180 IF k$="e" THEN GOSUB 3000
190 IF k$="d" THEN GOSUB 5000
200 IF k$="s" THEN CHAIN "saver.
bas"
210 IF k$="r" THEN GOSUB 6000
220 GOTO 100
2000 IF current>n THEN current=n
2010 IF current<1 THEN current=1
2020 FOR lines=1 TO fields%
2030 LOCATE 20,14+lines
2040 PRINT store$(lines,current)
2050 NEXT lines
2060 RETURN
```

```
3000 PRINT#1,"Make any required
changes then"
3010 PRINT#1,"press RETURN for n
ext field,"
3020 PRINT#1,"or just RETURN to
leave unchanged"
3030 FOR lines=1 TO fields%
3040 x%=20:y%=14+lines:in$=store
$(lines,current)
3050 length=60:GOSUB 4000
3060 store$(lines,current)=ret$
3070 NEXT lines
3080 CLS#1
3090 RETURN
4000 LOCATE x%,y%
4010 PRINT in$
4020 curcol=x%:LOCATE curcol,y%
4030 limit=x%+length:start=x%
4040 k$=""
4050 WHILE k$<>CHR$(13)
4060 CURSOR 1,1
4070 k$=INKEY$:IF k$="" THEN 407
0
4080 code=ASC(k$)
4090 IF code=242 THEN curcol=cur
col-1-(curcol=x%)
4100 IF code=243 THEN curcol=cur
col+1+(curcol=limit)
4110 IF code=127 THEN curcol=cur
col-1-(curcol=x%):LOCATE curcol,
y%:PRINT" ";
4120 IF code=16 THEN PRINT " ";
4130 IF code>31 AND code<127 THE
N PRINT k$;:curcol=curcol+1+(cur
col=limit)
```

```
4140 IF curcol>limit THEN curcol
=curcol-1
4150 LOCATE curcol,y%
4160 WEND
4170 CURSOR 0,0
4180 ret$=""
4190 WHILE start<limit
4200 LOCATE start,y%
4210 ret$=ret$+COPYCHR$(#0)
4220 start=start+1
4230 WEND
4240 RETURN
5000 PRINT#1,"Deleting now..."
5010 FOR records=current TO 99
5020 FOR field=1 TO fields%
5030 store$(field,records)=store
$(field,records+1)
5040 NEXT field
5050 NEXT records
5060 GOSUB 2000
5070 n=n-1
5080 CLS#1
5090 RETURN
6000 PRINT#1,"Are you sure (y/n)
"
6010 PRINT#1,"All data will be l
ost."
6020 CURSOR 1,1
6030 k$=INKEY$:IF k$="" THEN 603
0
6040 k$=LOWER$(k$)
6050 IF k$="y" THEN CHAIN "datab
ase.bas"
6060 CLS#1
6070 RETURN
```

When overlays are used, there is inevitably a pause when they are read in from disc. This can be irritating if it occurs too often. It is therefore a good idea for all the parts of the main file manipulating program to be in memory at once. However, there are often parts of the program which are only used at the beginning, and these can be overwritten by other parts, keeping the overall size down. This is what is done here.

When listing 10 originally loads, there are no lines between 220 and 2000. However, you will find references to line 1000 in the listing. In fact, two overlays use this line number range.

The first of these overlays is listing 11. This is CHAIN MERGEd by line 60 and then called by line 70. This overlay prints the commands and the field names on the screen. Note that line 60 specifies a start line number (70) for the combined program. If it did not, the combined program would restart from line 10, and would go into a recursive loop, continually reloading listing 11 and going back to the start.

```
1000 REM Listing 11
1010 REM save as "screen.ovl"
1020 LOCATE 2,2
1030 PRINT "COMMANDS:"
1040 LOCATE 2,4
1050 PRINT "Add Back First Last
Next"
1060 LOCATE 2,5
1070 PRINT "Edit Delete Save Res
tart"
1080 LOCATE 2,7
1090 PRINT "Press first letter"
1100 FOR names=1 TO fields%
1110 LOCATE 1,14+names
1120 PRINT store$(names,0)
1130 NEXT names
1140 RETURN
```

After drawing the screen, the program returns to line 80. This CHAIN MERGEs listing 12, which is the routine to add an entry to the file. Note that line 80 both specifies a start line number (90), and also a delete line range. It is always a good idea to delete the line range of the part of the program being overwritten, in case it includes line numbers not in the new part, which would remain.

```
1000 REM listing 12
1010 REM save as "add.ovl"
1020 PRINT#1,"Type in each line"
1030 PRINT#1,"then press RETURN"
1040 current=n
1050 GOSUB 2000
1060 FOR lines = 1 TO fields%
1070 x%=20:y%=14+lines:length=60
1080 in$="":GOSUB 4000
1090 store$(lines,n)=ret$
1100 NEXT lines
1110 n=n+1
1120 CLS#1
1130 RETURN
```

When using CHAIN MERGE, you will nearly always need to specify a start line number, and it will very often be the line after the one containing the CHAIN MERGE statement.

The program of listing 10+12 is quite easy to use, and instructions for each stage are printed on the screen. Note that lines 4000 to 4240 are exactly the same as the library editor, listing 6, so if you have this on disc it can be MERGEd and renumbered.

The next point of current interest is in line 200. This line chains listing 13, the file saving program. This program contains several interesting points.

Line 40 produces a file directory. However, it only lists files with the extension ".DBF". This is because this program will always save files with that extension, over riding any given extension. This is done by lines 80 to 110. Line 80 is a simple INPUT for the filename. If an extension is included in this, line 90 will detect the full stop, and line 100 will trim the string to remove the

```
10 REM Listing 13
20 REM save as "SAVER.BAS"
30 MODE 1
40 |DIR,"*.DBF"
50 LOCATE 1,20
60 PRINT "Please enter filename:
"
70 LOCATE 1,21
80 INPUT filename$
90 dotpos=INSTR(filename$,".")
100 IF dotpos THEN filename$=LEF
T$(filename$,dotpos-1)
110 filename$=LEFT$(filename$,8)
+".DBF"
120 OPENOUT FILENAME$
130 WRITE#9,"DATABASE PROG FILE"
140 WRITE#9,fields%,n
150 FOR records=0 TO n-1
160 FOR field=1 TO fields%
170 WRITE#9,store$(field,records
)
180 NEXT field
190 NEXT records
200 CLOSEOUT
210 CLS
220 LOCATE 10,10
230 PRINT "(C)ontinue with file
or"
240 LOCATE 10,12
250 PRINT "(S)tart new file?"
260 LOCATE 10,16
270 PRINT "press letter key"
280 LOCATE 10,18:CURSOR 1,1
290 k$=INKEY$:IF k$="" THEN 290
```

```
300 k$=LOWER$(k$):PRINT UPPER$(k
$)
310 IF k$="c" THEN MODE 2: CHAIN
 "DATAMAIN.BAS",60
320 IF k$="s" THEN CHAIN "DATABA
SE.BAS"
330 GOTO 280
```

extension. Line 110 makes sure the first part of the filename is not too long, removing any excess. Line 110 then adds the extension.

Note that these lines are identical to lines 80 to 110 of listing 14, which therefore will only load a file with the .DBF extension.

```
10 REM Listing 14
20 REM save as "LOADER.BAS"
30 MODE 1
40 |DIR,"*.DBF"
50 LOCATE 1,20
60 PRINT "Please enter filename:
"
70 LOCATE 1,21
80 INPUT filename$
90 dotpos=INSTR(filename$,".")
100 IF dotpos THEN filename$=LEF
T$(filename$,dotpos-1)
110 filename$=LEFT$(filename$,8)
+".DBF"
120 OPENIN FILENAME$
130 INPUT#9,ident$:IF ident$<>"D
ATABASE PROG FILE" THEN GOSUB 10
00:GOTO 30
140 INPUT#9,fields%,n
150 DIM store$(fields%,100)
160 FOR field=1 TO fields%
170 FOR element=1 TO 100
180 store$(field,element)=STRING
```

```
$(60,32)
190 NEXT element
200 NEXT field
210 FOR records=0 TO n-1
220 FOR field=1 TO fields%
230 INPUT#9,store$(field,records
)
240 NEXT field
250 NEXT records
260 CLOSEIN
270 MODE 2
280 CHAIN "DATAMAIN.BAS",60
290 END
1000 LOCATE 1,23
1010 PRINT "File not for this pr
ogram"
1020 PRINT "Press any key to con
tinue"
1030 WHILE INKEY$="":WEND
1040 RETURN
```

As further protection against loading a file which does not belong to the program, line 130 of listing 13 writes a line to the file which serves only to identify it. This line is checked by line 130 of listing 14, and if not correct, lines 1000 to 1040 of listing 14 are called.

Lines 140 to 190 write the data out to the disc file. After this, lines 220 to 330 offer the options of continuing with the existing file or starting a new one. If the first option is chosen, listing 10 is chained again, running from line 60, as after loading a file. If the other option is selected, the whole program sequence is started from the beginning by chaining listing 9.

The differences between the files created by this program and the files created by the programs of Chapter 4 show some interesting differences in approach. The files created by the Chapter 4 programs can have any filename and can easily be read

by other programs. Also, it is quite possible to read other types of file into the program of listing 3, though the results will often not be sensible. On the other hand, the programs in this chapter force an extension on the user, and include a device to identify the files belonging to the program in an effort to avoid errors. This leads to a loss of flexibility, as the files cannot then be so easily used by other programs. It is really up to you which approach you feel is more valid.

Writing and debugging a program using overlays is necessarily a more complex problem than writing and debugging a one-piece job. Often, you will need to write and save all the parts to disc before any running or testing can begin. If you find an error in a line when running the program with an overlay in, remember that correcting the line and saving the whole program will not correct the overlay, which is a separate file on disc. In any case you should not normally save the program with overlays in place. What you must do is load the overlay on its own, correct it, and then save it.

When writing your own overlay programs, remember that if you add lines to an overlay you may need to alter the delete line number range in the appropriate CHAIN MERGE statement. Any renumbering of programs which use overlays must be done with extreme care.

It follows from this that writing a program using overlays should not be undertaken lightly, and should only be considered if you really need to write a very complex program, or need to use large data files in memory.

Chapter 6

CP/M TURNKEY DISCS

In this chapter we will see how a CP/M program disc can be configured so that it will boot CP/M and load and run an applications program in a single operation. We will also look at some of the other things which may be done to configure a program for the user's particular combination of equipment, and in particular at the SETKEYS.COM program which allows the keyboard to be set up for the specific requirements of applications programs.

We will be looking at two programs in particular. Firstly, DR LOGO, because this is simple to set up, and because every Amstrad user has it as it comes free with the computer. Secondly, we will be looking at DR DRAW, because this is typical of more extensive CP/M applications, having a number of files and overlays, and requiring more extensive preparation.

Firstly, why 'turnkey'. This comes from the dim and distant days of computing (i.e. a couple of years ago) when computers had a key switch system, rather like car ignition keys, to turn them on. A turnkey disc is thus one which can be used right from the initial switch on.

When CP/M is first booted using the |CPM command from BASIC, as soon as the system files have been loaded the system checks to see if there is a file called PROFILE.SUB on the disc. If this file is present, the transient program SUBMIT.COM is loaded, and this program then executes the commands in the file PROFILE.SUB. If this file is not present, CP/M goes into console command mode.

The PROFILE.SUB file can contain any required commands, and will usually contain such things as commands to select the character set required for the language in use, for example LANGUAGE 3 for English users, which selects the pound sign instead of the hash among other things. It may also contain a SETKEYS command to configure the keyboard.

PROFILE.SUB may also contain the name of an applications program. If it does, and provided the files for the program are present on the disc, it will load and run. Thus you can have your auto-run application.

In fact, the system will return to the PROFILE.SUB file after the applications program has finished, provided it is terminated correctly and not by resetting the computer, so commands can be included in the PROFILE.SUB file after the applications program, and will be executed. These could include the file KEYS.CCP included on the CP/M discs, which sets up the keyboard for the console command processor. This would be particularly useful if the applications program requires a lot of keyboard configuration.

The following are the requirements to prepare a CP/M turnkey disc:-

1. The disc must be formatted in system format, as CP/M will be booted from it.
2. The CP/M system file (C10CPM.EMS) must be present on the disc.
3. There must be a PROFILE.SUB file on the disc.
4. The file SUBMIT.COM must be on the disc.
5. If a keyboard setup is required, the file SETKEYS.COM must be on the disc.
6. All necessary files for the applications program must be on the disc. However, some applications may allow some of the program files to be on drive B, be it an actual second drive or the second logical drive on the one physical drive (DR DRAW is like this).

Much of preparing a turnkey disc thus consists of copying the required files onto the disc (having first formatted it in data format) using PIP. You will, however, have to write the PROFILE.SUB file yourself. This is not difficult.

The PROFILE.SUB file is a simple text file, in ASCII format. It simply contains the required commands in the form in which you type them in at the keyboard. It can be prepared in several ways. Firstly, you could use the CP/M utility ED, but I wouldn't recommend it as this program is notoriously difficult to use. Secondly, you could use PIP to copy keyboard input to a disc file. This is quite easy, but if you make a typing error and correct it, both the mistake and the deletes are sent to the file – however this will not stop it working. Thirdly, you could use a word processor to create the file as most word processors for the Amstrad create straight ASCII files provided you do not insert any special printer

control characters. Finally, you could create the file from BASIC, using PRINT#9 (not WRITE in this instance) and this is a good method if you will not be preparing many discs.

We will now go through the preparation of a turnkey DR LOGO disc.

1. Format a disc in system format using DISCKIT3.
2. Using PIP, copy the following files from the CP/M discs supplied with the computer onto the working disc.

    C10CPM3.EMS
    LOGO3.COM
    SUBMIT.COM
    SETKEYS.COM
    KEYS.DRL
    LOGO3.SUB

In addition, if you want to leave the program and go back to the CCP, you should include the file KEYS.CCP, and I have also included the file AMSDOS.COM on my working disc to allow a return to BASIC without completely resetting the computer.

The following is the contents of the file LOGO3.SUB supplied on the discs with the computer.

```
setkeys keys.drl
logo3
```

In fact, this file could simply be renamed PROFILE. SUB and would have the desired effect. However, if you want to reset the keyboard for the console command processor on quitting LOGO, you could create a PROFILE.SUB file with the following contents.

```
submit logo3
setkeys keys.ccp
```

This could be created by the simple BASIC program of listing 15.

```
10 REM Listing 15
20 REM creates PROFILE.SUB for D
R LOGO
30 OPENOUT "PROFILE.SUB"
40 PRINT#9,"submit logo3"
50 PRINT#9,"setkeys keys.ccp"
60 CLOSEOUT
```

Alternatively, you could ignore the LOGO3.SUB program, and create the following PROFILE.SUB file, perhaps using listing 16.

```
setkeys keys.drl
logo3
setkeys keys.ccp
```

```
10 REM Listing 16
20 REM alternative PROFILE.SUB
30 REM OPENOUT "PROFILE.SUB"
40 PRINT#9,"setkeys keys.drl"
50 PRINT#9,"logo3"
60 PRINT#9,"setkeys keys.ccp"
70 CLOSEOUT
```

This will save 1k of disc space – perhaps useful if you want to keep LOGO program or picture files on the program disc. In any case, you should have over 80k of free space on the disc.

Having prepared this disc, you should be able, after turning the computer on, to insert this disc into drive A and simply type |CPM. This should result in CP/M being booted and DR LOGO loading and running. You should see the commands in the PROFILE .SUB file appear on the screen just before they are executed.

One further point about turnkey discs. The transient program SUBMIT.COM opens a small temporary file on the disc when it runs. It follows that the disc must be write-enabled, and must also have some free space on it.

We will now examine the production of a turnkey disc for the DR DRAW application program. This is a large and complex program which allows drawings and diagrams to be drawn on the screen and subsequently printed out onto a printer or plotter. The program uses GSX, and requires a large number of files to be on the disc(s), including a number of overlays, and also GSX device drivers, and font files for the three fonts (from a choice of eight) which the program can use for text.

In fact, all the required files cannot be fitted onto one side of a disc. If you use only a single disc drive, the files can be on the two sides of one disc. If you use two drives, the files must be on two

separate discs, with the disc with the system file in drive A. The space remaining on disc B (be it a separate disc or the other side of disc A) is used to store the picture file on which you are working, and as DR DRAW creates large files, it is desirable to leave as much space on this disc as possible (especially if you want to store more than one file on it).

To do this, it is a possible to put as many files as possible on the drive A disc, but if you are using a single drive, this can lead to a lot of disc swapping. The file arrangement suggested here is suitable for use with a single drive. The step-by-step procedure to create the disc is as follows:-

1. Format the disc for A as data format.
2. Copy the following files onto the disc for A with PIP.

    C10CPM3.EMS
    KEYS.CCP
    DDFXLR7.PRL*
    DDHP7470.PRL*
    SETKEYS.COM
    DRAW.COM
    ASSIGN.SYS
    DDMODE1.PRL
    GSX.SYS
    DDMODE2.PRL
    SUBMIT.COM

* These files are the GSX device drivers for hard- copy devices You nay need a different file for your printer.

3. Format the disc for B as data format.
4. Copy the following files onto the disc for B using PIP.

    DRAW.001 to DRAW.009 (use PIP B:=A:DRAW.00*)
    DRMSG80.TXT
    DRMSG40.TXT
    FONTA.BIN
    FONTB.BIN
    FONTC.BIN

In addition, you will need a PROFILE.SUB file on disc A, with the following contents:

```
SETKEYS KEYS.DRW
DRAW
SETKEYS KEYS.CCP
```

Listing 17 is a BASIC program which will produce this file.

You will also need to produce the file KEYS.DRW, so we will now look at the transient program SETKEYS.COM.

```
10 REM Listing 17
20 REM creates PROFILE.SUB for D
R DRAW
30 OPENOUT "PROFILE.SUB"
40 PRINT#9,"SETKEYS KEYS.DRW"
50 PRINT#9,"DRAW"
60 PRINT#9,"SETKEYS KEYS.CCP"
70 CLOSEOUT
```

SETKEYS.COM

This program reconfigures the keyboard from data contained in a special command file. These files are normally named KEYS and have an extension which indicates the program or purpose for which the file is intended. However, this program will accept any valid filename.

The command file is in ASCII format. Each line contains a key definition or an expansion token definition. Each key can be defined for its unshifted state, in combination with SHIFT, and/or in combination with CONTROL. An expansion definition assigns a string to an expansion token.

Each key definition starts with the number of the key to be redefined. These are the numbers in the chart on the right hand end of the CPC 6128 computer. Next comes the shift state or states for the definition. These are indicated by single letters, N or nothing for not shifted, S for the key in conjunction with shift, and C for the key in conjunction with control. Finally comes the character which the key is to produce. Anything after this on the line is treated as a comment.

If the character is a printing character (i.e. ASCII codes 32 to 255 or #20 to #FF hex.) other than quotation marks or the upward arrow (under the pound sign), it can be given in quotation marks.

If the character required is the upward arrow, use two upward arrows.

If the character required is a quotation mark, use an upward arrow followed by a quotation mark.

To produce control codes (i.e. ASCII codes below 32 or #20 hex.) you can use an upward arrow followed by a character. Thus " ↑ A" gives CTRL-A, ASCII code 1, " ↑ B" gives CTRL-B, ASCII code 2, and so on. Alternatively, you can use the vertical arrow followed by the name of the control code (see appendix C).

Examples:-

68 NSC " ↑ 'b4'" makes TAB produce code 180 in all states
18 NS " ↑ '#A'" makes normal & shifted RETURN produce LF
18 C " ↑ '#D'" makes RETURN/CONTROL produce CR
6 " ↑ 'FF'" makes unshifted ENTER produce form feed.

In fact, the purpose of the KEYS.DRW file on my disc is to reset the cursor keys to produce their default codes. They are modified by the contents of the file KEYS.CCP and this makes them unsuitable for use with DR DRAW. The contents of my KEYS.DRW file is as follows.

```
1 N " ↑ '#FB'"
1 S " ↑ '#F7'"
1 C " ↑ '#F3'"
2 N " ↑ '#F9'"
2 S " ↑ '#F5'"
2 C " ↑ '#F1'"
8 N " ↑ '#FA'"
8 S " ↑ '#F6'"
8 C " ↑ '#F2'"
0 N " ↑ '#F8'"
0 S " ↑ '#F4'"
0 C " ↑ '#F0'"
```

Further examples can be obtained by examining the HELP files on side 3 of the CP/M discs.

Now we come to expansion tokens. Expansion tokens are numbers in the range #80 to #9F hex (128 to 159 decimal). Any key can be set to return one of these values using SETKEYS as described above. If a string is then assigned to that token, each time the key is pressed, the string will be produced. Thus assigning a string to a key is a two stage process.

An expansion token consists of an E followed by the string, and can be followed by a comment if required.

Examples:-
E #80 "Babani Books"
E #81 "Amstrad Computers"

If you also have

15 S " ↑ '#80'"
13 S " ↑ '#81'"

in the KEYS file, key 10 shifted will produce Babani Books when pressed, and key f1 shifted will produce Amstrad Computers.

Appendix A

FILE COPYING AND TRANSFER

BASIC Programs

DISC TO DISC. Best done by loading from one disc, then saving onto a new disc. Can also be done with PIP, or by copying a whole disc with DISCKIT 3.

DISC TO TAPE. Load from disc, then give command |TAPE .OUT, then save to tape. Can also be done using CSAVE program under CP/M 2.2 (CSAVE must be on the same disc as the BASIC program if you have only a single drive).

TAPE TO DISC. Give command |TAPE.IN, load from tape, then save to disc. Can also be done using CLOAD program under CP/M 2.2 (CLOAD must be on the destination disc if you have only a single drive). Note that protected BASIC programs on tape cannot be copied or transferred.

Machine Code Programs

DISC TO DISC. Use PIP under CP/M. It is recommended to verify all copying operations on binary files when using PIP, as a bug in this program can corrupt long files. Note that some commercial machine code programs (especially arcade type games) may be protected against copying.

DISC TO TAPE. You need to know the length of the file, the load address, and the run address (if different to the load address).

Type:

```
h=himem
memory (load address)-1
load "(program name)"
|tape
```

You can then save the program to tape using

```
save "(program name)",b,(load addr.),(length),(run addr.)
```

The run address need not be given if it is the same as the load address.

TAPE TO DISC. You need to know the length of the file, the load address, and the run address (if different to the load address).

Type:-

```
h=himem
|tape
memory (load address)-1
load "(program name)"
|disc
```

You can then save the program to disc using

```
save "(program name)",b,(load addr.),(length),(run addr.)
```

The run address need not be given if it is the same as the load address.

Data Files

DISC TO DISC. With AMSDOS programs and your own BASIC programs it is often easiest to load the file into the program and then save it onto the new disc. With CP/M program files, and when the first method is not convenient, PIP can be used.

DISC TO TAPE. Use CSAVE program under CP/M 2.2. The program CSAVE must be on the same disc as the file to be copied if you have only a single drive.

TAPE TO DISC. Use CLOAD program under CP/M 2.2. The program CLOAD must be on the destination disc if you have only a single drive.

Appendix B

FILE EXTENSION TYPES

Extension	*Type*	*Example*
ASM	Assembly language source file for ASM. MUST be used.	PROG_A.ASM
BAS	Used for BASIC program files, including source files for compiled BASICs under CP/M.	FILER.BAS
BAK	Automatically applied to an old file when a new version is saved with the same name.	CHAPTER2.BAK
BIN	Used for binary (mostly machine code program) files.	INVADERS.BIN
COM	Used for CP/M transient programs and applications programs. Compulsory in most cases.	PIP.COM LOGO3.COM
DOC	Text files for word processors.	CHAPTER2.DOC
HEX	An alternative to BIN, used by CP/M programs in hex format ready to be LOADed.	PROG_1.HEX
OVL	Used for program overlays, both machine code and BASIC for CHAIN MERGE.	SCREEN.OVL
PIC	Used for DR LOGO picture files. Must be present for file to load.	PATTERN.PIC
PIX	Used for DR DRAW picture files. Must be present for file to be reopened.	DISC.PIX
PRL	Used for GSX device driver files. Compulsory.	DDFXLR7.PRL
SUB	Command file for the SUBMIT transient program.	LOGO3.SUB

Extension	*Type*	*Example*
SYS	Used for CP/M and GSX system files. Compulsory in most cases.	ASSIGN.SYS
$$$	Used for temporary or newly-opened files. Normally erased or changed to another extension on closing. If this extension is found, it normally means an error has occured at some time.	ANYTHING.$$$

Appendix C

CONTROL CHARACTERS

Name	*Hex*	*Decimal*
NUL	#00	0
SOH	#01	1
STX	#02	2
EXT	#03	3
EOT	#04	4
ENQ	#05	5
ACK	#06	6
BEL	#07	7
BS	#08	8
HT	#09	9
LF	#0A	10
VT	#0B	11
FF	#0C	12
CR	#0D	13
SO	#0E	14
SI	#0F	15
DLE	#10	16
DC1	#11	17
DC2	#12	18
DC3	#13	19
DC4	#14	20
NAK	#15	21
SYN	#16	22
ETB	#17	23
CAN	#18	24
EM	#19	25
SUB	#1A	26
ESC	#1B	27
FS	#1C	28
GS	#1D	29
RS	#1E	30

Please note following is a list of other titles that are available in our range of Radio, Electronics and Computer books.

These should be available from all good Booksellers, Radio Components Dealers and Mail Order Companies.

However, should you experience difficulty in obtaining any title in your area, then please write directly to the Publisher enclosing payment to cover the cost of the book plus adequate postage.

If you would like a complete catalogue of our entire range of Radio, Electronics and Computer books then please send a Stamped Addressed Envelope to:-

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

160	Coil Design and Construction Manual	£2.50
202	Handbook of Integrated Circuits (ICs) Equivalents and Substitutes	£2.95
205	Hi-Fi Loudspeaker Enclosures	£2.95
208	Practical Stereo and Quadrophony Handbook	£0.75
214	Audio Enthusiast's Handbook	£0.85
219	Solid State Novelty Projects	£0.85
220	Build Your Own Solid State Hi-Fi and Audio Accessories	£0.85
221	28 Tested Transistor Projects	£2.95
222	Solid State Short Wave Receivers for Beginners	£1.95
223	50 Projects Using IC CA3130	£1.25
224	50 CMOS IC Projects	£2.95
225	A Practical Introduction to Digital ICs	£1.75
226	How to Build Advanced Short Wave Receivers	£2.95
227	Beginners Guide to Building Electronic Projects	£1.95
228	Essential Theory for the Electronics Hobbyist	£2.50
BP1 + 14	First & Second Books of Transistor Equivalents & Substitutes	£3.50
BP2	Handbook of Radio, TV, Industrial and Transmitting Tube and Valve Equivalents	£0.60
BP6	Engineer's and Machinist's Reference Tables	£1.25
BP7	Radio and Electronic Colour Codes Data Chart	£0.95
BP27	Chart of Radio, Electronic, Semiconductor and Logic Symbols	£0.95
BP28	Resistor Selection Handbook	£0.60
BP29	Major Solid State Audio Hi-Fi Construction Projects	£0.85
BP33	Electronic Calculator Users Handbook	£1.50
BP34	Practical Repair and Renovation of Colour TVs	£2.95
BP36	50 Circuits Using Germanium Silicon and Zener Diodes	£1.50
BP37	50 Projects Using Relays, SCRs and TRIACs	£1.95
BP39	50 (FET) Field Effect Transistor Projects	£1.75
BP42	50 Simple LED Circuits	£1.95
BP44	IC 555 Projects	£2.50
BP45	Projects in Opto-electronics	£1.95
BP48	Electronic Projects for Beginners	£1.95
BP49	Popular Electronic Projects	£2.50
BP53	Practical Electronics Calculations and Formulae	£2.95
BP54	Your Electronic Calculator and Your Money	£1.35
BP56	Electronic Security Devices	£2.50
BP58	50 Circuits Using 7400 Series ICs	£2.50
BP59	Second Book of CMOS IC Projects	£1.95
BP60	Practical Construction of Pre-amps, Tone Controls, Filters and Attenuators	£1.95
BP61	Beginners Guide to Digital Techniques	£1.95
BP62	The Simple Electronic Circuit & Components (Elements of Electronics – Book 1)	£3.50
BP63	Alternating Current Theory (Elements of Electronics – Book 2)	£3.50
BP64	Semiconductor Technology (Elements of Electronics – Book 3)	£3.50
BP65	Single IC Projects	£1.50
BP66	Beginners Guide to Microprocessors and Computing	£1.95
BP67	Counter Driver and Numeral Display Projects	£2.95
BP68	Choosing and Using Your Hi-Fi	£1.65
BP69	Electronic Games	£1.75
BP70	Transistor Radio Fault-finding Chart	£0.95
BP71	Electronic Household Projects	£1.75
BP72	A Microprocessor Primer	£1.75
BP73	Remote Control Projects	£2.50
BP74	Electronic Music Projects	£2.50
BP75	Electronic Test Equipment Construction	£1.75
BP76	Power Supply Projects	£2.50
BP77	Microprocessing Systems and Circuits (Elements of Electronics – Book 4)	£2.95
BP78	Practical Computer Experiments	£1.75
BP79	Radio Control for Beginners	£1.75
BP80	Popular Electronic Circuits – Book 1	£2.95
BP82	Electronic Projects Using Solar Cells	£1.95
BP83	VMOS Projects	£1.95
BP84	Digital IC Projects	£1.95
BP85	International Transistor Equivalents Guide	£3.50
BP86	An Introduction to BASIC Programming Techniques	£1.95
BP87	50 Simple LED Circuits – Book 2	£1.35
BP88	How to Use Op-Amps	£2.95
BP89	Communication (Elements of Electronics – Book 5)	£2.95
BP90	Audio Projects	£1.95
BP91	An Introduction to Radio DXing	£1.95
BP92	Electronics Simplified – Crystal Set Construction	£1.75
BP93	Electronic Timer Projects	£1.95
BP94	Electronic Projects for Cars and Boats	£1.95
BP95	Model Railway Projects	£1.95
BP97	IC Projects for Beginners	£1.95
BP98	Popular Electronic Circuits – Book 2	£2.25
BP99	Mini-matrix Board Projects	£1.95
BP101	How to Identify Unmarked ICs	£0.95
BP103	Multi-circuit Board Projects	£1.95
BP104	Electronic Science Projects	£2.25

BP105	Aerial Projects	£1.95
BP106	Modern Op-amp Projects	£1.95
BP107	30 Solderless Breadboard Projects – Book 1	£2.25
BP108	International Diode Equivalents Guide	£2.25
BP109	The Art of Programming the 1K ZX81	£1.95
BP110	How to Get Your Electronic Projects Working	£1.95
BP111	Audio (Elements of Electronics – Book 6)	£3.50
BP112	A Z-80 Workshop Manual	£3.50
BP113	30 Solderless Breadboard Projects – Book 2	£2.25
BP114	The Art of Programming the 16K ZX81	£2.50
BP115	The Pre-computer Book	£1.95
BP117	Practical Electronic Building Blocks – Book 1	£1.95
BP118	Practical Electronic Building Blocks – Book 2	£1.95
BP119	The Art of Programming the ZX Spectrum	£2.50
BP120	Audio Amplifier Fault-finding Chart	£0.95
BP121	How to Design and Make Your Own P.C.B.s	£1.95
BP122	Audio Amplifier Construction	£2.25
BP123	A Practical Introduction to Microprocessors	£1.95
BP124	Easy Add-on Projects for Spectrum, ZX81 & Ace	£2.75
BP125	25 Simple Amateur Band Aerials	£1.95
BP126	BASIC & PASCAL in Parallel	£1.50
BP127	How to Design Electronic Projects	£2.25
BP128	20 Programs for the ZX Spectrum and 16K ZX81	£1.95
BP129	An Introduction to Programming the ORIC-1	£1.95
BP130	Micro Interfacing Circuits – Book 1	£2.25
BP131	Micro Interfacing Circuits – Book 2	£2.25
BP132	25 Simple Shortwave Broadcast Band Aerials	£1.95
BP133	An Introduction to Programming the Dragon 32	£1.95
BP134	Easy Add-on Projects for Commodore 64, Vic-20, BBC Micro and Acorn Electron	£2.95
BP135	Secrets of the Commodore 64	£1.95
BP136	25 Simple Indoor and Window Aerials	£1.75
BP137	BASIC & FORTRAN in Parallel	£1.95
BP138	BASIC & FORTH in Parallel	£1.95
BP139	An Introduction to Programming the BBC Model B Micro	£1.95
BP140	Digital IC Equivalents and Pin Connections	£5.95
BP141	Linear IC Equivalents and Pin Connections	£5.95
BP142	An Introduction to Programming the Acorn Electron	£1.95
BP143	An Introduction to Programming the Atari 600/800 XL	£1.95
BP144	Further Practical Electronics Calculations and Formulae	£4.95
BP145	25 Simple Tropical and MW Band Aerials	£1.75
BP146	The Pre-BASIC Book	£2.95
BP147	An Introduction to 6502 Machine Code	£2.50
BP148	Computer Terminology Explained	£1.95
BP149	A Concise Introduction to the Language of BBC BASIC	£1.95
BP150	An Introduction to Programming the Sinclair QL	£1.95
BP152	An Introduction to Z80 Machine Code	£2.75
BP153	An Introduction to Programming the Amstrad CPC 464 and 664	£2.50
BP154	An Introduction to MSX BASIC	£2.50
BP155	International Radio Stations Guide	£2.95
BP156	An Introduction to QL Machine Code	£2.50
BP157	How to Write ZX Spectrum and Spectrum + Games Programs	£2.50
BP158	An Introduction to Programming the Commodore 16 and Plus 4	£2.50
BP159	How to Write Amstrad CPC 464 Games Programs	£2.50
BP161	Into the QL Archive	£2.50
BP162	Counting on QL Abacus	£2.50
BP163	Writing with QL Quill	£2.50
BP164	Drawing on QL Easel	£2.50
BP169	How to Get Your Computer Programs Running	£2.50
BP170	An Introduction to Computer Peripherals	£2.50
BP171	Easy Add-on Projects for Amstrad CPC 464, 664, 6128 and MSX Computers	£2.95
BP173	Computer Music Projects	£2.95
BP174	More Advanced Electronic Music Projects	£2.95
BP175	How to Write Word Game Programs for the Amstrad CPC 464, 664 and 6128	£2.95
BP176	A TV-DXers Handbook	£5.95
BP177	An Introduction to Computer Communications	£2.95
BP178	An Introduction to Computers in Radio	£2.95
BP179	Electronic Circuits for the Computer Control of Robots	£2.95
BP180	Computer Projects for Model Railways	£2.95
BP181	Getting the Most from Your Printer	£2.95
BP182	MIDI Projects	£2.95
BP183	An Introduction to CP/M	£2.95
BP184	An Introduction to 68000 Assembly Language	£2.95
BP185	Electronic Synthesiser Construction	£2.95
BP186	Walkie-Talkie Projects	£2.95
BP187	A Practical Reference Guide to Word Processing on the Amstrad PCW 8256 and PCW 8512	£5.95
BP188	Getting Started with BASIC and LOGO on the Amstrad PCW 8256 and PCW 8512	£6.95
BP189	Using Your Amstrad CPC Disc Drives	£2.95
BP190	More Advanced Electronic Security Projects	£2.95
BP191	Simple Applications of the Amstrad CPCs for Writers	£2.95
BP192	More Advanced Power Supply Projects	£2.95
BP193	Starting LOGO	£2.95
BP194	Modern Opto Device Projects	£2.95
BP195	An Introduction to Communications and Direct Broadcast Satellites	£3.95
BP196	BASIC & LOGO in Parallel	£2.95

BERNARD BABANI BP189

Using Your Amstrad CPC Disc Drives

■ The purpose of this book is to show how the disc drive of the remarkable Amstrad CPC6128 can be used to best effect. Much of the information also applies to users of the CPC664 and 464 with DD1, and some of the general information and CP/M Plus details also apply to the PCW8256 and 8512.

■ This book covers all aspects of disc drive usage, from loading programs to using the discs to store data files from applications, and using the discs to help write your own programs in BASIC and other languages etc.

■ Both native AMSDOS and the CP/M operating system are covered, and the use of some of the most popular applications programs are briefly described including TASWORD 6128 and DR DRAW.

ISBN 0-85934-163-1

00295

9 780859 341639

£2.95

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/