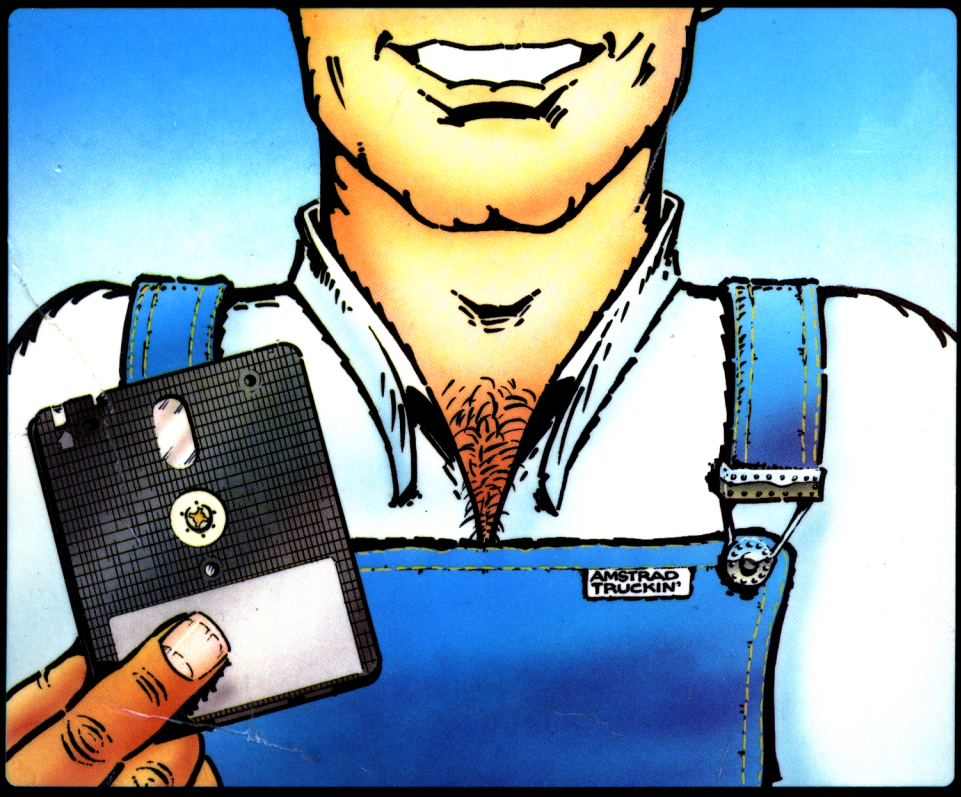
THE Amstrad Disc Companion

FOR THE CPC-464, 664, AND 6128



Simon Williams

SIGMA PRESS

The Amstrad Disc Companion

Simon Williams

Σ
SIGMA
PRESS

ISBN 1 85058 034 0

Published by:

SIGMA PRESS
98a Water Lane
Wilmslow
Cheshire
U.K.
Printed in Malta by Interprint Limited

Distributors:

U.K., Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester
West Sussex, England

Australia:
JOHN WILEY & SONS INC.
GPO Box 859, Brisbane
Queensland 40001
Australia

Acknowledgements
CPC-464, CPC-664 and CPC-6128 are trademarks of Amstrad Consumer Electronics PLC. CP/M is a trademark of Digital Research.

INTRODUCTION

When you decide to add a DDI-1 disc drive to your Amstrad CPC464 computer, or to buy an Amstrad 6128, which has an integral disc drive, you will no doubt have many questions about the use of discs. This book sets out to answer these, and to describe the many advantages of disc storage over cassette.

As well as showing you how to use discs with your Amstrad computer, later chapters outline some of the many application programs which are particularly suited to a computer with disc drives. These sections of the book concentrate on small-business uses of Amstrad's micros and the use of languages other than BASIC. Discs are most suited to business, as they offer the facility to store a large amount of information and to transfer it very quickly to and from the computer.

There are two ways of obtaining programs to use with your Amstrad discs.

The first is to buy a ready-made program to do the job. This book will look at several examples of the best software currently available.

The second is to write a program yourself. If you have only used the programming language BASIC until now, then you will be interested to know that adding disc drives to your computer will give you the option of several alternative languages which may be much better suited to certain jobs. Indeed, a second language, Logo, is supplied with the DDI-1 and the CPC6128/664. Logo was originally designed to help in the teaching of mathematics and logical thinking, but has also found many uses in Artificial Intelligence.

Another language which is particularly suited to use with discs is Pascal. A version of this language is available for the Amstrad micros, and its particular advantages will be looked at later on.

To show that writing your own software is not as daunting a task as you might at first think, the book rounds off with the listing of a ready-made application program which you can type into your machine and use with your disc drives.

This book should provide you with a good understanding of the following, in varying degrees of detail:

- ★ How a disc and disc drive work
- ★ AMSDOS and CP/M, and how to use them

- ★ A machine-code assembler and monitor
- ★ Logo and Pascal
- ★ A word processor, database and spreadsheet
- ★ How to write an application program using disc files

By introducing the CPC 6128, with its more sophisticated CP/M Plus operating system and extra 64K memory, Amstrad has added a worthy 'top-end' machine to the market. To fully cover the extended facilities of this machine, each chapter includes a '6128 Special' section, which describes the additional features of the machine and the differences in the way the new CP/M works. The earlier sections of each chapter are also applicable to the 6128, however, as the CP/M 2.2 operating system is supplied by Amstrad, to ensure compatibility with software written for the CPC464 and CPC664 micros.

Amstrad's micros offer a high-quality, low-cost entry into computing, and with the addition of one or two disc drives, they may be used in many business and educational applications. This book makes esential reading for all those who want to get the most out of their disc drives, and their Amstrad micros.

CONTENTS

Chapter One

Getting Started

CONNECTING THE DDI-1 TO THE CPC464

(Owners of CPC664 and CPC 6128 computers may safely skip this section).

If you have bought a DDI-1 for your Amstrad CPC464 the first thing you'll need to do is to connect the two units together.

The DDI-1 consists of the disc drive itself, connected by a length of ribbon cable to a rectangular grey box which houses the *disc interface*. The disc interface is a small extra circuit-board which handles communications between the computer and the disc drive. All you need to know about it at this stage is that the rectangular box has to be plugged in to the slot marked 'FLOPPY DISC' on the rear of the CPC464.

Position the disc drive as far away from your CTM640 or GT64 monitor as possible. If your monitor normally sits behind the computer, put your disc drive to the right of the keyboard unit. The positioning is important because the mains transformer inside the monitor may radiate a strong enough magnetic field to effect the information stored on your discs. This field is, of course, totally harmless to anyone using the computer.

Once you have wired a three pin plug to the end of the mains lead on the disc drive and plugged it in, you should switch the disc drive on (there's a small slide switch on the back of the drive), *before* you switch the computer on.

Each disc, like a cassette, can store information on both sides, and when you insert a disc into the drive you should make sure that the side you want to use faces upwards. To eject the disc, you press the small rectangular button on the front of the drive. *Never* try to eject a disc while you are loading or saving information on it, and always eject any disc *before* you switch the disc drive or computer off.

If you need any further information on setting up your DDI-1 disc drive, you should refer to the *User Manual* supplied with the drive.

GETTING GOING WITH DISCS

When you first use a home computer, you probably start by typing in some of the example BASIC programs from the Manual. As you do this, you discover many of the features of the computer, and may well want to come back to the programs later on, but without having to type in the whole listing from the Manual again.

It is probably at this point that you realise how useful it can be to record your programs on tape, so that you can load them again later, straight from the cassette. The CPC464 has a cassette recorder built-in and partly controlled by the computer itself.

By typing the command:

```
SAVE "program"
```

you can save the program currently in the memory of the computer onto a cassette. When you want to load the same program back into the micro, you simply wind the cassette back to the start, type:

```
LOAD "program"
```

and, hey presto, you can carry on where you left off.

When you want to save another program, you can either use a different cassette, or you can wind the tape on the original one to a point beyond the end of the first recording and type `SAVE "program"` again. The name between the quotation-marks can be anything you want of course, and can even be the same name as you used before. So you could have a cassette with lots of programs recorded on it, all called `program`. Right from the start, however, you will discover how useful it is to give your recordings different names, to distinguish between them. As you'll see, it's essential to use a different name for each recording when you use discs.

The recordings of a computer program (or the data it uses) are usually known as *files*, and the name given to each file is known, logically enough, as the *filename*. In the example just used, the filename was `program`.

When you record a file on cassette, the information is recorded along the length of the tape from the start of the file to its end. If you have several files recorded on a cassette, then to find a particular one you have to wind the tape through to the right starting point, past all the files recorded ahead of the one you now want. This can be quite a long-winded job, particularly if you use C90 cassettes!

With a disc however, things are rather different. Inside the tough case of

each disc is a thin plastic wafer, coated each side with the same kind of metal oxide as is used on cassette tapes. When you insert a disc into a drive, the drive mechanism engages in the sprocket hole in the centre of the disc and spins it round at several hundred revolutions per minute.

The disc drive has a recording head of much the same type as a cassette recorder, and this makes contact with the surface of the disc through the oval hole in the disc case. The metal shutter blocking this hole is automatically drawn back when you insert the disc into the drive. Unlike the recording head in a cassette recorder, the one in a disc drive can move. It moves in and out, from the edge to the centre of the disc. This means that any piece of information recorded on the disc can be reached very quickly, simply by moving the drive head to the right position as the disc spins round.

The drive head is moved automatically, under control of the computer. But how does it know where each file has been stored on the disc? The answer is that each disc has a catalogue of all its files stored on it. When you want to load a file from a disc, you still type `LOAD "program"`, but now the computer goes first to the catalogue section of the disc, discovers where the file `"program"` is stored, and then sends the disc drive head to the right place on the disc to load the file.

If there's no entry in the catalogue with the name `"program"`, the computer will tell you so. You can now see why you must give each file on a disc a unique name. The computer will not, in fact, accept a filename which is the same as any already in its disc catalogue.

If you want to see the files stored on a particular disc, you type:

```
CAT
```

as you would when using cassettes. With a cassette, you have to wait for the tape to play through before the catalogue appears, but a disc catalogue appears immediately. If you catalogue the disc which is provided with the DDI-1 or CPC664 with the side labelled 'CP/M 2.2' uppermost, the screen display will appear as in Figure 1.1.

The figure at the end of the display, 50K in this case, is the number of bytes of free space left on the disc. The figure after each filename shows how long that file is (to the nearest kilobyte). Each disc can hold 180K of programs or data on each side. The AMSDOS command *DIR* performs a similar function to *CAT*, but doesn't display the length of each file or sort the filenames into alphabetical order.

Unlike a cassette, which can be used for file storage from new, a disc needs to have certain signals recorded on it as markers, so files can be located quickly by the computer. This marking process is called *formatting* and must be done on each surface of the disc before it can be used.

```
AMSDOS      .COM    1K    EX1         .BAS    2K
ASM         .COM    8K    EX2         .BAS    1K
BOOTGEN     .COM    2K    FILECOPY    .COM    3K
CHKDISC     .COM    3K    FORMAT      .COM    3K
CLOAD       .COM    2K    LOAD        .COM    2K
COPYDISC    .COM    3K    MOVCPM      .COM   10K
CSAVE       .COM    2K    PIP         .COM    8K
DDT         .COM    5K    ROINTIME    .DEM   26K
DISCCHK     .COM    3K    SETUP       .COM    8K
DISCCOPY    .COM    3K    STAT        .COM    6K
DUMP        .ASM    5K    SUBMIT      .COM    2K
DUMP        .COM    1K    SYSGEN      .COM    2K
ED          .COM    7K    XSUB        .COM    1K

50K free
```

Figure 1.1 A catalogue of CP/M 2.2. system disc

Formatting, and many other operations with discs, are carried out by the *disc operating system*. This is a collection of programs which handles the discs directly. When you give your Amstrad computer a disc command, such as **FORMAT**, one of the disc operating system routines takes over and does the work for you.

The Amstrad disc operating system is called AMSDOS. The routines included in AMSDOS are held in a chip which is housed in the disc interface of the computer. They provide extra commands within Locomotive BASIC, which may either be used in programs or typed in directly at the keyboard.

In addition to AMSDOS, a second operating system is provided with the DDI-1 and CPC664. This is called CP/M (the initials stand for 'Control Program for Microcomputers'). The CPC 6128 is supplied with two different versions of CP/M. This is a more sophisticated operating system than AMSDOS, and is produced by Digital Research. A lot of business programs have been written to work with CP/M, as this disc operating system is available on many different computers. Several of the routines you will need to use with your disc drives are designed to work with CP/M rather than AMSDOS, and the formatting program is one of them. Although CP/M is covered in detail in Chapters 2 and 3, it will be used briefly here, so you can format some blank discs.

Assuming that you have switched on your DDI-1 and CPC464 and done nothing else, you should have the following display on your screen:

```
Amstrad 64K Microcomputer (v1)

1984 Amstrad Consumer Electronics plc
        and Locomotive Software Ltd.

BASIC 1.Ø

Ready
```

The CPC664 and CPC6128 display similar messages but with a revised date and version number for BASIC.

To format a blank disc, first put the master disc supplied into the disc drive, with the 'CP/M 2.2' side uppermost. Now call up the CP/M operating system using the command:

```
|CPM
```

The split vertical bar symbol, |, is essential and can be typed by pressing the '@' key, and the <SHIFT> key together. The reason for its use will be explained later.

The disc will rotate and the screen change to a lighter colour and mode 2, with the message:

```
CP/M 2.2 - Amstrad Consumer Electronics plc
A>
```

displayed at the top of the screen. You now have to load the formatting program from the CP/M disc. Type:

```
FORMAT
```

The disc will spin again, and the message:

```
FORMAT V2.Ø

Please insert disc to be formatted into drive A
then press any key: —
```

will appear. You should now eject the CP/M disc and insert the blank disc you want to format, with the side you want formatted uppermost.

When the prompt refers to 'drive A', it means the first drive connected to your computer. With a CPC664 or CPC6128 this means the one at the end of the keyboard, and with a CPC464 and DDI-1 it means the drive connected directly to the disc interface box. When you are using CP/M, the letter of the drive you are currently working with, appears before the > prompt.

It is possible to connect a second drive to any of the Amstrad computers,

and this would normally be referred to as 'drive B'.

When you 'press any key' (most keys work, but the <ENTER> key is best), the FORMAT routine starts operating. The screen will display the message:

```
Formatting started
Formatting track nn
```

'nn' is a number which counts up from 0 to 39 as the *tracks* of the disc are formatted. The easiest way to think of a track on a computer disc is to compare it to a very short track on a long-playing record. There are 40 tracks on each side of an Amstrad computer disc, with track number 0 at the edge, and track 39 at the centre. The disc drive head can move across them in the same way as the arm of a record player moves across a record.

When the disc has been formatted, the screen will display:

```
Formatting complete
Do you want to format another disc (Y/N):-
```

If you want to format the other side of your disc, or another disc, then you simply insert it into the drive and repeat the process. It's a good idea when you start computing with discs to buy, say, half a dozen of them and format them ready for use. Although you might think it'll be a long time before you fill six discs, you'll be surprised how soon you find uses for them.

When you have finished formatting your discs, you answer **N** to the above question and you will be asked to insert a *CP/M system disc.* This is simply any disc which has a copy of the CP/M operating system recorded on it, such as the disc supplied with the system. When you insert a system disc, the format program will finish and you will see the **A>** prompt appear on the screen again.

Using discs with AMSDOS

Now you have a few formatted discs, you can investigate the new commands available through AMSDOS. You've already seen that you can use the commands **LOAD** and **SAVE** as you would with cassettes, but with discs connected, your computer automatically directs commands to the disc drive instead of the cassette recorder.

Now, what if you want to copy some programs from a cassette onto your disc? You can switch from using a disc drive to using a cassette recorder by typing the command:

```
|TAPE
```

and switch back to the disc drive with:

```
|DISC
```

So you could switch to tape, load a program, switch back to disc and save it again. If you have more than one file you want to transfer in a session, however, you can type:

```
|TAPE.IN
```

and:

```
|DISC.OUT
```

These two commands tell the computer to select tape for all input, or loading, operations and disc for all output, or saving, ones. Once these commands have been given, you can use **LOAD** and **SAVE** as you would normally when copying from cassette to cassette.

To transfer files from disc to tape, you can use the equivalent commands **|TAPE.OUT** and **|DISC.IN** to set things up the other way round. You should note that many commercial tapes and discs are protected to stop illegal copying — these can not be copied in this way.

When you use cassettes with an Amstrad micro, you can load and automatically run a program by pressing the <CTRL> and small <ENTER> key together. This procedure doesn't work with discs, as you have to state the filename when you run a program. You can auto-run a program, however, by typing:

```
RUN "program"
```

The filenames you use when working with cassettes on your Amstrad micro may consist of up to 16 characters, which may be any characters on the keyboard. When using discs, this number is reduced to eight, but an optional *filetype* of up to three characters may be added, to show what kind of information is stored in the file.

If you type **SAVE "program"**, where **program** is a program written in BASIC, AMSDOS will automatically add a filetype **.BAS** (short for BASIC) to the filename. If you catalogue the disc after saving such a file, the entry will be **PROGRAM.BAS**. The full stop is used to separate the filename from the filetype. This style of filename is the same as that used by the CP/M operating system, and helps ensure that AMSDOS files are compatible with it.

Other filetypes which are commonly used are **.BIN** for a machine code file (BINary), **.BAK** for a BAcKup copy of a file, **.COM** for a COMmand file in CP/M (one you can run by just typing its name) and **.TXT** for a TeXT file from, for instance, a word processing program.

There are a few more AMSDOS commands which should be looked at before moving on to discuss CP/M. Two of these will erase and rename a file on a disc.

To erase a cassette file, you simply wipe the tape. With a disc, it's even simpler. You just type:

```
ERASE$="program"
|ERA,@ERASE$
```

The first line assigns the name of the file you want to erase to the variable **ERASE$**. This variable can be any string variable you like. You then type the erase command, again preceded by the vertical bar **|**. You follow this with an @ symbol (this tells the computer that the filename on which you want it to act may be found in the attached variable), and the name of the variable itself. You can't use the filename directly in this command on the CPC 464. It must always be used from within a variable. The command and its associated variable are separated by a comma.

Occasionally you may want to erase several files with similar names from one disc. Say, for instance, that you have three early versions of a program called **ATTEMPT1.BAS**, **ATTEMPT2.BAS**, **ATTEMPT3.BAS**. The **|ERA** command has a special facility — wild cards — which allows you to erase all these files in one go.

A wild card is much like the joker in certain card games (hence its name) in that it is used to replace *any* character, or characters, in the filename. The two wild cards used in AMSDOS are **?** and *****. **?** is used to replace any single character, while ***** is used to replace any number of characters of any type. Thus, for example, the commands:

```
FILES$="ATTEMPT?.BAS"
|ERA,@FILE$
```

will delete all three of the files just mentioned. Note that this won't work unless you include the filetype extension **.BAS** in the filename. If you wanted to delete all BASIC programs from a disc, you could do it by assigning ***.BAS** to the variable **filename$** in the example above.

The rename command, **|REN**, takes a very similar form to **|ERA**, except that two filenames are needed, the old one and the new. Again, each name has to be assigned to a string variable before the command is used, so you might type:

```
OLD$="old-prog"
NEW$="new-program"
|REN,@NEW$,@OLD$
```

The form of this command is very similar to an assignment statement in

BASIC, which would take the form `LET NEW$=OLD$`. The command takes the filename given by `NEW$` and changes it to the filename given by `OLD$`. Again, commas are used as separators. Since no two files on one disc may have the same name, the new filename must be unique to that disc.

The `|DRIVE` command is used to set the default drive on a twin-drive system. You would normally want AMSDOS to deal with drive A, which is the one directly connected to the DDI-1 or integral to the CPC664 and CPC6128, If you wanted it to deal with a second drive by default, however, you could tell AMSDOS by typing:

```
drive$="B"
|DRIVE,@drive$
```

The last command available under AMSDOS is `|USER`. Both CP/M and AMSDOS have the facility of dividing the disc directory into up to 16 parts. Each part, which is normally thought of as being available to a different user, can be separately catalogued. Files can be loaded and saved from a particular part by telling AMSDOS which user you are. Users are numbered from 0 to 15, and you are normally assumed to be user 0. If you want to refer to a file in a different user part of the disc, you can switch parts by typing:

```
|USER,n
```

where n is the number of the user part.

Under CP/M Plus on the CPC6128, you can assign passwords to each user, so that several people may use the same disc, without being able to see or use files in other user parts.

6128 SPECIAL

Formatting a Disc

The main differences between the CPC464 and CPC664 micros and the CPC6128 are the extra 64K of memory and the CP/M Plus (also known as CP/M 3.1) operating system. The advantages of the extra memory will be discussed later in the book, but you will come across CP/M Plus as soon as you try to format a disc.

The CPC6128 is supplied with two master discs, one containing CP/M 2.2 and the same subset of DR Logo as provided with the DDI-1 and CPC664, and the other with the more sophisticated CP/M Plus and a full implementation of Logo.

Amstrad has provided an easy-to-use utility program for formatting and copying discs on the CPC6128; the DISCKIT. This program is supplied in

two versions, DISCKIT2 which works under CP/M 2.2, and DISCKIT3, which works under CP/M Plus. Both versions offer a series of menus from which you select your required function. DISCKIT allows you to format discs in any of three formats, and on single or dual drive systems.

There is a step-by-step tutorial on how to use DISCKIT in Chapter 1 of the 6128 User Manual, but a brief summary of its use is included here. This description assumes you're working with a single-drive system.

If you are reformatting a disc which already contains programs or data, remember that they will all be wiped from the disc during formatting. Indeed, this is one of the most effective ways of clearing a disc of all files.

Start by loading your CP/M master disc, either side 1 for CP/M Plus or side 4 for CP/M 2.2, and typing:

`DISCKIT3` (`DISCKIT2` for CP/M 2.2)

The main DISCKIT menu will then be displayed and you can select the formatting option by pressing function key f4. Only the function keys work; pressing the number 4 on the top row of the keyboard will only produce a beep from the computer.

The next menu to be displayed offers the three different formats you can choose for your disc. The system format copies the CP/M system onto the disc, as well as formatting it. This means you can load CP/M from AMSDOS on the formatted disc and use any of the resident functions, described in Chapter 2.

The second format is for data only. This option will format the disc as before, but won't copy the CP/M system onto the disc. Although you will have to start CP/M from another disc, formatted as a system disc, a data format disc has the advantage that you can use nearly the full 180K for storing your data. This is probably most useful if you're using a word processor with two disc drives. You can then load the word processor from a system disc in one drive and have all your text documents on a data disc in the other.

The third format is known as vendor format, and is the way most commercial programs will be supplied. This format has the same amount of space available as the system format, but the CP/M system is not copied into the two tracks on the disc normally reserved for it. The reason for this is that each copy of CP/M is separately licenced to the individual purchaser. A software company selling you a program should not, therefore, supply a copy of its own CP/M to you.

To make a working copy of a vendor disc you should format a blank disc in system format, and copy the files from the vendor disc onto it. This is most easily done using the copy routine within DISCKIT. This program will format the blank disc automatically before copying.

Calling Utilities from AMSDOS

You can call the erase and renumber utilities from within AMSDOS on the CPC 6128 as you can with the DDI-1 extensions. As already described, this involves putting filenames and extensions into string variables and then calling the function with a '@' suffix to the variable name.

On the CPC 6128, however, you can simplify the calls by using the filenames directly. Thus, in the three examples quoted earlier, you could type:

```
|ERA,"program"
|ERA,"ATTEMPT?.BAS"
|REN,"new-prog","old-prog"
```

instead of the more convoluted commands of the earlier version of AMSDOS.

This short introduction to discs will hopefully have whetted your appetite for more. The next chapter looks at CP/M, the operating system which has been described as the most irritating and the most important piece of software in the history of the microcomputer!

Chapter 2

CP/M

A brief history

This is the second operating system available to you as a user of the Amstrad disc system. The initials stand for 'Control Program / Microcomputers' and CP/M is by far the most widely-used operating system available today.

CP/M was developed over ten years ago by Dr Gary Kildall in the United States. He designed it to handle all the straightforward functions in a microcomputer based on the 8080 family of microprocessor chips. Thus, in computers using CP/M, functions such as reading the keyboard to check if a key is being pressed, or sending a character to the screen or the printer are all handled by the operating system. These functions are often referred to as 'housekeeping' and, although very important to the running of the computer, are usually taken for granted by the person using it.

All the Amstrad computers use a Z80 microprocessor, which is a more advanced version of the 8080 chip, and CP/M is therefore a good choice of operating system for Amstrad's disc drives.

But why didn't Amstrad make AMSDOS do all the work that is, in fact, handled by CP/M? The main reason CP/M has become so popular is that it is designed to be very adaptable. Literally any computer using the right type of microprocessor can use the system. This flexibility has led to the use of CP/M in many (close to 100) different computers. This popularity has produced a snowball effect. Companies writing software have chosen to write it to work with CP/M so that they can sell it for any computer which runs this operating system.

Computer manufacturers launching new computers have looked at the available software and noticed how much of it works with CP/M. To ensure their new machines have immediate access to a large range of programs, they have chosen to install CP/M on them. And so it goes on... Amstrad have continued the trend by adopting CP/M as the main operating system for their disc drives. In doing so, they've also made it available on computers which are significantly cheaper than any it has been used on before.

AMSDOS is really the part of the disc operating system which is compatible with the version of BASIC used on the 464/664. When you forsake BASIC for other things (different languages or application programs) you start to work in the abbreviated world of CP/M.

When you come to use CP/M, one of the first things you notice is that it seems to be full of mysterious sets of initials, which at first glance appear pretty meaningless. This chapter should help to explain some of these abbreviations — there is an historical reason for them.

When CP/M was first written, discs were a very new medium for storing computer information. Only a few years before, anybody trying to use a computer in the home would have had to rely on punched paper tapes, or large reels of magnetic tape. The first floppy discs held only a small amount of data. The cost of computer memory was high and any useful operating system had to take as little room on disc and in memory as possible. Add to this the American penchant for over-complicated names and acronyms, and you may begin to understand how CCP, PIP and BIOS (among others) came about.

By keeping all instructions to the operating system and all messages from it very short, valuable room has been saved. Even though these restrictions have since become less important, CP/M has retained the abbreviations to ensure continued compatibility.

With the Amstrad systems, CP/M 2.2 takes up only 10.5K of the computer's memory. It needs this space to store what is called the CCP, or Console Command Processor. This is the program which interprets the commands you give the computer from the keyboard, and handles such things as reading the directory of a disc and calling up a file from it. The routines which do this work are known as *resident* routines.

Transient programs, on the other hand, are called into memory only when they're needed. When they've finished their function they're overwritten by another routine. The formatting routine described in Chapter 1 is an example of a transient program.

GENERAL PURPOSE CP/M COMMANDS

Let's have a look now at the routines which CP/M provides on the Amstrad micros. Some of them are resident programs, while others are transient.

The CP/M programs which are not specific to the Amstrad system are **DIR, ERA, MOVCPM, REN, STAT** and **TYPE**. Any CP/M utility, whether it's loaded from disc before running or held in the CP/M area of the computer's memory, is called directly by typing its name. Several of them are very similar to their AMSDOS equivalents which were described in Chapter 1.

DIR

This command will display the directory of the currently selected drive. Each program name, with its filetype extension, is listed in the order that it's found in the disc's directory. If you follow the command with `B:`, CP/M will try and read the directory of drive B. If you only have one drive connected, the error message:

```
Drive B: disc missing
Retry, Ignore or Cancel?
```

will appear. There is no point retrying the command in this situation, and ignoring it will crash the system (you'll have to restart from scratch), so you should type `C`, for Cancel, in answer to the prompt. A second message, `Bdos Err On B: Select` will then appear. Pressing any key will clear this and return you to the `A>` prompt.

What is happening is that CP/M has gone away and tried to read the directory of the non-existent drive B, and has then reported that it's missing. An Amstrad routine has intercepted the normal CP/M message (the rather cryptic `Bdos Err On B: Select`) and replaced it with the more helpful `Retry, Ignore or Cancel?`. If you had a second drive, but had just forgotten to switch it on, you could have done so and then typed `R` for retry. When you tell CP/M to cancel the command, the original error message comes through.

`Bdos Err On B: Select` means that CP/M has detected some sort of error in the way you are trying to access a disc, using the part of CP/M known as Bdos. Bdos stands for Basic disc operating system, and is one of the abbreviations for which CP/M is reknowned. The error is in the selection of drive B (because it's not there!)

ERA

This command is short for ERAse, and as you might imagine, is used to remove unwanted files from a disc. The format of the command is:

```
ERA FILENAME.TYP
```

CP/M will respond to the command by searching the disc directory for the given filename and filetype extension, and will then remove both from the directory. As with the AMSDOS command of the same name, you may use wild cards to delete all files with a specific filetype, or with similar names. the filename `*.*` will remove all files on the disc, but only after you've confirmed that this is your intention, by answering the prompt:

```
ALL (Y/N)?
```

with a Y. You should obviously be cautious when using this command, as deleted files are not easily recovered (see Chapter 4).

MOVCPM

You'd have thought, even with memory space at a premium, that the 'E' in the middle of this command could have been left in! The command will move the resident part of CP/M from the top of memory, where it normally lives, to a specified *page boundary* lower down.

The memory of your CPC464 or CPC664 is divided into a number of notional *pages*, each 256 bytes long. This is a convenient sub-division for handling large amounts of memory. A page boundary, as you might expect, is the imaginary line dividing two adjacent pages. CP/M can begin in the first byte of any new page, and this is referred to as being positioned at the page boundary.

In the 64k Amstrad machines you can move CP/M down by as much as 116 pages, or 116/4=29K, and this is done by specifying a parameter in the range 64 to 179 after the MOVCPM command. When you load CP/M from the disc supplied with your Amstrad Micro, it automatically loads at the top of memory, which is equivalent to a parameter setting of 179. This is known as the default setting.

You may want to store programs or data at the top of memory, unseen by CP/M, and to do this you would need to use MOVCPM to reposition the operating system lower down. For example, the command:

```
MOVCPM 163 *
```

would move CP/M down by 16 pages (4K), and prompt for you to save the new version as a separate file on the disc. If you omit the asterisk, then this prompt won't be displayed.

On typing `MOVCPM 163 *`, CP/M will display:

```
CONSTRUCTING 40k CP/M vers 2.2
READY FOR "SYSGEN" OR
"SAVE 34 CPM40.COM"
```

If you want to save the new version of CP/M on the disc in the same way as any other program, then you type `SAVE 34 CPM40.COM` as instructed by the prompt. If, instead, you want the new version to become the default version on that disc, you should use SYSGEN to copy the 40K version you've just generated onto the system tracks of the disc. Refer to the section on SYSGEN (page 26) for an explanation of this process.

REN

This command is again very similar to the AMSDOS command of the same name. As with ERA, though, you enter the two filenames directly, rather than first having to assign them to variables. The new filename must be different from any already on the disc, otherwise CP/M will respond with the message `FILE EXISTS`.

The format of the REN command is:

```
REN FILE2.TYP=FILE1.TYP
```

and is again similar to the LET command in BASIC.

STAT

This command, which is short for STATus, will display a variety of information about a disc and the files stored on it. In addition, it can be used to 'lock' a file so that it can only be read and not written to, or to hide it from the directory command and file-copying routines in CP/M.

In its simplest form, typing `STAT` will return a display similar to:

```
A: R/W, Space: 40k
```

This tells you which disc drive you're using (A:), whether the disc may be read and written to (R/W) or read only (R/O), and the amount of space remaining on the disc (40K).

If you just want to know how much space is left on the disc, or on either disc if you have twin drives, you can add the drive letter and a colon to the command, `STAT A:`, and the display will show:

```
Bytes Remaining on A: 40k
```

If you add a filename to the command, CP/M will show you additional information about the file. For instance, if you type `STAT ED.COM` with a copy of Amstrad's CP/M disc in your disc drive, you will see:

```
Recs  Bytes  Ext  Acc
 52    7k     1   R/W A:ED.COM
Bytes Remaining on A: 40k
```

displayed on your monitor.

The four headings stand for Records, Bytes, Extents and Access. The first three show the length of the file and how it's stored on the disc. The Access

heading shows the read and write status of this particular file.

You can use wild cards with the STAT command, so `STAT *.*` will display information on all files on the disc. The files will be listed in alphabetical order.

STAT can be used to list all manner of other information about your CP/M disc system, but most of this will only be of use to you very occasionally. STAT can be used to change certain features of the files on your disc, however, and these will be more frequently used.

If you follow the filename in a STAT command with `$R/O`, you will set that file to 'read only' status, so that you can't inadvertently erase it or overwrite it with another file of the same name. The new status will last even if you switch off the system and start it up again later. A file can only be accessed again if the STAT command is reversed by typing: `STAT A:R/W`.

The second alteration you can make to a file using the STAT command is to remove it from the directory, by making it a 'system' file. This is useful if you want to try and hide some of the files on a disc. You might want to do this, for example, to help prevent illegal copying of programs or data held on the disc. To remove a file with the filename `FILENAME.TYP` from the directory, you use the command:

```
STAT A:FILENAME.TYP $SYS
```

You can reinstate the file in the directory with the command:

```
STAT A:FILENAME.TYP $DIR
```

The only way you can tell what system files are on a disc is to list its contents with `STAT *.*`. All the files will then be listed, with system files shown bracketed.

TYPE

This command will display the contents of any ASCII file on the disc. You can use it to look at BASIC programs, the source files for assembler programs or language compilers, or a text file from a word processor.

ASCII stands for the American Standard Code for Information Interchange. It is a system of coding all the alphabetic, numeric and punctuation characters by giving them numbers between 0 and 255. Any file which makes use of this coding can be translated by the TYPE command to show the equivalent characters.

Beware that files with filetypes `COM` or `BIN` are not in ASCII form, and

trying to display their contents with TYPE can cause the computer to crash. If you must have a go, and you do crash the computer, you can of course restart it by pressing <CTRL>, <SHIFT> and <ESC> or, if that doesn't work, switching everything off and then on again.

AMSTRAD-SPECIFIC CP/M COMMANDS

The CP/M utilities already described are the same as on any other machine running this operating system. There are a number of other routines running on Amstrad's version of CP/M, however, which have either been specifically written by Amstrad to work with CP/M, or are CP/M utilities, tailored to run correctly on Amstrad computers.

AMSDOS

The disc commands available within BASIC are designed to be totally compatible with the CP/M operating system. The AMSDOS operating system can be called directly from CP/M simply by typing its name, and may therefore be thought of as a CP/M utility, calling up a disc version of BASIC.

BOOTGEN

This is a program provided by Amstrad which allows you to convert a disc without a configuration sector in to one which has. The configuration sector of a disc holds all the information selected using the SETUP command (see page 21).

An Amstrad disc sold to you with a CP/M program on it will normally not have CP/M, or your own configuration (if you've set one up using the SETUP command) stored on it. You will therefore need to copy both CP/M and the configuration onto a working disc which already contains a copy of the program from the disc you've bought. The BOOTGEN command copies any special configuration from one disc to another.

When you type the command, you will be prompted with:

```
Please insert SOURCE disc into drive A then press
                    any key:
```

When you've done this, a similar message prompts for the destination disc, and the configuration is copied. You can repeat the process if you have more than one disc to convert.

CLOAD

This utility is used to copy files from cassette to disc. The only sensible use

for it is in transferring ASCII or Basic files. The most obvious ASCII files you might want to transfer are text files from word processors such as Amsword or Easi-Amsword.

The form of the command is:

```
CLOAD "FILENAME" FILENAME.TXT
```

and once you've typed this, you will be prompted to start the tape. You should wind it through to a position just before the file you wish to copy, and then press <PLAY> and any key on the keyboard. The file **FILENAME** will then be loaded from cassette and transferred to disc with the name **FILENAME.TXT**.

You can omit the second parameter if you want the disc file to have the same name as the cassette file. If you omit both, the routine will copy the first cassette file it comes to. The inverted commas enclosing the cassette filename are essential.

You won't be able to transfer a cassette file if it was stored with a P (for protected) suffix in the filename.

CSAVE

This routine does exactly the opposite of CLOAD, and transfers a disc file onto cassette. Again it's only really suitable for ASCII or Basic files. Its main use would be to keep back-up copies of important documents without wasting disc space.

The first two parameters are the two filenames and the third is the recording speed of the tape file. If this parameter is 0, the file will be recorded at 1000 baud, if it's 1 it will be recorded at twice this speed. Note that the cassette filename must again be enclosed in inverted commas, and that if the first character of the cassette filename is **!**, then no cassette messages will be displayed. A typical command might be:

```
CSAVE FILENAME.TXT "!FILENAME" 1
```

CHKDISC and DISCCHK

These two routines are provided to check two discs and verify that they contain exactly the same files. DISCCHK is for use on a single drive system, while CHKDISK is for twin drives. The DISCCHK routine will be described here.

If you have a copy of a disc and you want to check that it's the same as your original, load a copy of the CP/M system disc and type **DISCCHK**. You will

then be asked to repeatedly insert the source and destination discs (the original and your copy) and they will be compared, eight tracks at a time. If any differences are found, the message:

```
Failed to verify the destination disc correctly:
                track XX sector YY
```

will be displayed, and a warning message:

```
WARNING: Failed to compare discs correctly
```

will be given at the end of the comparison.

If you are using twin drives, CHKDISC performs exactly the same comparison, but with the source disc in drive A and the destination disc in drive B. The whole process is carried out automatically.

The main reason for comparing two discs in this way is if you suspect one of them has become corrupted.

COPYDISC and DISCCOPY

These two utilities copy the contents of one disc onto another. Again, DISCCOPY is for a single drive system and COPYDISC is for twin drives. The operation of both routines is very similar to the checking routines just described.

You first load either COPYDISC or DISCCOPY from your CP/M system disc, and you will be prompted to insert the source and destination discs, either one into each drive (COPYDISC) or alternately while the copying process is underway (DISCCOPY). Once the copying is complete, you have the option to copy further discs or to return to CP/M.

Beware that both of these routines will wipe the contents of the destination disc as they copy. If you want to preserve files already stored on the disc, you should use the FILECOPY routine instead.

FILECOPY

Filecopy will copy a named file from one disc to another. You load the routine from your CP/M system disc by typing its name together with the name of the file you want to copy:

```
FILECOPY FILENAME.TYP
```

It's important to remember the filetype extension, (if there is one).

Once the routine has been loaded, you're prompted to insert the source disc (the one you want to copy *from*). When the file has been loaded, you are asked to load the destination disc (to copy *to*), and the copy will then be made. You can repeat the process for further files.

The advantages of FILECOPY over DISCCOPY are that you can specifiy which files you want to copy, and that the files will be added to the destination disc *without* affecting any files already on that disc.

FORMAT

This routine, whose use was described in detail in Chapter 1, sets up a blank disc so that it can be used for storing either AMSDOS or CP/M files.

SETUP

This utility is used to define a number of features of your Amstrad computer and disc system. In most circumstances you should find that the default values already programmed in are correct. There are certain times, though, when you might want to change some of the values. For instance, you might want to change the ASCII codes produced by some of the keys, to make a particular program easier to use. We'll run through each of the categories, and a useful set of key definitions.

When you run the program, the first question will be:

```
** Initial command buffer empty

Is this correct (Y/N):-
```

The initial command buffer is a section of the CP/M program recorded on a particular disc. It is normally empty, which means that when you call CP/M, you will just be prompted with A>, and left to select a program or utility of your choice. If you want to, however, you can force CP/M to execute a set of commands every time the disc is loaded.

For example, you might want to run a particular program each time you load a given disc. To do this, you enter a string of commands into the initial command buffer. To make the disc automatically display its directory when you load it, you would do the following. First reply **N** to the above question. The display will then show:

```
Enter new initial command buffer:-
```

Type in **DIR^M** and press <ENTER>. When SETUP finishes, the CP/M part of your disc will be reprogrammed with the new command. When you load the disc and type **|CPM** from within AMSDOS, the revised version of CP/M will be loaded and the directory of the disc will be displayed automatically.

The ^ M the end of the command you type into the initial command buffer, stands for <CTRL>M. The ^ symbol stands for <CTRL> in a program. If you refer to the Appendices of your *Amstrad Manual*, you will see that all the ASCII characters from 0 to 26 can be obtained by typing <CTRL> along with a letter of the alphabet.

<CTRL>M, ASCII value 13, produces a carriage return, which is the same as pressing the <ENTER> key on the keyboard. Other useful ASCII codes are 10 (<CTRL>J) — line feed, 12 (<CTRL>L) — form feed and 7 (<CTRLL>G) — bell, which produces a beep from the loudspeaker.

The next option in the SETUP program is to alter the 'Sign-on' string. This string of characters governs the colours of the display, and the message which is displayed when you load CP/M from disc. You would not normally want to change the message (unless you want to add your name and address to it for security!), but you might find it useful to alter the colour scheme. The default Sign-on is:

```
^ \@ww^ a@@^ ]wwCP/M 2.2 - Amstrad Consumer
Electronicsplc^J^M
```

Each of the characters preceding the message CP/M 2.2, etc represents a command or value governing the colours used in the screen display. The first symbol (^), as already explained, stands for <CTRL>. The backslash (\ ) immediately following has a value of 28, as it is the second character after Z (the 26th character of the alphabet) in ASCII order.

Don't worry if you can't follow that, all that really matters is that its value is 28, which corresponds to the SET INK command. The @ represents the number of the pen ink being set, and the two ws are the values of the two pen ink colours. The next group of characters ^ /a@@ does the same thing for the paper ink colours, and the final group ^]ww set the border colour. If you want to reset the colours used in the CP/M screen display, then you can use the table in Figure 1. to select them.

Having decided what colours you want, the next option in the SETUP utility is the 'Printer power-up' string. This is normally left blank, and if you're using Amstrad's DMP-1 printer you won't need to put anything in here. Some other printers, however, particularly daisywheel ones, require a sequence of <CTRL> or <ESC> codes to set them up properly.

For instance, the Daisystep 2000 printer needs the sequence <ESC>S sent to it to select bi-directional printing. By setting up the power-up string as ^ [S, this will be done automatically when you call CP/M on that disc. Bear

in mind, though, that you must switch your printer on and select it *before* calling CP/M, if you use a power-up string, otherwise CP/M won't load properly.

Colour	Colour No.	Letter	Colour	Colour No.	Letter
Black	0	@	Pastel blue	14	n
Blue	1	a	Orange	15	o
Bright blue	2	b	Pink	16	p
Red	3	c	Pastel magenta	17	q
Magenta	4	d	Bright green	18	r
Mauve	5	e	Sea green	19	s
Bright red	6	f	Bright green	20	t
Purple	7	g	Lime green	21	u
Bright magenta	8	h	Pastel green	22	v
Green	9	i	Pastel cyan	23	w
Cyan	10	j	Bright yellow	24	x
Sky blue	11	k	Pastel yellow	25	y
Yellow	12	l	Bright white	26	z
White	13	m			

Fig 2.1 Table of Amstrad colour codes and corresponding ASCII characters

The next option SETUP offers is to produce keyboard translations. This means that you can define keys on the keyboard to produce codes other than their normal ones. This facility can be very useful. For example, the word processor WordStar, which has proved extremely popular, uses a number of <CTRL> sequences to call its many functions. This system of <CTRL> sequences was designed so that the word processor could be operated using any standard keyboard.

The keyboard on Amstrad micros, however, has a set of arrow keys which are much easier to use for moving the cursor than the <CTRL> sequences. It's a simple job to redefine the arrow keys to produce these sequences directly. The Hisoft text editor, ED80 (described in Chapter 4), uses the same sequences as WordStar, so these translations would work just as well with this program.

The translations are:

Key code	*Normal*	*Shift*	*Control*
0	5	18	0
1	4	6	0
2	24	3	0
8	19	1	0
9	17	0	0
16	7	20	25

These translations produce the following actions in WordStar or ED80:

	Alone	*+<SHIFT>*	*+<CTRL>*	*Preceded by <COPY>*	*Preceded by <COPY> and +<SHIFT>*
↑	1 line	1 screen	—	top of screen	top of text
↓	1 line	1 screen	—	bottom of screen	bottom of text
←	1 char	1 word	—	start of line	—
→	1 char	1 word	—	end of line	—
CLR	1 char R	1 word R	whole line	—	—
DEL	1 char L	1 char L	—	—	—

After keyboard translations come keyboard expansions. This option of the SETUP program performs a similar function to BASIC's KEY command. You can redefine any of the function key strings normally assigned to the numeric key pad, for the keys on their own, with <SHIFT> and with <CTRL>. You can redefine these keys to produce a string of characters of your choice.

You might find it useful to program a set of functions such as the following:

Function key token	*String*
0	DIR M
1	REN
2	ERA
3	AMSDOS M

These settings will make the keypad keys '0' to '3' produce the corresponding CP/M commands. Note that the first and last will execute the command, while the middle two will just print the command name. This is because you'll need to add filenames to the REN and ERA commands. You can add further strings as you wish, up to a maximum of 32.

The remaining options within the SETUP program are for specialist use and would not be of much use in normal operation of a disc-based Amstrad micro. For the sake of completeness, this is what each does:

I/O byte settings

There are four *logical* devices within CP/M, and each is assigned to a *physical* device, so that a number of different peripherals may be connected to the system.

The logical and physical devices are:

Logical devices	*Physical devices*
CON: (CONsole)	TTY: (TeleTYpe)
RDR: (ReaDeR)	CRT: (Cathode Ray Tube/keyboard)
PUN: (PUNch)	BAT: (BATch processing)
LST: (LiST device)	PTR: (Paper Tape Reader)
PTP: (Paper Tape Punch)	LPT: (Line PrinTer)

In addition, there are a number of user-defined physical devices (e.g. UC1:, UP2:). The only assignments allowed are:

CON:	TTY:	CRT:	BAT:	UC1:
RDR:	TTY:	PTR:	UR1:	UR2:
PUN:	TTY:	PTP:	UP1:	UP2:
LST:	TTY:	CRT:	LPT:	UL1:

Defaults:

Slow mode — When your computer is running programs in machine code, it can either maintain the secondary register set of the Z80 microprocessor or not. If you are fully aware of what you're doing, you can select fast mode, which will not maintain the secondary registers. You are strongly recommended to leave this option alone under normal use.

BIOS messages — These are the rather cryptic error messages from the Basic Input/Output System, e.g. **?** when a command isn't understood.

Clear initial command buffer on keyboard input — If you have set up an initial command (see above), then if this option is set you can override the buffer by typing characters from the keyboard as CP/M is loading.

Motor on delay — The length of time allowed for the disc to reach speed before data is read to or from it.

Motor off delay — The length of time the disc is kept spinning after data has been read from or to it.

Stepping rate — The rate at which the drive head is moved across the disc surface.

Z80 SIO Channel A — These parameters control the function of Serial Input/Output channel A. They are correctly set by default to work with Amstrad's RS232 interface unit, but may be changed if you're using another manufacturer's interface.

Z80 SIO Channel B — These parameters control the function of Serial Input/Output channel B as above.

It is sensible to leave these default settings unaltered, unless you are fully aware of what you're doing.

SYSGEN

This utility is used to copy CP/M from one disc to another, and to copy a revised version of CP/M onto the *system tracks* at the start of the disc. The first use is very staightforward. Type **SYSGEN** and you will be prompted for source and destination discs while the copying is in process. If you have altered the set-up options within your own version of CP/M, using the SETUP utility just described, then SYSGEN is very useful for transferring the new version to other discs.

You don't have to have a copy of CP/M on each of the discs you use, but remember that you can't load the operating system (and get back to the '>A' prompt) without it being on the disc.

The second use of SYSGEN is to load an amended version of CP/M which you have created with the **MOVCPM** command, to reserve space in memory, 'unseen' by CP/M. This is done by adding the filename of the revised CP/M to the end of the command. Thus, if you had produced a version of CP/M to run in only 32K of memory, and saved it on the disc as 'CPM32.COM', you could produce a disc to work with this configuration by typing:

```
SYSGEN CPM32.COM
```

6128 SPECIAL

There are a number of extra utilities provided with CP/M Plus on the CPC6128. CP/M Plus itself is a lot more comprehensive an operating system than CP/M 2.2, which is what you might expect when you realise that it occupies some 25K of RAM memory, against the 10.5K of CP/M 2.2.

Some CP/M Plus utilities use different names from their CP/M 2.2 counterparts, and require you to use different techniques to achieve the same result. For instance, there are no FILECOPY or DISCCOPY programs on the CP/M Plus master disc, and no COPYSYS program is provided to copy the CP/M system from one disc to another. The DIR command has extensions in CP/M Plus which take over some of the work done by STAT under CP/M 2.2. Before describing the new utilities available under CP/M Plus, therefore, it's useful to run through the changes to the CP/M 2.2 routines already described.

The DIR command in CP/M Plus still displays the disc directory, but you can now add one or more extension to the command, to display the size of each file, its write-protection and password status and any date-stamping applying to it. The ideas of file security and date-stamping are new with CP/M Plus and are described further in the section on the SET command.

Perhaps the two most useful extensions to the DIR command are the SIZE

and FULL options. CP/M Plus command options are typed after the command and are enclosed in square brackets.

The command:

```
DIR [FULL]
```

used with the CP/M Plus master disc in the drive would produce the display shown in Figure 2.2.

Name		*Bytes*	*Recs*	*Attributes*	*Name*		*Bytes*	*Recs*	*Attributes*
AMSDOS	COM	1k	8	DirRW	BANKMAN	BAS	1k	7	DirRW
BANKMAN	BIN	2k	12	DirRW	C10CPM3	EMS	25k	200	DirRW
DATE	COM	3k	23	DirRW	DEVICE	COM	8k	58	DirRW
DIR	COM	15k	114	DirRW	DISCKIT3	COM	6k	48	DirRW
GED	COM	10k	73	DirRW	ERASE	COM	4k	29	DirRW
GET	COM	7k	51	DirRW	KEYS	CCP	1k	3	DirRW
KEYS	WP	1k	3	DirRW	LANGUAGE	COM	1k	8	DirRW
PALETTE	COM	1k	8	DirRW	PIP	COM	9k	68	DirRW
PROFILE	ENG	1k	1	DirRW	PUT	COM	7k	55	DirRW
RENAME	COM	3k	23	DirRW	SET	COM	11k	81	DirRW
SET24X80	COM	1k	8	DirRW	SETDEF	COM	4k	32	DirRW
SETKEYS	COM	2k	16	DirRW	SETLST	COM	2k	16	DirRW
SETSIO	COM	2k	16	DirRW	SHOW	COM	9k	66	DirRW
SUBMIT	COM	6k	42	DirRW	TYPE	COM	3k	24	DirRW

Total Bytes =146k Total Records =1093 Files Found=28
Total 1k Blocks =146 Used/Max Dir Entries For Drive A:29/64

Figure 2.2 Display of disc directory, using DIR [FULL] command

Notice that the files are sorted into alphabetical order, and that several other pieces of information about the status of the disc are included in the display.

The DIR [SIZE] command shows just the lengths of each file without the record or attribute information. The directory is still sorted, though.

The ERA utility will still erase files and you can use the ? and * wildcards as with the CP/M 2.2. If you're using wildcards to specify a set of similarly named records, however, you can add the [CONFIRM] option to the end of the ERA command. This forces CP/M Plus to prompt you with each filename in turn and wait for confirmation before deleting it.

The MOVCPM command isn't available in CP/M Plus, as this operating system assumes a micro with at least 64K of memory. The CPC 6128, of course, has 128K available.

You can use REN to rename a CP/M Plus file, as with CP/M 2.2. With the new routine, though, you can also rename a series of files using wildcard descriptions.

For example, the command:

```
REN FRED.*=TEST.*
```

would rename all the files on a disc with a filename TEST and any filetype, giving them the same filetype as before but the new filename FRED. If you have several files with similar names this wildcard facility can save you quite a bit of work.

The STAT command is also not available in CP/M Plus, but is replaced by the new options on the DIR command, and by the SET and SHOW commands.

You can display ASCII files with the TYPE command as before, but you can now choose to display a page (24 lines) at a time by adding the [PAGE] option after the command. This is useful if you have a long file to display on the screen.

You may wonder how these extra facilities can be built into CP/M Plus commands. The answer is that the resident commands DIR, ERA, REN and TYPE all make use of files called from the system disc. Although DIR and TYPE will still perform their basic functions, ERA and REN won't work at all without the file of the same name on the current disc, and DIR and TYPE won't offer their extra features.

The extra utilities provided by CP/M Plus deal with status information about the system and customising the keyboard, display screen, printer and serial I/O port. Taking them in alphabetical order:

DATE

This routine allows you to enter the current date and time, and then call it from the 'A>' prompt. To set the date and time, which you will need to do at the start of each work session if you want to use the function, you type:

```
DATE SET
```

and then answer the prompts:

```
Enter today's date (MM/DD/YY)
Enter the time (HH:MM:SS)
Press any key to set time
```

As you can see, the counter doesn't start until after you enter the time, so the best method is to set the displayed time a minute or so ahead of the actual

time and press a key when the displayed time is reached. Notice also that the date is set in the American format, with the month before the day.

DEVICE

This command takes over some of the work done by STAT or SETUP under CP/M 2.2. Using DEVICE, you can display and set logical to physical device assignments. Typing DEVICE on its own will display something like:

```
Physical Devices:
I=Input, O=Output, S=Serial, X=Xon-Xoff
CRT   NONE   IO   LPT    NON O    SIO   9600
IOS

Current Assignments:
CONIN:    = CRT
CONOUT:   = CRT
AUXIN:    = SIO
AUXOUT:   = SIO
LST:      = LPT

Enter new assignment or hit RETURN
```

The physical devices known to CP/M are listed first (these may be displayed on their own by typing **DEVICE NAMES**). The abbreviated name, speed of communication (baud rate), and type of communication are listed for each physical device.

The current assignments of physical devices to CP/M's logical devices are displayed below this (these may be displayed separately by typing **DEVICE VALUES**).

Finally, you're prompted to enter a new assignment (or to revise an existing one) by entering it in the form:

```
logical-device=physical-device
```

You might, for example, type:

```
LST:=CRT
```

to force printed output to the screen. If you want to disconnect a logical device from its physical assignment, you simply replace the physical device name with 'NULL', as in:

```
LST:=NULL
```

You can also type **DEVICE** followed by a logical or physical device name, in which case all its attributes are shown. Any of these attributes, such as the baud rate of the serial port (SIO), can be altered by enclosing the revised value in square brackets after the command.

An interesting example of this last facility is a command like:

```
DEVICE CONSOLE[COLUMNS=40,LINES=16]
```

which will reduce the CP/M screen to a 40x16 character area.

GET

This command tells CP/M to take input from a named file, rather than from the keyboard. Although in some ways similar to the **SUBMIT** command, **GET** can transfer control *after* loading an application program, as well as executing a series of CP/M commands from a file.

The ability to delay the transfer of control is useful in several ways. You may, for example, want to load a program which requires you to enter repetitive data into it. You can set up a file with this data in it, and then use GET to instruct CP/M to take data from that file, after loading and running the program.

To execute a series of CP/M commands from a file, you add the option 'SYSTEM' to the GET command. If your file of commands is called 'SETUP', the GET command to run it would look like this:

```
GET FILE SETUP [SYSTEM]
```

Notice that you have to use the word FILE in the command. You can choose not to have each command in the file echoed to the screen by adding the option [NO ECHO].

To redirect control back to the keyboard you use the command:

```
GET CONSOLE
```

HELP

The help utility is not so much a CP/M command as a 'manual-on-a-disc'. By inserting the master disc with the help system on it and typing **HELP,** followed by the name of a CP/M Plus command, the chances are that you'll be presented with a screen of information on that command, including examples of its use. If you type **HELP** on its own, a list of all the available commands is displayed.

While only CP/M Plus .COM files are detailed in this way, and the

descriptions supplied tend to be quite technical, its very useful to be able to obtain information 'on-line'.

INITDIR

This command is used to set up a CP/M Plus disc so that the directory can include date and time 'stamps'. These stamps are simple records of the date and time that you save each file on the disc. Obviously you have to set up the correct date and time at the start of each work session, before the stamping will work. You do this with the DATE SET command.

The disc doesn't have to be freshly formatted, but if there are files on it, CP/M will check that there's sufficient directory space before reformatting the directory.

To call the command you type its name followed by the disc letter. For example:

```
INITDIR A:
```

LANGUAGE

This command changes some characters in the standard (American) character set, to suit the language requirements of particular countries. There are seven alternative character sets, which are called by typing `LANGUAGE n.` n is a number between 1 and 7, and selects a language according to this table:

Value of 'n'	*Language*
1	French
2	German
3	UK English
4	Danish
5	Swedish
6	Italian
7	Spanish

For users in the UK, typing LANGUAGE 3 will change the hash character, '#' (ASCII 35), to the pound sterling sign, £. The other language options change several more characters on screen.

If you want to print out documents using any of the alternative character sets, you will of course have to select the appropriate option on the printer, as well. This is often achieved by flicking a small switch inside its case.

PALETTE

As you may guess from the name, this command allows you to select the colours of your CP/M Plus display. In mode 2 (80 columns), which is the default mode for CP/M, only two colours can be displayed at once, and these normally use ink 0 for the background and ink 1 for the foreground.

The palette command allows you to alter the colours of any or all of the 16 inks available on the CPC6128. The command takes the form:

PALETTE ink0,ink1...ink15

where ink0, ink1 etc are the numbers representing the colours you want to assign to each ink, and their position in the list is the number of the ink. Thus the command:

```
PALETTE 0,63
```

would set ink 0 (the first figure in the list) to colour 0 (black), and ink 1 (the second in the list) to colour 63 (bright white).

PROFILE

Unlike the other commands listed so far, PROFILE is not a utility in itself. It is, in fact, the name of a special file which can contain a series of CP/M Plus commands. When you first switch to CP/M by typing `:CPM` from AMSDOS, CP/M Plus will be loaded, and will then search for a file called PROFILE.SUB. If it finds one, it will automatically execute any commands within the file, before returning control to the keyboard.

This is a very useful feature as it allows you to set up a series of commands which will adapt the operating environment to your own requirements. For example, if you have a disc with a word processor on it, you could create an effective PROFILE.SUB file by doing the following:

```
A>ED PROFILE.SUB
          NEW FILE
            :*I
          1:PALETTE 0,63
          2:SETKEYS KEYS.WP
          3:LANGUAGE 3
          4:WS
          5:(press <CTRL>Z)
            :*E

A>
```

This short sequence uses the CP/M Plus line editor ED (to be described in Chapter 3) to create a PROFILE.SUB file. This file will change the screen colours to bright white on black, set the cursor keys to produce sequences suitable for a word processor, change the character set to UK English and run the word processor WordStar.

PUT

This is a similar command to GET, except that it redirects *output*, either from the screen or from the printer, to a named disc file. The command has a similar syntax to the GET command, too:

PUT CONSOLE (or PRINTER) filename

will transfer output from the screen or printer to the file named as 'filename'.

You can return output to the normal output device with the command:

PUT CONSOLE (or PRINTER) CONSOLE (or PRINTER)

If you find this command confusing, you can put the optional words 'OUTPUT TO' between the source and destination for output.

SAVE

This utility saves a section of memory to a named disc file. To use the routine, you first load it by typing:

SAVE

You must then load the program to be saved into memory. The easiest way to do this is to load the debugger SID (see Chapter 3 for a description of this program) with the name of the file you want to save. If you were going to save PIP in this way, you would issue the command:

SID PIP.COM

SID is then loaded and itself loads PIP. You've now got three utilities in memory. Run SID by typing #g0. When SID has finished and tries to exit to the CP/M system, SAVE intercepts it and presents its own prompts:

```
SAVE Ver 3.0
File (or RETURN to exit)? filename
Delete filename? Y or N
From? start address
To? end address
```

The result of this little exchange is that the section of PIP between 'start address' and 'end address' has been saved to disc as a file with the name 'filename'.

SET

This command controls a number of related features of the disc and drive, depending on the options given.

On its own, with options provided within square brackets, [], SET controls the labelling, password protection and time stamping of files on disc. If you intend to use time stamps on your discs, you must run INITDIR on each disc first. The following examples show how SET works in this mode.

`SET [NAME=MYDISC]` labels the disc in the default drive, 'MYDISC'.

`SET [PASSWORD=XYZZY]` sets a password 'XYZZY' on the entire disc. Be careful to record the password, as you won't be able to use the disc without it. Press <RETURN> after PASSWORD to remove an existing password.

`SET [RO]` sets the disc to read-only status, so all write operations are prevented. [RW] resets the disc to read-write status.

`SET [CREATE=ON]` sets time stamping for each new file created.

`SET [ACCESS=ON]` sets time stamping for each access to a file, for whatever purpose. You can't have CREATE and ACCESS switched on together.

`SET [UPDATE=ON]` sets time stamping for each update (alteration) of a file. Each of these commands acts on the default disc drive, but if you have a dual drive system, you can add the drive letter (e.g. B:) between SET and the options in square brackets.

You can also use SET to alter the attributes of a particular file. You can make a file into a SYStem file, which removes it from the directory display, or make it read-only. There are a number of other attributes also available, and you can turn protection and time-stamping on and off for individual files. Here again are some examples:

`SET EXAMPLE [SYS]` Removes the file EXAMPLE from the DIRectory display, and makes it a SYStem file. Use SET EXAMPLE [DIR] to reinstate the file in the directory.

`SET EXAMPLE [RO]` Makes the EXAMPLE file Read Only. CP/M will prevent the file being overwritten. Use SET EXAMPLE [RW] to reset the file to Read and Write.

`SET EXAMPLE [ARCHIVE=OFF]` Stops CP/M automatically taking a back up of the EXAMPLE file, when changes are made to it. Back ups can be switched on again using SET EXAMPLE [ARCHIVE=ON].

`SET EXAMPLE [PASSWORD=SECRET,PROTECT=DELETE]` Sets up a password for the EXAMPLE file, and prevents deletion without first giving that password. PASSWORD can be any word you like, and PROTECT can be READ, WRITE, DELETE or NONE (which removes PASSWORD).

SETDEF

Like the SET command, SETDEF has several uses. Most of these are to do with loading CP/M programs and SUBMIT files, and the way they're displayed on the screen. There are a couple of other useful options, though.

The command:

```
SETDEF
```

on its own will produce a display like this:

```
Drive Search Path:
1st Drive                 - Default

Search Order              - COM

Temporary Drive           - Default

Console Page Mode         - On

Program Name Display      - Off
```

This display shows the current settings of all the facilities SETDEF controls.

When you issue a command to CP/M Plus to load a program, it will normally just search the directory of the default drive (drive A:), but you can change this with a command like:

```
SETDEF B:,A:
```

Any searching will now take place first on drive B:, and if the required file isn't found, drive A: will then be searched too.

Normally, you can only search for files with filetypes .COM or .SUB, and .COM files will be searched for first. However, the command:

```
SETDEF [ORDER=(SUB,COM)]
```

will change the order, so that .SUB files are searched for first.

You can also define a different filetype for executable files (.COM files). If you give all your files with the .COM filetype, a different filetype, such as .DOG, you can force CP/M Plus to search for them alone with the command:

```
SETDEF [ORDER=(DOG)]
```

CP/M Plus, and many application programs, make use of 'temporary' files while they're working. Normally, these files will be held on the default drive (A:), but you may want to alter this. On a twin drive system, for instance, you might want to hold your application program on a disc in drive A:, with all your data files, including the temporary ones, on drive B:. You can set this up with the command:

```
SETDEF [TEMPORARY=B:]
```

Under CP/M 2.2, programs which display a lot of information, such as DUMP, will scroll the screen automatically when its full (to stop it you type <CTRL>S). With CP/M Plus, things are a bit more civilised, and programs will normally only display a 'page' (24 lines) at a time, and then prompt you with the message:

```
Press RETURN to continue
```

If you want to turn this feature off, you can do so with the command:

```
SETDEF [NOPAGE]
```

and turn it back on again with:

```
SETDEF [PAGE]
```

When CP/M Plus loads an executable file, it normally just declares the name and version number of the program. You can make it display the drive letter, filename, filetype (if any) and user number as well, by first typing:

```
SETDEF [DISPLAY]
```

The command SETDEF [NO DISPLAY] reverses the process.

SETKEYS

Amstrad has provided a utility to set up the special keys on the CPC 6128 keyboard (such as the cursor arrows) to produce special control sequences suitable for three different applications.

These applications are CP/M itself, text editors and word processors which use WordStar commands (there are a lot!), and Dr Logo. To set up the keyboard using one of these files, you give the command:

```
SETKEYS filename
```

where the three filenames are KEYS.CCP, KEYS.WP and KEYS.DRL, respectively. The best way to set up key definitions is probably to call the SETKEYS command from within a PROFILE file on your working disc.

SETLST

This command is used to send control sequences to a printer to set up particular features of the machine (different fonts, overprinting etc).

The codes themselves are written into a file using ED or PIP (see Chapter 3), and the file is saved with a suitable filename. If you call the printer set up file PRINTER, you would give the command:

```
SETLST PRINTER
```

to send the series of codes to the printer.

The codes themselves are produced by pressing the <CTRL> key with another letter. The <CTRL> sequence is represented on the screen as '^'. See the entry for SETLST and SETLST.ESCAPE in the HELP system supplied on one of the master discs.

SETSIO

This command sets up the serial i/o port, which is also known as the RS232C port. The RS232C port is an extra unit which can be attached to the expansion port on the back of the CPC6128. Amstrad make such a unit, but there are several others also available from independent manufacturers.

You can use an RS232C interface to connect your CPC 6128 to certain types

of printer, to other computers (for transferring files) and to information services such as Prestel and Telecom Gold.

The SETSIO command takes the form:

```
SETSIO,RX 9600,TX 9600,PARITY NONE,STOP 1,BITS
8,HANDSHAKE ON,XOFF OFF
```

where each parameter can take a range of values. The values given are the default values when you switch the system on. The parameters and there ranges are as follows:

RX - receive speed (baud) -50,75,110,150,200,300,600,1200,1800
TX - transmit speed (baud) -2000,2400,3600,4800,9600,19200
PARITY - ODD, EVEN or NONE.
STOP - 0, 1 or 2.
BITS - 6, 7 or 8.
HANDSHAKE - ON or OFF.
XOFF - ON or OFF.

Any or all of these parameters may need to be set up to match the equipment to which you connect the RS232C interface. Refer to the manual supplied with your interface.

SET24X80

This command sets the screen size to 24 rows of 80 columns. This size of screen has become something of a 'standard' for CP/M machines and many application programs assume it. The normal Amstrad screen is 25 x 80, with the 25th line reserved for status information and error messages.

To switch on the 24 x 80 display, use the command:

```
SET24X80 ON
```

and replace ON with OFF to turn it off again.

SHOW

This is another command with a number of options. You use it to display a variety of information about the disc. The following examples should illustrate its many uses:

`SHOW A:` shows the read/write status of the disc in the named drive, and the amount of space on it.

`SHOW A: [LABEL]` shows the directory label (if you've assigned one), any password in force, and the current

time stamps on the directory.

`SHOW A: [USERS]` displays a breakdown of the current users of the disc, the number of files assigned to each of them, and the number of free directory entries on the disc. If you're the only person using your CPC 6128, you'll probably only need to use User 0. This function will not, therefore, be of much use to you.

`SHOW A: [DIR]` shows the number of free directory entries remaining on the named disc.

If you leave the drive letter off any of the preceding commands, the SHOW utility will act on the currently selected drive.

These are all the simple utility programs offered by CP/M, but there is a host of other 'goodies' also supplied on the Amstrad master disc/s, and it is these that are described in Chapter 3.

Chapter 3

CP/M PROGRAMMING AIDS

As well as the utility programs described in the last chapter, a number of other programs are supplied on Amstrad's CP/M system disc. These can be very useful and represent a considerable amount of free software. In this chapter we will be looking at each of them and pointing out their uses and limitations. The programs supplied with CP/M 2.2 are:

ASM	A Z80 machine-code assembler
DDT	A debugger (conveniently the initials for Dynamic Debugging Tool!)
DUMP	A routine which will produce a hexadecimal dump of contents of a section of memory
ED	A text editor
PIP	The Peripheral Interchange Program, which is used mainly for moving files between different peripherals
SUBMIT	A program which allows a sequence of CP/M commands to be saved as a file and executed in one go. It's used similarly to the initial command buffer in the SETUP utility (described in chapter 2)
XSUB	A program which allows a sequence of commands within an application program or a language to be stored in a separate file and executed from CP/M.

These programs will be described in alphabetical order for ease of reference, but you will probably use PIP most frequently, followed by ED, ASM, DDT, DUMP, SUBMIT and XSUB.

ASM

This is an 8080 assembler. It allows you to take source programs, written in 8080 assembler mnemonics, and convert them into machine-code. These machine-code object programs may then be directly executed from CP/M as .COM files. The 8080 microprocessor was a predecessor of the Z80. ASM

supports all of the 8080 instruction set, which is included in the Z80 instruction set. There are, however, some Z80 instructions which are not supported by ASM.

ASM is a fairly basic assembler and doesn't support the relocation of code from one part of memory to another. In some applications this won't be important, but for many programs it can present quite a problem.

The assembler also lacks the facility to create *macros*. A macro is a sequence of assembly language commands which may be inserted into a program by calling the sequence by name. When the source code is assembled, the name is replaced by the full set of commands within the macro. A library of macros is particularly useful for every day functions such as accessing disc files.

ASM expects a source file to have a filetype .ASM and each line of the code to take the following form:

Line Number label:opcode operands;comment

So a typical line of assembly code might look like this:

```
20 LOOP:LD A,"Z";loads accumulator with 90 (ASCII
'Z')
```

You would normally write the source code of an assembly program with a text editor, such as ED or Hisoft's ED80 (see chapter 4), and save it to disc with the **.ASM** filetype. To call ASM, you would then issue the command:

```
ASM FILENAME
```

This form of the command is, in fact, short for **ASM FILENAME.AAA**, where each of the three A s covers a different option of the assembler.

When ASM successfully assembles a source program, it produces two files. One is the machine-code object file, which is automatically given the filetype **.HEX**. The other is a copy of the source program, complete with line numbers and comments, so that you can easily obtain a print-out of your program. For this reason, it is given a filetype **.PRN**.

The three options at the end of the ASM command are all drive letters. The first letter is for the drive where the source code is stored, the second is the drive where you want the .HEX file stored and the third is the drive where drive you want the .PRN file stored. All three options default to drive A, but if you have twin drives you might want to specify drive B for some of them. You can suppress either or both of the .HEX and .PRN files by specifying Z as the appropriate drive option.

When you call ASM, it will assemble the source program and display an assembly listing as it does so. This listing is similar to the contents of the

.PRN file. If the assembly is successful, the assembler signs off with the message:

```
xxxH USE FACTOR
END OF ASSEMBLY
```

where xxx represents the proportion of the symbol table the program has used.

If things don't go so well, ASM will add error codes to the displayed listing and the .PRN file, or stop short with an error message. The error codes and their meanings are as follows:

D **Data error.**
E **Expression error.** An assembler expression has been misspelt, e.g. SDD A instead of ADD A.
L **Label error.** A label has been incorrectly used, or duplicated (each label must be unique).
N **Not implemented.** You have tried to use a feature of ASM which it doesn't support e.g. defining a macro.
O **Overflow.** The assembler has tried to evaluate an expression which is too complicated.
P **Phase error.** A label has a different value on each pass of the assembler (ASM passes through each program twice).
R **Register error.** A value assigned to a register in an assembler statement is incorrect.
V **Value error.** An operand in an assembler statement is incorrect, e.g. LD A,355 instead of LD A,255.

The error messages of errors which will stop the assembly short are:

NO SOURCE FILE PRESENT	There is no file on the specified disc with the filename 'filename.ASM'.
NO DIRECTORY SPACE	The disc's directory is full.
SOURCE FILE NAME ERROR	The source filename has been misformed and ASM cannot read it.
SOURCE FILE READ ERROR	The source file has error(s) in it and ASM cannot read it properly.
OUTPUT FILE WRITE ERROR	Either or both of the two output files (.HEX and .PRN) cannot be written to the disc. It may already be full.
CANNOT CLOSE FILE	An output file cannot be closed. Perhaps the disc is write-protected.

It's outside the scope of this book to delve into assembly language programming, and you should refer to one of the books listed in the Bibliography if you want to learn more about the subject. If you find ASM lacks the sophistication you need for your assembly language work, then you

should consider buying an alternative development package, such as Hisoft's Devpac 80. This product is described in Chapter 4.

DDT

Digital Research's *Dynamic Debugging Tool* is a general-purpose machine-code monitor and debugger. Its main use is to look at object code generated by ASM, or any other assembler, to run the code and to alter it directly. DDT also contains a disassembler (which will regenerate assembler mnemonics from object code), as well as a number of other useful utilities.

To call DDT, you insert a CP/M system disc and type **DDT** either on its own, or with a filename (which should have either a **.HEX** or **.COM** filetype extension). If you don't include a filename, the display will show:

```
DDT VERS 2.2
```

DDT uses **-** as a prompt, to distinguish it from other CP/M utility programs.

If you include a filename in the DDT command, the file will be loaded automatically after entering DDT. If, for example, you enter:

```
DDT ED.COM
```

to examine CP/M's text editor (also on the Amstrad system disc), the display will show:

```
DDT VERS 2.2
NEXT PC
1B00 0100
-
```

The extra information here is the first free byte of memory after the program under examination (**NEXT**), and the current position of the program counter (**PC**), both in hexadecimal.

There are thirteen commands available within DDT, some of which may take numbers or addresses as parameters. The commands are as follows, and the examples used assume you have indeed loaded ED.COM for examination.

(Note that a pair of angled brackets, < >, enclosing an address or number indicates that the parameter is compulsory, whereas a pair of curved brackets, ( ) , shows an optional parameter. Unless otherwise stated, there should be no space or comma between the command letter and any following parameters.)

A<start address> will assemble standard 8080 mnemonics one instruction

at a time and overwrite any existing code at the current address. If you change an object program using this option, and save it back to disc, you will still need to alter the source code separately. The start address specified should be four hex digits (e.g. 0020 and not just 20). Pressing <ENTER> on its own will leave this option.

D(start address) - will DUMP 256 bytes to the screen, starting at the current address. If the start address is specified, the dump will start from there rather than the current address. Each byte is shown as a hex number and as its corresponding ASCII character (if that character is displayable).

If you type **D** directly after entering DDT with the ED.COM file loaded, the display will be as shown in Figure 3.1

```
0100 C3 C0 01 20 43 4F 50 59 52 49 47 48 54 20 28 43 ... COPYRIGHT (C
0110 29 20 31 39 37 39 2C 20 44 49 47 49 54 41 4C 20 ) 1979, DIGITAL
0120 52 45 53 45 41 52 43 48 20 44 49 53 4B 20 4F 52 RESEARCH DISK OR
0130 20 44 49 52 45 43 54 4F 52 59 20 46 55 4C 4C 24  DIRECTORY FULL$
0140 46 49 4C 45 20 45 58 49 53 54 53 2C 20 45 52 41 FILE EXISTS, ERA
0150 53 45 20 49 54 24 4E 45 57 20 46 49 4C 45 24 2A SE IT$NEW FILE$*
0160 2A 20 46 49 4C 45 20 49 53 20 52 45 41 44 2F 4F * FILE IS READ/O
0170 4E 4C 59 20 2A 2A 24 22 53 59 53 54 45 4D 22 20 NLY **$"SYSTEM"
0180 46 49 4C 45 20 4E 4F 54 20 41 43 43 45 53 53 49 FILE NOT ACCESSI
0190 42 4C 45 24 42 41 4B 24 24 24 42 41 4B 24 24 24 BLE$BAK$$$BAK$$$
01A0 2D 28 59 2F 4E 29 3F 24 4E 4F 20 4D 45 4D 4F 52 -(Y/N)?$NO MEMOR
01B0 59 24 42 52 45 41 4B 20 22 24 22 20 41 54 20 24 Y$BREAK "$" AT $
```

Figure 3.1 Using DDT to dump part of the ED.COM file

You can see that this first section of the ED.COM program contains some of the error messages that are used by the program. In fact, there's a machine code jump instruction in locations 0100, 0101 and 0102 which causes ED to start running from location 01C0. You might like to have a look at that section of memory.

F<start address>,<end address>,<number> will FILL the specified area of memory with the hex number included as the third parameter. The area of memory filled will include both the start and end addresses given. If you give the command:

```
F1B00,1B10,2A
```

and then dump the 256 bytes from location 1B00 onwards, using **D1B00**, you will find that the first seventeen bytes have been set to **2A**, which is the ASCII code for an asterisk. The rest of the dump will probably be full of **00** bytes.

G(start address),(breakpoint 1),(breakpoint 2) will GOTO and start execution of a machine-code program from the start address specified. If no

start address is given, execution starts from the current address. Once execution has started the object program takes control. DDT will only take control again if you press <DELETE>, or a breakpoint is encountered in the object program.

Breakpoints may either have been included in the original source program before it was assembled, or may be added from within DDT by specifying up to two of them as parameters to the G command. A breakpoint consists simply of the hex address at which you want the program to stop.

The command **G0** has a special use, as it restarts CP/M, and may therefore be conveniently used to leave DDT.

H<number 1>,<number 2> will take the two HEX numbers given as parameters and display their sum and difference, again in hex. For example, if you type **H004A,002F** the display will show:

```
0079  001B
```

The H command uses *two's complement* arithmetic, so negative results may be displayed. Try reversing the order of the two numbers in the example above.

I<filename> will INPUT a filename to the *file control block*. This command must be used before using the R command to read an object file into the computer. As an alternative to the DDT ED.COM command described at the beginning of this section, you could have used the three commands:

```
DDT
IED.COM
R
```

to achieve the same effect. Obviously it is quicker to use the I and R commands to read a new file from within DDT, than to leave DDT and restart the program with a new filename.

L(start address),(end address) will LIST a disassembled sectioned of object code, so you can see the source statements from which it's assembled. This command is very similar in use to the D command. If you type **L0100** with the ED.COM program loaded, the display will look like this:

```
0100 JMP 01C0
0103 ??= 20
0104 MOV B,E
0105 MOV C,A
0106 MOV D,B
0107 MOV E,C
0108 MOV D,D
0109 MOV C,C
010A MOV B,A
010B MOV C,B
010C MOV D,H
```

You can see the JMP instruction that was mentioned in the description of the D command. The other instructions are actually meaningless. The disassembler has tried to make the ASCII characters used in the ED error messages into assembler statements. When reading a disassembled listing, you have to decide whether each hex byte should be interpreted as an assembler instruction, an ASCII character, or a piece of numeric data.

Unless you supply an end address as well as a start address.the 'L' command will list twelve lines of the program. As in the above example, though, this doesn't necessarily mean the contents of 12 bytes. Each line of the listing will be a valid assembler statement, and these may be one, two or three bytes long.

M<old start address>,<old end address>,<new start address> - will MOVE the section of machine code delimited by the old start and end addresses (and including the bytes at both of them) to start at the new start address.

The move is actually a copy, as the section of program is not erased from its old position, but duplicated from the new start address onwards. If you type `M0120,0127,1B00` and then `D1B00` you will see that the word `RESEARCH` has been copied from its original position to start at 0B00.

R(offset) will READ a file whose filename has already been put in the file control block (using the I command) into memory. If a hex number has been included as an offset parameter, then the file will be loaded into memory starting at address 0100+offset.

S<start address> will SUBSTITUTE a hex byte entered from the keyboard for one in memory at 'start address'. The original byte and its address are displayed and you may enter any valid hex value (00 to FF), which will be written to memory when you press <ENTER>. If you press <ENTER> alone, the byte will be left unchanged.

After you have changed the byte at 'start address', the next address and its contents will be displayed and you can continue to substitute bytes until you press '.', followed by <ENTER>, which will end the substitution and return you to the '-' prompt.

As an example, you could change the ED error message 'DISK OR DIRECTORY FULL' to 'DISC OR CATALOGUE FULL' by using the S command as follows:

```
Existing bytes    Keyboard entry
  S0131
  013144              43
  013249              41
  013352              54
  013445              41
  013543              4C
  013654              4F
  01374F              47
  013852              55
  013959              45
  013A20              .
```

If you now type **D0100** you will see that the error message has been changed. The bytes entered are, of course, the ASCII values for the letters C A T A L O G U E, in hex.

T(number) - will TRACE a program by executing one instruction and displaying the contents of the Z80 registers afterwards. If you add a number parameter, that number of instructions will be executed, with the registers displayed after each.

U will UNTRACE a program and execute all the instructions in it, stopping and displaying the register contents only after the last.

X(register name) will EXAMINE the flags and registers within the Z80 microprocessor, and display their values. If used on its own the X command will display all the registers in the following form:

```
C0Z0M0E0I0  A=00  B=0000  D=0000  H=0000  S=0100
P=0100 JMP 01C0
```

The first part of the display, **C0Z0M0E0I0,** shows the current state of the Z80 status flags. The five flags are **C** for Carry, Z for Zero, M for Minus, E for Even parity, and I for Inter-digit carry. If you're not familiar with these flags, you should consult a good textbook on assembly language programming (see Bibliography).

The remainder of the display shows the current contents of the Z80 registers, and a disassembly of the instruction pointed to by the program counter (the P register). The abbreviations for the registers are:

A	Accumulator
B	BC register pair
D	DE register pair,
H	HL register pair,
S	Stack pointer
P	Program counter.

The X command can also be used to examine any of the Z80 registers or flags individually, and to alter their contents. If you follow the X command with a valid register letter (A,B,D,H,S,P), or flag letter (C,M,E,Z,I) the contents of the register or flag will be displayed, and you can change this by typing the new value and pressing <ENTER>. If you press <ENTER> on its own, the register or flag will be left unaltered. Note that values entered into any register but the Accumulator should be four hex digits. The Accumulator should be two hex digits.

DUMP

This is a hex dump routine, similar in output to the D command within DDT, but without the display of ASCII characters. If you type <CTRL>P first, the output from DUMP will be copied to a printer. You can either specify a filename after DUMP, such as:

```
DUMP ED.COM
```

which will dump 'ED.COM' to the screen or printer, or use the wild card *, in a command such as:

```
DUMP *.BAS
```

which will dump the first file found on the disc with the filetype .BAS. You can specify a file on a drive other than A by preceding the filename with the drive letter, e.g. DUMP B: FILENAME.TYP would dump FILENAME.TYP from drive B.

ED

This is the text editor supplied with CP/M. It should not be mistaken for a word processor, such as Amsword or WordStar, but is quite adequate for entering the text of a program which will later be compiled, or for storing notes or simple data in a CP/M file.

To call ED, make sure you have a system disc in the drive with the file ED.COM in its directory, and type:

```
ED FILENAME.TYP
```

ED will be loaded and FILENAME.TYP will be loaded into the File Control Block (FCB). The FCB is the place where CP/M stores the names of files it's dealing with. The prompt used by ED is a *, and this will be displayed on the screen when you call the program.

You will probably notice immediately that the file is not loaded or displayed

on the screen. Most text editors *assume* you want to load and display the file you've named, but ED has to be told exactly what to do.

The second thing you'll need to know before being able to understand how ED does things, is that it uses a Character Pointer (CP) within the text file in much the same way as CP/M uses a cursor to show your typing position within the screen display. Confusingly, ED's CP is not tied to the screen cursor, so that the cursor may be at one end of the file, while the CP is at the other! Once you get used to this, however, it becomes second nature to position the CP in the right place before manipulating text in the text file.

For instance, to display EX1.BAS (from the Amstrad CP/M disc) under ED, you could enter ED with the command `ED EX1.BAS`. You would then have to load the file into memory, move the CP to the beginning of the file and display the first 24 lines of the file (all that will fit on the Amstrad screen). To do this, you could enter:

```
B#A23T
```

Your worst fears about CP/M look like coming true! But let's run through what's actually been done, and shed a little light on ED's set of commands.

The first thing to note is that the above characters represent more than one command. In fact, the sequence `B#A23T` is three commands: `B`, `#A` and `23T`. Most of ED's commands can be combined like this. It keeps typing down to a minimum, but does little to aid understanding.

The B command moves the CP to the Beginning of the text file. Even if the text buffer's empty, as in this example, it's a good idea to make sure you know where the CP is before you start.

The A command Adds the file whose name is in the FCB to the text buffer. In effect, this is the command which loads a file into memory. The # means 'as much as possible'; in this case, load all the file into the text buffer. You'll use # a lot if you make use of ED — it's a useful symbol.

The T command Types text in memory onto the screen. As you might have guessed, the 23 refers to the number of lines to be displayed. Most of the modifying parameters in ED are typed *before* the command letter itself.

If you've been working through this example at the keyboard of your CPC micro, you'll have noticed that ED has put line numbers before each line of the EX1.BAS text file. ED provides its own line numbers, which always start at 1 and increase by one for each line. These numbers are for *your use within ED*, but are *not* part of the text file, and will not be saved when you save the file to disc. They do help in finding out where the CP is, though, since the current line number is displayed before ED's * prompt.

ED Commands

Here are all the commands for ED, listed in alphabetical order. The memory aids, in italics, are my own and not necessarily those originally intended by Digital Research. They should help you memorise the commands, though. The commands all use a similar syntax to the three just described, so don't be afraid to play around them and find out how they work. They'll all seem a bit strange to start with, particularly if you've been used to using a word processor, but it's worth persevering — unless you can afford to buy an alternative CP/M editor (see Chapter 4 for a description of one of these).

(number)A *adds* (loads into memory) lines from a disc file. If you precede A with a number, that number of lines will be loaded.

(–)B moves the CP to the *beginning* of the text file. If you precede the command with a '–', the CP will move to the end of the text file, instead.

(–)(number)C moves the CP forward (backward if '–' is used) the number of characters specified by 'number'.

(–)(number)D *deletes* the number of characters specified by 'number', forward from the CP (backward if '–' is used).

E - *ends* the current ED session, closes the text file and returns you to CP/M. This command must not be combined with any other.

(start line No::end line No)(number)F<string><CTRL>Z finds the first occurrence in the text file of the string of characters defined by 'string'. If you precede the F command with a number (call it n), ED will find the n th occurrence of the string. ED moves the CP to the appropriate line and displays the line number.

If you don't want to search through the whole file, you can state two line numbers (ED's line numbers, that is) separated by '::', and ED will only search between those two lines, including both. Typing a lower case 'f' will make ED search for an exact match of 'string'. Using a capital 'F' will force 'string' to upper case before the search is started.

You should get into the habit of finishing the search string with <CTRL>Z. Although you'll be alright when you use the command on its own, if you use more than one command in combination e.g. f<string>0T, ED will be confused and search for string0T, because it doesn't know where the search string ends.

H *halts* the current edit, saves the modified text file, and moves the CP to the beginning of the file. This command must not be combined with any other.

(line number:)I *inserts* characters from the current position of the CP (not the display cursor, remember). Pressing <CTRL>Z will end the insertion. If you precede the I command with a line number and ':', the CP is moved to that line before starting the insertion.

You don't have to start inserting text from the start of a line, of course. You can use the C command to move the CP forward or backward in the line, although you will have to count the number of characters by eye! There is no 'move to the end of the current line' command, although you could make use of the F command.

(number)J<search string><CTRL>Z<insert string><CTRL>Z('delete to' string) joins one string to the end of another. When it receives this command, ED will search from the current position of the CP until it finds 'search string', and will then insert 'insert string', directly after it. If a 'delete to' string is also included in the command, all the text between the end of the 'insert string' and the beginning of the 'delete to' string will be deleted.

You should use <CTRL>Z (which appears on the screen as ^ Z) to separate each string in the J command. If you precede the command with a number, the Joining process will be repeated that number of times.

(start line No::end line No)(number)K *kills* (deletes) the current line. If preceded by a number, ED deletes that number of lines, starting from the position of the CP. If start and end line numbers are given, all lines between and including those numbers will be deleted. It's worth remembering the use of #. For example, the command 20:#K will delete all lines from line 20 to the end of the text file.

(–)(number)L moves the CP forwards (backwards if '–' is used) the number of *lines* specified by 'number'.

(start line No::end line No)(number)M<commands> will repeat the set of commands following M a *multiple* number of times. If nothing precedes M, the commands will be repeated until the end of the text file is reached, or an error is created.

If a number is given, the commands will be repeated that many times, and if line numbers are given, the commands will be repeated as many times as possible between those lines. The M command is very useful for speeding up repetitive editing tasks, such as renaming variables in a program (see the section on the S command for an example of this).

(number)N<search string><CTRL>Z searches the text file in memory, and also the source file on disc. This is particularly useful if you haven't loaded the complete file into ED. The N command has the same rules as the F command.

O takes you to your *original* position before you started to edit the current text file. You will have to confirm the command by answering ED's `O-(Y/N)` prompt.

The O command clears memory and you will loose any modified text file that is in it. The CP will be positioned at the beginning of memory. This command must not be combined with any other.

(-)(number)P displays a *page* (or 'number' of pages) of the text file. Rather than listing a set number of lines of text, you can use the P command to display a page of 23 lines. The command will normally display a page ahead of the position of the CP, but if '–' is used, the page before the CP will be displayed instead. Thus, the command B23T can be replaced with B–P.

Q *quits* the current editing session, leaving your source file unaltered on disc, and returns you to CP/M's `A>` prompt. This command must not be combined with any other.

(line No:)R(filename) *reads* text from a temporary file held on disc (or from a library file, which must have a .LIB filetype extension) into the text file, at the current position of the CP. If you precede the R command with a line number, the text will be inserted into the text file starting at that line, rather than at the position of the CP.

ED maintains a temporary file so that you can move or copy blocks of text from one place in the text file to another. You first copy the section of text into the temporary file, using the X command, then read it back in at the new position in the text file. You can then delete the original lines from the text if you wish.

The .LIB filetype stands for LIBRARY, and may be effectively used to mark useful subroutines or procedures stored on disc. You can then use these by reading them into an ED text file loaded with a program under development.

(number)S<search string><CTRL>Z<substitute string><CTRL>Z searches for the given search string and, when found, substitutes it with the 'substitute' string. It does this once, or a given 'number' of times. Again, the search and substitute strings should be separated and followed by <CTRL>Z characters.

As an example of its use, you could replace each occurrence of the variable 'x' in the EX1.BAS program, with 'xpos', by using the combined command:

```
BMSx^Zxpos^Z
```

If you follow through the different commands, you can see that this command moves the CP to the beginning of the text file, and then searches through the file, repeatedly replacing 'x' with 'xpos'. Remember, that to ED, the strings 'x' and 'xpos' are *just* strings. The editor doesn't differentiate variables from any text in a program.

(–)(number)(start line No::end line No)T - *types* lines of the text file onto the display. A T command on its own displays the current line, and –T displays the preceding line. Adding a number to the command displays that many lines. You can also display a range of lines, by specifying the start and end lines of the range.

(–)U makes ED force lower case entries to UPPER case. To disable this feature, type –U.

(–)V enables ED's line numbers. There doesn't seem to be any connection between this command letter and its function! Typing –V disables the line numbers. 0V displays the amount of memory available for the text file and the total size of memory, both as decimal numbers, in the form:

```
23923/25Ø15
```

Subtracting the first from the second gives the size of the text file.

(number)W *writes* the specified 'number' of lines of the text file to disc, starting from the start of the file.

(number)X *extracts* 'number' of lines of the text file and stores them in a temporary disc file X$$$$$$$.LIB. If there is no file of this name on your disc, ED will create one automatically as part of the X command. The command 0X will delete the X$$$$$$$.LIB temporary file.

(number)Z makes ED pause (sleep?) for 'number'/2 seconds between each command. This gives you the chance to see what you're doing when executing combined commands!

(–)number moves the CP forward (backward if '–' is used) that number of lines and displays the new line.

<ENTER> moves the CP one line forward and displays that line.

– moves the CP one line backward and displays that line.

As well as the single character commands just described, ED makes use of a number of <CTRL> sequences, some of which you have already met. These are as follows:

<CTRL>C aborts the current editing session, loosing the text file in memory, and restarts CP/M.
<CTRL>E moves the display cursor to the start of the next line. Useful when typing in a long command.
<CTRL>H deletes the last character typed. Use when inserting text.
<CTRL>I moves cursor to the next tabulation.
<CTRL>J performs the same function as pressing <ENTER>.
<CTRL>L inserts a <Line Feed><Carriage Return> code into a command line. Using this code within an 'F' or S' command enables you to search for the end of a line.
<CTRL>M performs the same function as pressing <ENTER>.
<CRTL>X deletes the current command line.
<CTRL>Z defines the end of strings in ED commands.

When you use ED, particularly before you're fully familiar with the command keys, you will occasionally see an ED error message. These take the form:

```
BREAK "'indicator'" AT 'command letter'
```

where 'indicator' can be '?', '>', '#' or 'O' and command letter is any of the ED commands or control sequences. The error indicators have the following meanings:

? – ED can't recognise the command.
> – the text file is full. You'll need to save at least part of the file on disc.
– ED can't obey a repeated command the number of times specified.
O – ED can't open the '.LIB' file specified in an 'R' command. Perhaps you've put the wrong disc in the drive, or have a filetype other than '.LIB'.

ED can seem rather daunting when you first come to it, and it does operate in a rather different way from more modern editors. When you get used to its economy of command and operational elegance, though, you may well become a devotee. It has remained virtually unchanged in all the versions of CP/M and, in computer terms, is something of a 'classic'.

PIP

PIP is another memorable acronym, this one standing for Peripheral Interchange Program. As the name suggests, PIP's main function is to move files around between peripherals. The peripherals you will usually be concerned with will be disc drives, but PIP may also be used to move files to any other peripheral. You can use PIP, for instance, to print out a file or to send a file to a serial interface (perhaps connected to a modem).

There are two ways of using PIP. You can issue a command which will load the program, transfer the file from one place to another, and then reload CP/M. Alternatively, you can load PIP into memory and use it repeatedly from there.

It must be said that PIP is much more use to you for disc transfers if you have more than one disc drive. Although it's quite possible to transfer a file to and from the same drive (in effect duplicating it), it's not possible to stop the copying once its started and swap discs in the same drive. This means that you can't use PIP for backing up a disc in a single drive system. Amstrad provide their FILECOPY and DISCCOPY routines to overcome this problem. These programs are described in Chapter 2.

To copy a file from one drive to another, you could use the PIP command:

PIP <destination drive letter:>=<source drive letter:><filename>

As an example, if you have twin drives and want to copy ED.COM from drive A: to drive B:, the actual command would be:

```
PIP B:=A:ED.COM
```

Note that the destination drive is put first. The command is saying 'let drive B take a copy of the file ED.COM from drive A'. The syntax is the same as BASIC's LET command.

This is about the simplest form of disc-to-disc file transfer that you can make PIP do. You can add a whole host of options to the command, though, and use wild cards to specify more than one file in a single PIP command.

You can save the file copy under a different name by using that name after the destination drive letter. Using the same example, but saving the copy under the name EDITOR.COM, you would use:

```
PIP B:EDITOR.COM=A:ED.COM
```

To save a copy of the file back on the same disc, you have to use different filenames, but you can omit the source and destination drive letters. PIP will assume the current drive for both, e.g.:

```
PIP EDITOR.COM=ED.COM
```

will save EDITOR.COM back on the same disc that ED.COM came from.

To save all the files from the disc in drive A onto drive B you could use the command:

```
PIP B:=A:*.*[V]
```

The ? and * wild cards can be used as before to replace a single character, or any number of characters, respectively. The V in square brackets is one of the PIP options just mentioned, and stands for Verify. The V option makes PIP check each file copied to the destination disc by reading it straight back, from the disc and comparing it with a copy of the source file in memory. The above PIP command is ideal for making safe backup copies of discs between twin drives. PIP options are listed at the end of this section.

If you state more than one source file in a PIP command, and separate each with a comma, you can add several files together under a new name. For example, to combine EX1.BAS and EX2.BAS into one file called COMBINED.BAS, you could use the command:

```
PIP COMBINED.BAS=EX1.BAS,EX2.BAS
```

PIP pays no attention to line numbers in program files, and the two files in this example are not *combined*, but simply added together.

As well as copying files from one disc to another, PIP can be used for displaying or printing files. All you need do is replace the destination drive in the general command, above, with a logical device name. Thus, to display

the file EX1.BAS on the screen, you could use:

```
PIP CON:=EX1.BAS
```

and to print it out you simply replace CON (for CONsol) with LST (for LiSTing device):

```
PIP LST:=EX1.BAS
```

Remember you can only do this with a text file or program source file. If you try and display or print a machine-code file with PIP, it will have very unpredictable results. Try:

```
PIP CON:=ED.COM
```

if you want a bit of fun — but you will need to reset your micro afterwards!

PIP options

The various options you can use with PIP, which must all be enclosed in square brackets at the end of the command, are as follows:

[B] *block* read. Loads a block of the source file at a time, before saving it to the destination file. Can be useful for handling devices (such as cassette players) over which the computer has no start/stop control. Not much use on the Amstrad machines, as it is hard to configure a cassette player into their CP/M systems.

[D](number) sends only 'number' characters to each line of the destination file, deleting the rest. Useful for trimming a file with long lines to fit each onto a device with limited line length (an 80 column printer, for instance).

[F] removes *form feeds*, (ASCII character 12), from the destination file as it's saved.

[F] removes form feeds, (ASCII character 12), from the destination file as it's saved.

[H] (for technical users only) assumes Intel *hexadecimal* file format, and reports if any of the characters in the file are invalid.

[I] (for technical users only) *ignores* ':00' records when transfering Intel Hexadecimal files. Setting I automatically sets H as well.

[L] converts all upper case characters in the source file to *lower* case in the destination file.

[N] adds line *numbers* to each line of the destination file before saving it. If N2 is used, PIP adds leading zeroes to the line numbers and a Tab character afterwards.

[O] - assumes the files being transferred are *object* files, and ignores

<CTRL>Z (characters, which are end of file markers). Useful when adding object files together with PIP.

[P](number) - forces PIP to insert a form feed character (ASCII 12) after each page of the destination file. If no 60 lines is same as 'number'.

[Q]<search string><CTRL>Z transfers the source file until the search string is read (the string itself will be transferred). The string must be terminated with <CTRL>Z.

[R] *reads* system files (those with the filetype .SYS). PIP normally ignores these files.

[S]<search string><CTRL>Z transfers the source file from the point where the *search string* is read (the string itself will be transferred). Using the Q and S commands together can extract and transfer a section of the source file.

[T]<number> expands *tabulations*. When the tab character (<CTRL>I) is read in the source file, the appropriate number of spaces are substituted so that the next character in the file is positioned at a point in the current line which is divisible by 'number'.

[U] converts all lower case characters in the source file into *upper* case in the destination file.

V *verifies* the destination file. Each section of the destination file is read back into the computer and compared with the copy of the source file in memory.

W *writes* the destination file over an existing file on the destination disc with the same name, even if it is set to 'READ/ONLY'. Normally PIP will ask you for confirmation before overwriting an existing file.

X copies files which are not strings of ASCII characters. Normally, PIP checks that files are ASCII format.

If you have several file transfers to do, you may find it easier to load PIP as a separate program. To do this, you just type PIP. The program will be loaded and an * prompt will appear. You can then type standard PIP commands, but without the word PIP preceding each. When the transfer has been completed, the prompt will return. To leave PIP, press <ENTER> on its own, or type <CTRL>C.

SUBMIT

This program acts in a rather different way from the others described in this chapter. It takes a file of CP/M commands and obeys each one in turn, without any intervention from the keyboard (apart from typing the SUBMIT<filename> command in the first place). You can set up a SUBMIT file using ED, PIP or some other text input utility, and save it to disc as normal, but with the special filetype .SUB.

On its own, SUBMIT will probably not be a lot of use to the average Amstrad micro user. Its main use is to combine a series of CP/M commands to save unnecessary typing, but Amstrad already supplies it own utilities (such as SETUP) which do most things SUBMIT can do. However when combined with the last CP/M utility, XSUB, a SUBMIT file can be made a lot more useful.

XSUB

This utility can be thought of as an extension of the SUBMIT program. It allows not only CP/M commands to be put into the submitted file, but also commands used within a CP/M program. So now, as well as loading, say, PIP, you can issue commands to transfer a set of files or to dump them to the screen or printer. These are the same commands you would use if you were using PIP from the keyboard. Any CP/M utility or machine code program may be used in this way — unless it is an interpreted language, such as Microsoft BASIC (which you are unlikely to be using), or Dr Logo (which will give you a 'Bad Command' message and crash the computer).

As an example of how to use SUBMIT and XSUB together, you could set up a short SUBMIT file which will load ED,read a text file into it, move the CP to the beginning of the text file, and display the file. A handy textfile to use is the SUBMITted file itself, call it TEST.SUB.

To set up this submit file, you would do the following:

```
ED TEST.SUB
```

This loads ED and sets up a new file with the name TEST.SUB.

```
BI
```

This ensures the CP is at the start of the file, and enters the insert mode ('1:' will appear on the screen). You can then enter the following lines, finishing each with <ENTER>, and pressing <CTRL>Z to leave ED's insert mode.

```
1: XSUB
2: ED TEST.SUB
3: B
4: #A
5: -P
6: ^Z
```

If you now press 'E' against ED's '*' prompt, your 'SUBMIT' file will be saved and the 'A>' prompt will return. Now type 'SUBMIT TEST' and the TEST.SUB file will be loaded and run. The screen will look like this:

```
A>XSUB

A>ED TEST.SUB
    : *B
    : *#A
   1  : *-P
   1  : XSUB
   2  : ED TEST.SUB
   3  : B
   4  : #A
   5  : -P
   1  : []
```

As you can see, CP/M echoes each instruction it obeys from the 'SUBMIT' file, to the screen. Although, for clarity, each of the 'ED' commands in this example was put on a separate line, line 3 of the file could just as well have been 'B#A-P', saving lines 4 and 5.

6128 SPECIAL

CP/M Plus, as implemented on the CPC 6128, provides several programming facilities extra to those already described. As with the system utilities outlined in Chapter 2, some of these improvements are available as later versions of existing CP/M 2.2 programs, and some are introduced with the new operating system.

Taking the existing CP/M 2.2 utilities first, ASM, DUMP, ED, PIP and SUBMIT are still there, but SID replaces DDT and MAC and RMAC are available as alternatives to ASM. XSUB and SUBMIT are combined into one program and HEXCOM and GENCOM have been added to translate assembled machine-code into executable .COM files.

ASM

This is the same as supplied with CP/M 2.2. In fact, although it's supplied on the CP/M Plus disc, Its the only version of the program provided. See the main ASM description earlier in this Chapter.

DUMP

This program is similar to the CP/M 2.2 version, except that the display shows ASCII characters as well as the hexadecimal value of each displayed byte. The format of the display is very similar to that produced by DDT/SID.

ED

ED is basically the same program in both versions, but the CP/M Plus ED has the extra ability to deal with an input and output file separately. This means that if you call ED with a command such as:

```
ED OLDFILE NEWFILE
```

ED will automatically draw text from OLDFILE, when told to with the 'A' command, and save it to NEWFILE when given an 'E' or 'H' command. OLDFILE and NEWFILE may be preceded by a drive letter, so if you have a twin drive system you can have the two files on different drives. Note that the new file name follows the old file name, which is the opposite of the normal CP/M arrangement.

PIP

The version of PIP provided with CP/M Plus incorporates all the features of the 2.2 version, except the [B] option, which reads a file in predefined 'blocks'. PIP under CP/M Plus has two extra options, [A] and [C], which do the following:

A - Archive. This option forces PIP to copy only those files which have been changed since the last copy was made.

C - Confirm. If you use this option, PIP will prompt you to confirm each file before copying it. Useful if you're using wildcards.

Amstrad has patched PIP on the CPC 6128 so that you can save files from one disc to another using a single drive. On the CPC 464/664 this function is performed by the FILECOPY program, but PIP is quite happy to do it on the 6128. To copy a file from drive A to drive B, you use the command:

```
PIP B:=A:FILENAME.TYP
```

where the filename and filetype are for the file you want to copy. PIP will prompt you to insert each disc in turn with a message like:

```
Please put the disc for B: into the drive then press
                    any key
```

which will scroll across the bottom row of the screen. Swap discs and allow the copy to be made, and PIP will prompt for the A: disc to complete the transfer and reload the system. What is happening is that PIP is being fooled into using drive A: as both drives A: and B: for the purposes of copying, hence the term 'the disc for A:/B:' in the prompt.

You can still use PIP for copying from keyboard or serial port to the screen, a file or the printer, as with CP/M 2.2. In CP/M Plus, however, the logical devices have different names, and are a little easier to understand. You can PIP from:

filename.typ - a file.
AUX: - the RS232C port.
CON: - the keyboard.
NUL: - special device, producing 40 hexadecimal zeros.
EOF: - a special device which only produces the end of file marker, <CTRL>Z, once, at the end of the file.

and PIP to:

filename.typ - a file.
AUX: - the RS232C port.
CON: - the screen.
PRN: - a printer which will expand any TAB characters, and form feed every 60 lines.
LST: - a normal printer.

As you can see, the keyboard and screen are assigned the same logical device name, CON:, and the same name is used for both input and output. The same is true for AUX:, which can be used for the source or destination of a file transfer via the RS232C port.

You can use PIP as a very simple editor by typing:

```
FILENAME.TYP=CON:
```

and this is useful, among other things, for setting up SUBMIT files, as is shown in the next section.

SUBMIT

All the uses described for SUBMIT under CP/M 2.2 are equally applicable to the CP/M Plus version. XSUB is now integrated with SUBMIT, however, so you can send commands through to application programs without relinquishing control to them.

For example, if a file COPY.SUB contains the lines:

```
PIP
<B:=ED.COM
<AUX:=SID.COM
<
 DIR
```

and the command:

```
SUBMIT COPY.SUB
```

is typed at the keyboard, SUBMIT will load PIP, copy ED.COM onto drive B:, send SID.COM to the RS232C port, leave PIP and display the directory of the disc. The 'less than' sign (<) is used within a SUBMIT file to represent <RETURN> in an application program.

The SETUP program provided with CP/M 2.2 on the CPC 464/664 isn't available on the CPC 6128 under CP/M Plus, but many of the same facilities can be set up using the PROFILE.SUB file, which CP/M Plus automatically searches for when you first call it from AMSDOS. A useful PROFILE.SUB file can be set up like this:

```
PIP PROFILE.SUB=CON:<RETURN>
SETKEYS KEYS.CCP<RETURN><CTRL>J
LANGUAGE 3<RETURN><CTRL>J
PALETTE 0,63<RETURN><CTRL>J
```

The first line of this little exchange sets PIP up to copy whatever you type at the keyboard onto the file PROFILE.SUB. The next line sets up a series of key sequences using the Amstrad program SETKEYS. The file KEYS.CCP is suitable for working with CP/M utilities such as ED, ASM or SID. There are other files suitable for word processing and Logo. These have the filetypes .WP and .LOG, respectively.

The third line sets the English character set (switches '£' for '#'). The last line sets the display colours to bright white on black, which gives better contrast on a green screen monitor than the default colours; white on blue. You have to use <RETURN> and <CTRL>J after each command to move the cursor to the start of the next screen line.

To get this SUBMIT file to work, you have to have all the relevant program and data files on the disc. In this example, these would be:

```
SUBMIT.COM
SETKEYS.COM
KEYS.CCP
LANGUAGE.COM
PALETTE.COM
PROFILE.SUB
```

(N.B. the file you want SUBMIT to read must have a .SUB filetype)

The Amstrad micros have a relatively small disc capacity (around 170K) and it therefore pays to have as few superfluous files on your discs as possible. The six files shown in this example use a total of 12K so don't represent a heavy overhead. Remember when preparing a PROFILE.SUB file, however, that the more you put into it the less room there is on the disc for application programs.

SID

SID stands for Symbolic Instruction Debugger; in other words it's a machine-code monitor like DDT. It's more powerful than DDT, and some of the commands differ. You load SID by typing its name, with an optional filename. If you use a filename, that file will be loaded into memory as well as SID.

Here's a summary of SID's commands:

A<start address> - Enter and *assemble* code starting at the specified address.

C<start address><number1>< number2> - *call* a subroutine, starting at the specified address. The two numbers, which are optional, are loaded into the BC and DE register pairs, respectively, before the call is made.

D<W><start address><end address> - *dump* 16 lines of memory, in hex and ASCII format (compare with DDT), starting and ending at the addresses specified. If the W parameter is added, the dump is in 16 bit words, rather than bytes.

E<filename1><filename2> - *employ* a file (load it) from disc. If you use the optional second filename, this file must hold the 'symbol table' of the machine-code program.

F<start address><end address><number> - *fill* a section of memory starting and ending at the specified addresses, with the character represented by the hex number in the third parameter.

G<start address><stop address1><stop address2> - *goto* and start execution of a program from the specified address, stopping at whichever stop address is reached first.

H,<number1>,<number2> - print the sum and difference of the two hex numbers given. Note the commas separating the parameters.

I<filename> - *input* a filename and set up a file control block for a file to be read by the R command.

L<start address><end address> - *list* a disassembly of the contents of memory between the two specified addresses. If no addresses are given, disassembly starts from the current address.

M<start address><end address><new address> - *move* the block of memory between the start and end addresses specified, to start at 'new address'.

P<pass address> - record the number of *passes* the program makes through the specified address.

R<offset> - *read* the file specified by the previous I command, starting at the address given as 'offset'. If none is specified, the file will be loaded at 0100 hex.

S<W><start address> - *substitute* hex bytes entered from the keyboard for those in memory, starting from the specified address. If the 'W' parameter is used, 16 bit words are substituted instead of bytes.

T<number> - *trace* the number of instructions specified, by executing them and displaying the contents of the Z80 registers after each. T alone traces one instruction.

U<number> - *untrace* the number of instructions specified, by executing all of them, and displaying the contents of the Z80 registers after the last.

V - display four *values* as follows: NEXT - next address in memory MSZE - next address after largest file PC - current program counter address END - current end-of-memory address

W<filename><start address><end address> - *write* (save) the section of memory between the specified address to disc, using the filename supplied.

X<register or flag> - *examine* the contents of a Z80 register or flag. If used without a register or flag letter, all registers and flags will be displayed. The letters used are the same as for DDT.

There are two extra utilities supplied with SID, however, which are not available with DDT. These are HIST.UTL and TRACE.UTL.

HIST.UTL

This program produces a bar chart (histogram) showing the usage of various sections of a machine-code program. It is particularly useful if you are trying to speed up a program, as it shows where efforts to improve the coding will be best rewarded.

TRACE.UTL

This program produces a trail showing up to 256 instructions executed before a specified breakpoint is reached. The routine is most useful when trying to check the structure and logic of a program.

Details of these specialist utilities are beyond the scope of this book, but either may be loaded by attaching their names to the SID command, like this:

```
SID HIST.UTL (or SID TRACE.UTL)
```

For further details consult the bibliography at the end of the book.

MAC

MAC is an assembler like ASM, but with the big advantage that it can handle macro assembly. A macro is pre-defined section of a machine-code program which is used as a form of 'shorthand' within it. A macro is defined once in a program, and is then referred to by name when needed during programming. On assembling the program, the macro's name is replaced by the appropriate section of code.

An extension of a simple macro is the idea of a 'library' of macros, stored separately from the program and called in, when needed, during assembly. You can build up a library of macros for use with MAC, and store each on disc, using filenames with a .LIB filetype. The statement MACLIB will then load and assemble macros from the library. This facility is of limited usefulness with non-relocatable code, however, as the macros must always be loaded at their original addresses. Library modules are much more valuable when used with the RMAC assembler, described later.

To call MAC to assemble a source file, which must have a filetype .ASM, you type:

```
MAC FILENAME.ASM $options
```

The options follow the filename, and are separated from it by a '$', but MAC will assume defaults if you don't select any of them. The options mainly refer to the drives on which you want MAC's various output files to appear. They are as follows:

A - source drive for the .ASM file (A or B).
H - destination drive for the .HEX file (A, B or Z).
L - source drive for .LIB files, if used (A, B, X, P or Z).
P - destination drive for .PRN file (A, B, X, P or Z).
S - destination drive for .SYM file (A, B, X, P or Z).

The three files produced by MAC all have the same filename as the source file, but have three different filetypes:

.HEX - object code, in hexadecimal format.
.PRN - annotated printer file. May be dumped with PIP.
.SYM - sorted list of symbols (labels) used in the file. The H, P and S options refer to these files.

To specify the drive you want each of them saved on, you put the filetype letter and the required drive letter together after the $ separator. For example:

```
MAC TEST.ASM $AA HA PP SX
```

would take the TEST.ASM file from drive A: (AA), put the .HEX file back on A: (HA), send the .PRN file to the printer (PP) and send the .SYM file to the screen (SX). The special destinations X and P refer to the screen and the printer. Using Z suppresses that output file.

RMAC

This is Digital Research's Relocatable MACro assembler which does everything MAC will do, but creates relocatable machine-code. Relocatable code doesn't have to be loaded at a particular address in memory, but will work wherever it is put. Any jumps and subroutines within the program are worked out relatively, and don't refer to absolute addresses.

RMAC works in exactly the same way as MAC, except that it produces .SYM, .PRN and .REL files. A file with a .REL filetype is a relocatable object code file which will be recognised by the LINK utility, described later. The destination drive letters are also the same as for MAC, so the command:

```
RMAC TEST.ASM $PX SA RB
```

will send the .PRN file to the screen, the .SYM file to drive A: and the .REL file to drive B:.

As well as being much more versatile in use, relocatable routines can be easily built into libraries. In this way, when you develop a useful routine for one program, you can save it as a library file in its own right and use it whenever you need to do the same thing in other programs. Once you've built up a library of routines, you can link them together into programs. There are two programs supplied with CP/M Plus which are designed specifically for these purposes.

LINK

This is a utility program which combines relocatable machine-code object files (which must have the filetype .REL or .IRL) into an executable command file with a .COM filetype. The command:

```
LINK FILE1,FILE2,FILE3,FILE4
```

will link together the files FILE1 to FILE4, convert the resulting file into a .COM file and save it as FILE1.COM. Each linked file will normally use others as sub-routines.

An alternative way of calling subroutines from a file is to have several of them in a single library file. This library may be maintained using the LIB routine, described next. If you want to combine a main object program-file with modules held on a library file, you would use the LINK command like this:

```
LINK MAINFILE,LIBRARY[S]
```

The S option in brackets causes LINK to search the file LIBRARY for the routines required within MAINFILE, and to link them in, creating MAINFILE.COM and saving it to disc. There are a number of other options available for LINK, but these are beyond the scope of this book.

LIB

This is the CP/M Plus utility which handles libraries of relocatable object-code modules for use by LINK. With it you can create a library, and add, delete, replace and select modules from it. Each object module must have a filetype .REL or .IRL, where .IRL stands for an Indexed ReLocatable file and may be created from a .REL file using LIB itself. The .IRL module has the advantage of being a lot quicker to find within a library file.

You may add any of four options to the end of a LIB command, enclosing them within square brackets. They are:

I – creates an *indexed* file, with a .IRL filetype, from a .REL .REL module or program.
M – displays all the *module* names within the specified library file.
P – displays module names and any *public* variables shared with other modules.
D – *dumps* the contents of the named modules in ASCII form.

A few examples of LIB commands should serve to show how the utility is used:

```
LIB LIBRARY[M]
```

displays a list of all the module names in the library file, LIBRARY.REL.

```
LIB NEWLIB=OLDLIB1(MODULE1- MODULE4,MODULE6),
                  OLDLIB2
```

copies modules MODULE1 to MODULE4, and MODULE6 from the library file OLDLIB1.REL and combines them with all the modules in OLDLIB2.REL, creating a new library file NEWLIB.REL to hold them.

```
LIB NEWLIB[I]=OLDLIB1<MODULE2=>
```

creates an indexed library file, NEWLIB.IRL, from the modules in OLDLIB1.REL, excluding the OLDLIB1 module, MODULE2.

HEXCOM

This utility does a similar job to LINK, but for a single file. It takes an object file from ASM or MAC, with the filetype .HEX, and converts it into a command file (with a filetype .COM), ready for immediate execution. The format of the command is:

```
HEXCOM FILENAME
```

You don't need to include the filetype in the command, and the resulting .COM file will have the same filename as the .HEX file it takes as input. You can precede the filename with a drive letter, if you want.

GENCOM

This utility converts from a .HEX file to a .COM file, but allows the use of Resident System eXtensions, RSXs, within the command file it produces.

This completes the descriptions of the CP/M utilities provided with

Amstrad's disc systems. They are actually quite comprehensive, but rather less than friendly to the uninitiated. There are many people, however, who have battered through CP/M's initial resistance, and have found it to be a robust and surprisingly flexible operating system.

Some of these afficianados form the CP/M User Group, whose address is given at the end of the book. As well as being a friendly bunch who will always try and help with obscure problems, they also hold a very large library of *free* software. This consists of programs donated to the 'public domain' by interested users, both private and professional. The quality of this software is variable, but some of it is very good, and all is available for a membership fee and a small additional charge for each volume copied.

Chapter 4

Other Machine Code Utilities - an Example

In the last chapter we looked at the utilities supplied on your Amstrad CP/M system disc. Although there are programs there that cover most of the low-level applications you are likely to need, there are areas in which the supplied programs fall short of modern requirements.

In particular, the development of machine-code programs can be speeded up by the use of a more sophisticated editor for writing the source code, an assembler with extra facilities (such as the ability to use macros), and a monitor which is simple and quick to use. There are several packages of this type available for CP/M systems, but in this chapter we'll be looking at Devpac 80, from Hisoft.

DEVPAC 80

Devpac 80 is a machine-code software development package designed for any CP/M system. It is divided into the three sections just mentioned: editor, assembler and monitor. Hisoft refer to their products as ED80, GEN80 and MON80, respectively.

To show briefly how these programs work, and how they differ from the CP/M utilities described in Chapter 3, we'll develop a short machine-code program under CP/M. This is, in fact, a routine written by David Link of Hisoft, and I am very grateful to him for permission to use the program.

DEVELOPING UNERA.GEN

'UNERA.GEN', when assembled and stored on a CP/M disc as a UNERA.COM file, will restore a disc file which you have erased using the ERA command. You should use the routine immediately you discover your mistake, as the ERA command works by 'marking' the filename in the disc's directory, and making the disc space available for new files. If you have already re-used this space by storing another file, you may not be able to bring the old file back to life.

ED80

The first thing you need to do is to enter the *source* code, in Z80 mnemonics, so that you can then assemble it into machine-code. Using Devpac 80, you type `ED80` to load the text editor, and the screen then clears to show just two lines of text, at the extreme top and bottom. This is the first major difference between ED80 and ED. ED is a line-based editor, where you perform each function with a single letter command, whereas ED80 is a full-screen text editor, much more like a word processor. In fact, Hisoft have made a conscious effort to make ED80 appear this way, and have adopted identical commands to those used by *WordStar*, a very popular, commercial word processor. The two lines displayed when you load ED80 show a number of pieces of information about your text file. These are the filename, the cursor position (as row and column coordinates), the command currently being executed, whether the editor is inserting or overtyping text, the amount of free space left in memory, and 'search and replace' strings.

To enter the source code into the editor you simply type. Each character will appear on the screen as you type it, and you can use the <TAB> key to move to pre-set tabulation positions, and the <ENTER> key to move to the next line. If you make a mistake as you're typing, you may move the cursor through the text using a series of <CTRL> key sequences, and correct your error.

Your lines of text are not limited to the width of the screen, but may be up to 255 characters wide. In practice, you will probably be limited to the maximum number of characters your printer can cope with (often 80 or 132). If you type more than 80 characters in a line (without pressing <ENTER>), the text area will scroll sideways to give you more room. When you come to the end of the line you're typing, and press <ENTER>, the screen will automatically scroll back to display the first 80 columns of your text.

The source code of UNERA.GEN is shown in Figure 4.1. Don't worry if you don't understand how it works, as we'll run through the main sections of the program shortly.

```
;Unerase a File                    ;4 November 1984.

;Copyright David Link 1984

;A program to unerase a file that has been accidentally
;erased. Should be used immediately after erasing the
;file since if used later, some blocks may have been re-used.

;Format is:  UNERA filename

DEFCB EQU #5C                     ;the default FCB area.

fnamelen EQU 8                    ;filename length.
extlen   EQU 3                    ;extension length.
extent   EQU 12                   ;offset to extent in FCB.
dirlen   EQU 32                   ;directory entry length.
```

```
;CP/M BDOS call numbers.

OPEN EQU 15
CLOSE EQU 16
SEARCH EQU 17
SRCH_AGAIN EQU 18
MAKE EQU 22
SETDMA EQU 26

;Default workspace for file reads.

tbuff EQU 128


;Macro to call CP/M setting DE and C.

DOS MACRO @DE,@C
 LD DE,@DE
 LD C,@C
 CALL Dos
 ENDM

;Macro to call CP/M setting C.

SDOS MACRO @C
 LD C,@C
 CALL Dos
 ENDM


;COM files begin at #100.

 ORG #100

 LD SP,(6)                      ;set stack to top of TPA.
 LD HL,FCBSPACE                 ;intialise pointer to current FCB.
 LD (FCBPTR),HL
 DOS tbuff,SETDMA               ;set disc I/O to tbuff.
 DOS DUMFCB,SEARCH              ;and search for first entry in directory.
More_Search INC A
 JR Z,End_of_Directory          ;no more entries.

;Compare found filename with required filename.

Continue DEC A                  ;adjust because we INCed it.
 ADD A,A                        ;multiply by dirlen to get position
 ADD A,A                        ;of entry in catalogue.
 ADD A,A
 ADD A,A
 ADD A,A
 LD D,0
 LD E,A
 LD HL,tbuff                    ;point to found file.
 ADD HL,DE
Again PUSH HL                   ;and compare it with required filename.
 INC HL
 LD B,fnamelen+extlen           ;both name and type. (8, name: 3, type).
 LD DE,DEFCB+1                  ;filename starts at FCB + 1.
Match LD A,(DE)
 LD C,(HL)
 RES 7,C                        ;some CP/M's set bits on filename.
 CP C                           ;so make sure top bit is reset for comparison.
 INC HL
 INC DE
 JR NZ,NoMatch                  ;not this one.
 DJNZ Match                     ;good so far....keep going.

;Match found
```

```
 POP HL                          ;filename match but...
 PUSH HL
 LD A,(HL)                       ;..is it erased?
 CP #E5                          ;#E5 in first byte of directory means erased.
 JR NZ,NoMatch                   ;not erased...so search some more.
 LD (HL),0                       ;it was erased, so unerase it.
 LD DE,(FCBPTR)                  ;and store the FCB information in our
 LD BC,dirlen                    ;temporary table.
 LDIR
 LD (FCBPTR),DE                  ;updating table pointer afterwards.

;Search for another entry.

NoMatch POP HL
 DOS DUMFCB,SRCH_AGAIN           ;search for next entry in directory.
 JR More_Search


;We have now exhausted the directory search and built
;up our table of directory entries for the required file.

End_of_Directory

 LD DE,FCBSPACE                  ;start processing table of matched
;                                ;directory entries.

MAIN_LOOP PUSH DE                ;save table pointer.
 LD HL,extent                    ;address extent byte.
 ADD HL,DE
 LD A,(HL)                       ;make an extent of new,
 LD (DEFCB+extent),A             ;unerased file...
 DOS DEFCB,MAKE
 POP DE                          ;recover pointer to FCB....
 SDOS CLOSE                      ;...and close it.
 LD HL,dirlen
 ADD HL,DE                       ;have we reached end of table?
 EX DE,HL
 LD HL,(FCBPTR)
 OR A
 SBC HL,DE
 JR NZ,MAIN_LOOP                 ;no, so unerase next extent...
 RST 0                           ;...otherwise finished, so return.


;Subroutine to call CP/M.
;Preserves HL, DE and BC.

Dos PUSH HL
 PUSH DE
 PUSH BC
 CALL 5                          ;standard entry point to BDOS.
 POP BC
 POP DE
 POP HL
 RET

FCBPTR DEFS 2                    ;pointer to our FCB table.

;Dummy FCB used for searching the directory.

DUMFCB DEFM "???????????"
 DEFW 0,0
 DEFS 16
 DEFW 0,0

;Table of matched directory entries.

FCBSPACE EQU $
```

Figure 4.1 Source code of UNERA.GEN

This code can be entered directly into ED80 using the pre-set tabulations to space the text out. GEN80 is intelligent enough to work with a source file laid out freely on the page, as long as there is at least one space or tab character between each label, assembler mnemonic and operand on each line. The listing GEN80 produces on assembly is automatically formatted in pre-defined columns across the page.

ED80 uses <CTRL> sequences to move the cursor around the text file, and these are set up to mimic those used by WordStar. If you don't like the particular sequences chosen (and they're an acquired taste), or want to make use of your micro's cursor and function keys, you can redefine ED80's controls to suit. The Devpac manual also leaves gaps in its instructions for you to write in your preferred control sequences.

As well as being able to type in your text and edit it by inserting and deleting characters, you can move *blocks* of it around and search for specific words or phrases.

You can define a block within your text file by marking its beginning and end. To do this, you move the cursor to the point where you want the block to begin and use a <CTRL> sequence to mark it (although no physical mark shows on-screen). You then move to the end of your block and mark that in a similar way.

Once the block is defined, you can move it from its present position to another, copy it as many times as you like, or delete the whole thing. Each of these manipulations is done using other <CTRL> sequences. It's a good idea to draw up a quick reference card of the commonly used sequences, or to keep the manual by you until you've memorised them.

The search and replace function of ED80 allows you to search for a specific string of characters in the text and either display it in context on the screen or replace it with another string of your choice. These functions are again started with a <CTRL> sequence, and you then type in your 'search' and 'replace' strings on the bottom line of the ED80 display.

As an example of how you might use this feature, image you had used the name 'TB' for the temporary buffer label in the source file above. If you then decide to make the program more readable by replacing 'TB' with 'TempBuffer', you could do this all in one go using the search and replace function. Your search string would be 'TB', and the replacement would be 'TempBuffer'.

You can even use this system as a simple form of shorthand — you can save quite a bit of typing by using single letter labels and constants, and can then convert them to more readable names with the search and replace function.

Once you have typed the source program into ED80, you can save it to disc using another <CTRL> sequence. You can either save it on its own, or with

a second *backup* copy saved automatically. It's wise to keep a backup copy of all your files, in case you accidently wipe your main one. If you are using ED80 to prepare an assembly source file, you should save the file with a filetype .GEN. This is so that GEN80 can find the file easily when you use it to assemble your program.

Before going on to describe the use of GEN80, it is worthwhile briefly running through some of the important features of the UNERA.GEN source program. Several of these are only made possible by the facilities provided by more sophisticated assemblers, such as GEN80.

When you erase a file from a CP/M disc, the data is not wiped from the disc surface. Instead, the entry in the disc's directory is specially marked so that CP/M knows the file has been erased. If the file is over 16K long, then there will be more than one entry for it in the directory. Each entry will have its own marker. When new files are saved on the disc, the space occupied by the erased file is made available for new storage. The new file may then overwrite the old, erased one, if necessary.

To *unerase* a file, therefore, all that's required is to change the marker(s) which indicates to CP/M that the file has been erased. The marker(s) is normally set to 0 for an existing file, and &E5 (229) for an erased one. Before you can alter the marker(s), however, the disc directory has to be searched for the right filename, so that it's only the required file which is restored.

UNERA creates a temporary buffer, and copies into it each directory entry which matches the required filename, but with its erased marker reset. Once all the entries have been found, the directory is copied back to the disc, and the routine exits by restarting CP/M. Checks are made to ensure that the filename given doesn't refer to an existing file, and that the complete directory hasn't been read without finding a match.

GEN80

GEN 80 provides several facilities which can be used to make assembler programs easier to write and to understand at a later date. Assembler *directives* are worth particular mention, and perhaps the most useful of these is the macro.

A macro is similar in some ways to a sub-routine in a language such as BASIC, but with one important difference. A sub-routine is a section of a program which only exists in one place. When it is called from another part of the program, program control passes to the sub-routine, which is then executed before control is passed back, via a RETURN statement.

A macro, on the other hand, is simply a shorthand way of writing a repeated section of source program. Each time the macro's name is encountered during assembly, it is replaced by the instructions it contains. This means

that the program 'grows' during compilation, and will end up just as long as if the instructions in the macro had been written into the program at each point that the macro's name was used.

When writing assembler source code, it's important not to use macros where sub-routines should be used, as macros will make the program longer than necessary. With a long assembler program, however, they can save a lot of typing, and do make the source program a lot easier to read.

There are other directives supported by GEN80, which are also aids to programming. Again they are not converted to machine-code by the assembler, but, as the name implies, they give it directions during the assembly process. In practical terms, they give you the option to create and use named 'constants' within your programs, reserve space within memory for data storage and assemble sections of your program only if set conditions are met.

Constants within an assembly program are handled by the **EQU** directive. This allows you to EQUate an expression to a given label. The expression may be a simple number, or contain arithmetic and logical operators. The directive takes the form:

```
label EQU expression
```

The expression may use any of the following operators:

```
+ - * /   standard arithmetic operators
EXP   exponential operator
MOD   integer division remainder
= EQ, < LT, > GT, ULT, UGT   conditional operators
& AND, OR, XOR, NOT   logical operator
SHL, SHR   shift left and right, logical operators
```

The directive **DEFL** may be used in a similar way to EQU, and may also be used to redefine a label within the program. (EQU can only define a label once).

The **DEFB, DEFM, DEFS** and **DEFW** directives all reserve space for particular uses within a program.

DEFB<expr+expr...> reserves a byte for the result of each expression. The value of each result must lie between 0 and 255. You can define as many expressions with each 'DEFB' statement as will fit on one line of the source program.

DEFM<"string"> reserves a series of bytes and fills them with the contents of "string". The string may be up to 255 characters long.

DEFS<expression> reserves the number of bytes defined by the result of a single expression, and fills them with zeros.

DEFW<expr,expr...> reserves a two-byte word for the result of each expression. The value of the result must lie between 0 and 65535.

Using GEN80, you can direct the assembler to assemble only certain parts of your program. This might not at first appear particularly useful, but it does mean you can include special sections of code to help debugging, which you can then 'switch off' when your program is working properly. There are other uses of conditional assembly which are beyond the scope of this book.

You can trigger the assembly of sections of code with a series of conditional directives: **IF, COND, ENDC, ELSE** and **END.** The IF and COND directives are identical, but both words are provided for compatibility with other assemblers. Either directive will turn off assembly if the result of the expression following it is FALSE (0).

The **ENDC** directive marks the end of the section subject to the **IF** or **COND** condition, and is where assembly will continue from, if the preceding section is not assembled. The **ELSE** directive toggles the assembly on if it's off, and off if it's on! **END** simply switches off the assembly of all subsequent lines.

As well as assembler directives, GEN80 also supports a set of options and a number of explicit commands. The options govern things like the listing of the assembly to either screen or printer, and the production of object code. Options have to be placed at the start of the file.

Assembler commands may be placed anywhere within the program and look after the format of any listing that's produced. There is also a command to 'include' other source files. This gives you the opportunity to build up your programs in a modular way, testing each section of the program separately, and then combining them all in the final assembly. This really is *assembling* a program!

One final directive is the **ORG** statement. This defines where the object program will be positioned in memory. The directive takes the form:

```
ORG<expression>
```

and it will position the code starting at the ORiGin specified by the result of the expression following the directive.

Having prepared the file UNERA.GEN, using ED80, and having saved a copy to disc, you can invoke GEN 80 by typing its name and file you want to assemble. GEN80 assumes a filetype .GEN so you need only specify the filename of a file with this filetype. A typical session, assembling the UNERA.GEN file, would look like this.

```
A>GEN80 UNERA
GEN80 1.01 Copyright (C) HiSoft 1985
Pass 1 errors: 00

Pass 2 errors: 00

*WARNING* ORGs used :01
Symbol Table used: #016F out of #2A00

A>
```

The command **GEN80 UNERA** puts in no assembler options, but these could be added after the filename. For example, to produce a listing of the assembly on the screen, but with no object code produced, you would type:

```
GEN8Ø UNERA;LIST ON,NO OBJECT
```

If there are any errors in the source program, these will be reported, with their corresponding line numbers, as they are found. If there are errors, you will have to reload the source program into ED80 and correct them there.

The assembler passes through the source program twice so it can assign forward references within the code to their appropriate labels.

The two messages GEN80 issues at the end of the assembly deal with the positioning of the object program in memory, and the space taken by the labels used in the program. The **ORG** warning checks that you've only set one origin for the object program within your source. This is important as a CP/M command file must consist of one continuous piece of code.
The symbol table is the amount of memory assigned to labels, constants and the like, within memory. You can use the information provided to tailor the size of the table to your program's use of it.

The assembly listing produced by GEN80 consists of the source program, complete with comment lines if used, together with a list of the object codes (the machine-code) produced. The lines of the code are also numbered, from 1 upwards. Figure 4.2 shows the assembly listing produced by GEN80 for the UNERA program.

```
HiSoft GEN80 Assembler 27 Mar 85    Page:     1

Pass 1 errors: 00

005C              1 DefaultFCB      EQU       #5C
0008              2 FilenameLn      EQU       8
0003              3 FiletypeLn      EQU       3
000C              4 ExtentLn        EQU       12
0020              5 DirLength       EQU       32
0080              6 TempBuffer      EQU       128
0010              7 CLOSE           EQU       16
0011              8 SEARCH          EQU       17
0012              9 SRCHAGAIN       EQU       18
```

```
0016                10 MAKE          EQU     22
001A                11 SETDMA        EQU     26
0100                12 CALLDOS       MACRO   @P0,@P1
0100                13               LD      DE,@P0
0100                14               LD      C,@P1
0100                15               CALL    Dos
0100                16               ENDM
0100                17               ORG     #100
0100 ED7B0600       18               LD      SP,(6)
0104                19               CALLDOS DefaultFCB,SEARCH
010C 3C             20               INC     A
010D C20000         21               JP      NZ,0
0110 21C301         22               LD      HL,FCBSpace
0113 229D01         23               LD      (FCBPtr),HL
0116                24               CALLDOS TempBuffer,SETDMA
011E                25               CALLDOS DummyFCB,SEARCH
0126 3C             26 More_Srch     INC     A
0127 2840           27               JR      Z,End_of_Dir
0129 3D             28 Continue      DEC     A
012A 87             29               ADD     A,A
012B 87             30               ADD     A,A
012C 87             31               ADD     A,A
012D 87             32               ADD     A,A
012E 87             33               ADD     A,A
012F 1600           34               LD      D,0
0131 5F             35               LD      E,A
0132 218000         36               LD      HL,TempBuffer
0135 19             37               ADD     HL,DE
0136 E5             38 Again         PUSH    HL
0137 23             39               INC     HL
0138 060B           40               LD      B,FilenameLn+FiletypeLn
013A 115D00         41               LD      DE,DefaultFCB+1
013D 1A             42 Match         LD      A,(DE)
013E 4E             43               LD      C,(HL)
013F CBB9           44               RES     7,C
0141 B9             45               CP      C
0142 23             46               INC     HL
0143 13             47               INC     DE
0144 2018           48               JR      NZ,No_Match
0146 10F5           49               DJNZ    Match
0148 E1             50               POP     HL
0149 E5             51               PUSH    HL
014A 7E             52               LD      A,(HL)
014B FEE5           53               CP      #E5
014D 200F           54               JR      NZ,No_Match
014F 3600           55               LD      (HL),0
0151 ED5B9D01       56               LD      DE,(FCBPtr)
0155 012000         57               LD      BC,DirLength
0158 EDB0           58               LDIR
015A ED539D01       59               LD      (FCBPtr),DE
015E E1             60 No_Match      POP     HL
015F                61               CALLDOS DummyFCB,SRCHAGAIN
0167 18BD           62               JR      More_Srch
0169 11C301         63 End_of_Dir    LD      DE,FCBSpace
016C 2A9D01         64 Main_Loop     LD      HL,(FCBPtr)
016F B7             65               OR      A
0170 ED52           66               SBC     HL,DE
0172 CA0000         67               JP      Z,0
0175 D5             68               PUSH    DE
0176 210C00         69               LD      HL,ExtentLn
0179 19             70               ADD     HL,DE
017A 7E             71               LD      A,(HL)
017B 326800         72               LD      (DefaultFCB+ExtentLn),A
017E                73               CALLDOS DefaultFCB,MAKE
```

```
0186 D1          74                POP       DE
0187 0E10        75                LD        C,CLOSE
0189 CD9301      76                CALL      Dos
018C 212000      77                LD        HL,DirLength
018F 19          78                ADD       HL,DE
0190 EB          79                EX        DE,HL
0191 18D9        80                JR        Main_Loop
0193 E5          81 Dos            PUSH      HL
0194 D5          82                PUSH      DE
0195 C5          83                PUSH      BC
0196 CD0500      84                CALL      5
0199 C1          85                POP       BC
019A D1          86                POP       DE
019B E1          87                POP       HL
019C C9          88                RET
019D             89 FCBPtr         DEFS      2
019F 3F3F3F3F    90 DummyFCB       DEFM      "?????????????"
01AB 00000000    91                DEFW      0,0
01AF             92                DEFS      16
01BF 00000000    93                DEFW      0,0
01C3             94 FCBSpace       EQU       $
01C3             95
Pass 2 errors: 00

*WARNING* ORGs used: 01
Symbol Table used: #016F out of #2A00.
```

Figure 4.2 GEN80 assembly listing of UNERA.GEN.

If the assembly is error-free, and your machine-code program does what it's meant to do, you can now run it as you would any other CP/M command file — GEN80 produces an executable .COM file.

MON8Ø

Such is the way of machine-code programs, however, that it's unlikely to work first time. At this stage it is very useful to be able to run the program you're developing in a controlled way, so you can determine where the errors lie, and correct them. The third part of Hisoft's Devpac, MON80, does just this.

If ED80 is a more sophisticated version of ED, and GEN80 is a souped-up ASM, then MON80 is a more advanced form of DDT. You call MON80 by typing its name. The screen display is composed of three sections as shown in Figure 4.3

Down the left-hand side of the screen is a disassembly of several lines of machine-code, starting from the current address, which by default is &0100, the start address for user programs under CP/M.

Next to this, in the centre of the screen, is the register display, which shows the current state of all the Z80 registers. The numbers to the left of the HL, DE, BC and AF register pairs refer to the contents of the secondary register set, used by the operating system.

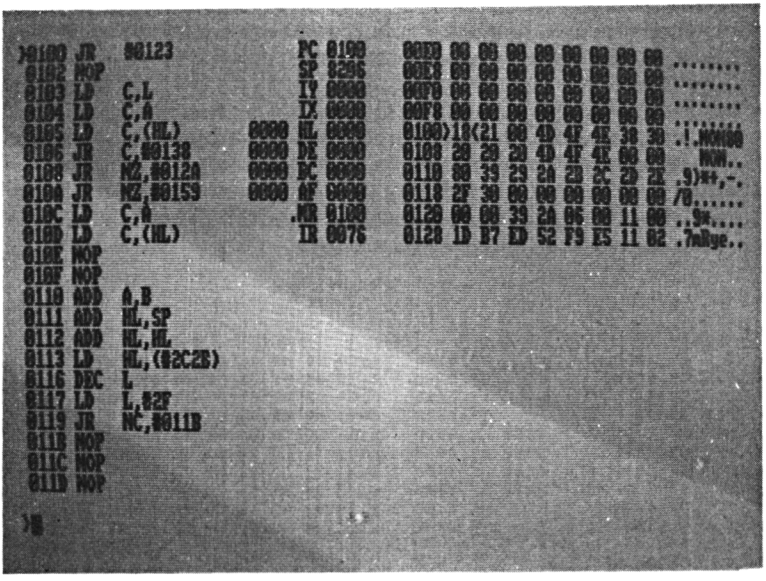


Figure 4.3 - MON80 screen display

Finally, on the right-hand side of the screen, is the memory display. This shows the hexadecimal and ASCII content of 80 bytes of memory, centred around the current program counter address.

Each display maintains its own cursor, so that you can see where you are. The disassembly display uses a '›', the register display uses a '.' and the memory display uses '>nn<' around the current byte.

If you want to monitor a machine-code program, the first thing you need to do is load it into memory. MON80's commands consist of a single character, sometimes together with a decimal or hexadecimal number as a parameter — if a parameter is required, MON80 will prompt you for it.

To load a program into memory — UNERA for instance — you use the R (read) command. On typing R you are asked for the address in memory where you want the program loaded. If you press <ENTER> here MON80 will assume an address of #0100.

Once the program is loaded, the memory display will be updated to show the start of the program, but the disassembly will still show its original display. To update the display, you press 'L' (list). The screen will then appear as in Figure 4.4

If you want to examine a different part of the code, either in hexadecimal or disassembled form, you can change the area displayed by moving the appropriate cursor. The memory display cursor can be moved up, down, left or right to scroll the display through memory.

Other sections of memory may be disassembled by first changing the value in the PC (program counter) register. This is done by moving the register cursor (the '.') until it lies alongside **PC**, by repeatedly pressing the '.' key. The register may then be changed by typing the new address, followed by another '.'. For example, you can look at the next few lines of UNERA by typing:

```
                                    012F.<ENTER>
                                    L<ENTER>

)0100 LD   SP,(#0006)          PC 0100   00E0 00 00 00 00 00 00 00 00 ........
 0104 LD   DE,#005C            SP 8206   00E8 00 00 00 00 00 00 00 00 ........
 0107 LD   C,#11               IY 0000   00F0 00 00 00 00 00 00 00 00 ........
 0109 CALL #0193               IX 0000   00F8 00 00 00 00 00 00 00 00 ........
 010C INC  A              0000 HL 0000   0100>ED<7B 06 00 11 5C 00 0E m{...\..
 010D JP   NZ,#0000       0000 DE 0000   0108 11 CD 93 01 3C C2 00 00 .M..<B..
 0110 LD   HL,#01C3       0000 BC 0000   0110 21 C3 01 22 9D 01 11 80 !C.".....
 0113 LD   (#019D),HL     0000 AF 0000   0118 00 0E 1A CD 93 01 11 9F ...M....
 0116 LD   DE,#0080           .MR 0100   0120 01 0E 11 CD 93 01 3C 28 ...M..<(
 0119 LD   C,#1A               IR 006E   0128 40 3D 87 87 87 87 87 16 @=......
 011B CALL #0193
 011E LD   DE,#019F
 0121 LD   C,#11
 0123 CALL #0193
 0126 INC  A
 0127 JR   Z,#0169
 0129 DEC  A
 012A ADD  A,A
 012B ADD  A,A
 012C ADD  A,A
 012D ADD  A,A
 012E ADD  A,A

>
```

Figure 4.4 - MON80 Screen Showing Section of **UNERA.GEN**

The original disassembly displayed code from 0100 to 012E, so by loading the next address in sequence into the PC register and relisting the disassembly, the next section of the program is disassembled. Note that the memory display is automatically updated when **012F** is loaded into the PC. Any of the registers may be altered in the same way — by moving the register cursor and typing the new address.

Programs may be run from within MON80 in one of two ways. You can either run the program normally, stopping at a specified breakpoint within it, or *single-step*, instruction by instruction. Both of these methods have their uses, and allow you to control the program flow very closely. You can even choose to execute a subroutine as one instruction, stopping again immediately afterwards. This is particularly useful if you are making calls to CP/M routines, as in the UNERA program. You know these calls will work, and can therefore save yourself some time by skipping over them while working through your program.

By checking the contents of each register at various points in the program, you should be able to see when unexpected values start to appear in them. This will then show you where the errors lie in your object program, and you can use ED80 to re-edit your source program, or change the object program directly from MON80.

One last feature of MON80 which deserves mention is the alternative memory display. You can switch between the normal memory display (which if you remember shows the hexadecimal and ASCII values of 80 bytes, centred around the current PC address) and an alternative display which shows something similar to the following:

```
  SP     IY     IX     HL     DE     BC     AF
8206   0000   0000   0000   0000   0000   0000

06C3   C3  C  C3  C  C3  C  C3  C  C3  C  C3  C
C39F   03  .  03  .  03  .  03  .  03  .  03  .
8CDC   AD  -  AD  -  AD  -  AD  -  AD  -  AD  -
9D5B   81  .  81  .  81  .  81  .  81  .  81  .
4F4D   00  .  00  .  00  .  00  .  00  .  00  .
384E   C3  C  C3  C  C3  C  C3  C  C3  C  C3  C
2030   06  .  06  .  06  .  06  .  06  .  06  .
4328   82  .  82  .  82  .  02  .  82  .  82  .
```

Each column of this alternative display shows the contents of a Z80 register pair, together with the contents of the eight bytes (or words in the case of the stack pointer, SP) addressed by the contents of the register pair. Each byte is shown in hexadecimal and with any corresponding ASCII character. In the above example, all the register pairs are set to 0000, so each column shows the same values. This display is particularly useful for looking at tables of data that have been set up in memory to be referenced via one of the register pairs.

You should be able to see from this necessarily brief description, that a properly integrated editor, assembler and monitor can greatly ease the development of machine-code programs under CP/M. Devpac is not, of course, the only such suite of utilities, but it is one of the most sophisticated packages available , and makes good use of the facilities of a CP/M micro.

Like much CP/M software, Devpac is designed to run on any Z80-based CP/M machine, and if you have access to another micro which supports CP/M, you should be able to use Devpac in exactly the same way as you use it on your Amstrad 464/664.

6128 Special

The functions of the Devpac80 suite described in this chapter are the same under CP/M Plus as under CP/M 2.2. The programs are designed to be adaptable to any computer running CP/M, in any of its versions. What will change though, are the control codes ED80 and MON80 use to operate the screen. Operations such as clearing the screen or moving the cursor must be tailored to the particular machine the programs are run on.

Although all three CPC machines are made by Amstrad, there is a difference between the CPC 464/664 and the CPC 6128. The CPC 6128 has a 'terminal emulator' built in, so that CP/M programs can be more easily 'installed'. A terminal emulator is a program which acts as a buffer between an application program, such as ED80, and the hardware of the computer. The emulator appears to the application program as a well-known computer or terminal.

In CP/M terms, Amstrad doesn't yet rank as well-known, so the CPC 6128 terminal emulator mimics a Zenith Z19/Z29 VDU. In many cases, a CP/M program will offer a series of 'standard' terminal types to choose from in its installation program. If you are installing a program with these options, simply pick the Zenith option and you should find the program will then work with the CPC 6128.

The version of ED80 supplied for the CPC 464/664 has had special machine-code patches put into the standard program to make it run properly on these machines. These patches prevent you using the installation programs provided with Devpac80 to re-install ED80 for the CPC 6128. All is not lost though, as you can return the programs to HiSoft, with an exchange fee of £5.00, and they will send you a disc with both the CPC 464/664 and CPC 6128 versions of the programs on it. All new copies of the package are supplied like this.

In the last two chapters the accent has been very much on assembly language program, partly to introduce the utilities available for use under CP/M, and partly to briefly describe some of the calls available to the machine-code programmer within CP/M.

Machine-code is by no means everybody's cup of tea, however. CP/M offers a lot to those who don't wish to endulge in the nitty-gritty of assembler mnemonics. As a sound basis on which languages and application programs may be built, CP/M is a very valuable operating system. In the next chapter, we'll explore some of the alternative languages that run under AMSDOS and CP/M. One of these, Logo, is supplied on the 'flip-side' of your Amstrad system disc.

Chapter Five

Alternative Languages

So far we've concentrated on the facilities offered by CP/M and Amstrad's own operating system, AMSDOS. Many of the functions provided are aimed at the low-level language programmer, working with Z80 assembly language.

In this chapter, however, we'll turn the coin and look at what's available for high-level programmers working with discs. There's nothing to stop you continuing to work with BASIC, of course, but there are alternatives. In this chapter, three of these will be described: Logo, Pascal and C.

Each of these languages will bring its own advantages and problems. Among the advantages are new ways of tackling programming problems, the speed of compiled languages, and the extra readability of the programs. On the other hand, each new language has another set of commands and a different syntax to learn.

Once you start to experiment with different languages, however, you will discover how rewarding it can be. Not only will you broaden your understanding of the way you can communicate with your 464/664, but you'll begin to see how a given problem can have a number of different programming solutions.

Logo

Most home micro users will be more than happy with the facilities offered on the Amstrad micros by Locomotive BASIC. It is a fast and well-equipped version of the language, and for many small to medium-sized applications will prove more than adequate. The happy BASIC user may therefore feel the existence of other languages, and the 'free-gift' of one of them, Logo, as largely irrelevant. There may also be a feeling that Logo is, anyway, a language primarily intended for teaching children computing — it's good at drawing pretty patterns, but not really very useful for any serious work.

It would be a great shame if these feelings prevented newcomers from experimenting with Logo. It is, in fact, a very versatile graphics and list-

processing language, which in certain applications, has rather more to offer, in certains applications, than BASIC.

Logo has its roots in the list-processing languages developed for work in *Artificial Intelligence*. It's immediate anticedant is LISP, a language whose name is in fact short for LISt Processor. Logo was developed at the Massachusetts Institute of Technology (MIT) in the early 1970s. It's the brainchild of Seymour Papert, a mathematician and developmental psychologist, who created it as a way of introducing children to mathematics and computing through their own experimentation.

One of the concepts (but by no means the only one) was to develop a graphics system which would allow students using Logo to move an imaginary 'turtle' around the screen. The turtle is thought of as holding a 'pen', so that it can be instructed to draw a line indicating its path, and thus build up figures and patterns on the screen.

Dr Logo (like CP/M, a product of Digital Research) is a full implementation of the language, although the 464/664 version lacks a few of the features of the full DR program.

Having said that Logo is more than just turtle graphics, these display routines and the commands controlling them are probably the easiest route into the language, and so the first part of this description will be devoted to a short 'dialogue' session which will be used to devise a picture which should be recognisable to anybody over the age of two (I successfully tested it on my two-year-old).

You load Logo by switching on your machine, inserting the CP/M system disc with the Dr Logo side (side B) uppermost, and typing ':CPM'. You will see a sign-on message and then the screen will clear and leave you with just a '?' prompt and a block cursor.

This is a full text screen and the prompt is inviting you to enter a command which Logo can obey. As with BASIC, Logo works in immediate mode, meaning that many of your commands will be obeyed immediately, without you having to run a program. Indeed Logo is not geared for program writing — rather, you construct a series of procedures, something like BASIC subroutines, to do your work for you. Before describing the creation of procedures, though, lets look at what Logo can do with its own, built-in, commands, which are known as *primitives*.

Start by typing `fd 100`. Several things will happen: the screen will clear, a small arrowhead (the *turtle*) will appear in the centre of the screen, move forward up the display leaving a line in its wake, and the ? prompt will reappear near the bottom of the screen.

The command `fd` means forward and makes the turtle move a number of arbitary units in the direction it is heading. If you issue the command without

a graphics screen on the display, Logo shrinks the size of the text screen to a small window on the bottom five lines of the display screen, so that you can see the turtle. Once it has executed your command, the prompt returns on the top line of the text window.

Type `bk 100`. You will see the turtle move back to its starting position, and the prompt will return again — `bk` obviously stands for backward. You can use any number you like for the turtle move distance — but, if the number is too small, you won't be able to see the turtle move, and if it's too large, the turtle may well move off the screen!

Now type `rt 90`. The turtle turns right through 90 degrees. The two commands `rt`, (for right), and `lt` (for left), turn the turtle clockwise and anticlockwise respectively, by the number of degrees you specified.

Before you type the following sequence of commands, try and work out what will happen on the screen:

```
fd 100 lt 90 fd 100 lt 90 fd 100 <ENTER>
```

It should be fairly easy to see that the turtle will draw the other three sides of a square, in the right half of the screen. You could, of course, have typed each command on a separate line, by pressing <ENTER> after each. Logo doesn't mind.

It's important to remember the *space* between each command and its amount. Logo uses the space character as a *separator*, so it knows where one piece of information finishes and the next begins.

Right, you know how to draw a square, but how would you draw a hexagon? Or a dodecahedron (12 sides)? You could type `fd 50 rt 60` six times for the hexagon, and `fd 50 rt 30` twelve times for the dodecahedron, but it all becomes a bit long-winded.

To get round this we can use a *repeat* loop. This is similar in operation to the *WHILE... WEND* loop in BASIC. The format of the repeat loop to draw a hexagon in Logo is:

```
repeat 6 [fd 50 rt 60]
```

As you can see, the instructions you want repeated are put inside square brackets, and the number of times you want the loop repeated is put between the repeat command and the opening bracket. Again, don't forget the spaces.

Start by clearing the screen, using the `cs` command. The text window will still show at the bottom of the display. The turtle will be moved back to its original 'home' position.

Type in the repeat command, and watch the hexagon being drawn. This is a rather slow business, because Logo has to keep redrawing the turtle after each movement and turn. To speed things up a bit you can type `ht`, short for hideturtle, and the turtle will disappear. Typing `st` (showturtle) will reverse the process.

The above repeat loop is quite simple to type, but it would be easier if you could type 'hexagon' and get the same result. You *can* do this by defining a Logo procedure which draws a hexagon and calling it 'hexagon'. Whenever you type the procedure's name Logo will draw it for you.

Throughout Logo you will find the *obvious* way you would think of solving a problem is reflected in the way the language works. Defining a procedure is very straightforward.

Procedures and parameters

To define a Logo procedure `hexagon`, you start by typing:

```
to hexagon
```

The word `to` is a Logo primitive which tells the system you want to define a procedure. The word directly after `to` is taken to be the name of the procedure. When you enter the above line at the keyboard, you'll see the prompt change from '?' to '>', as a reminder that you're defining a procedure.

The next line is the repeat loop you've just used to draw your hexagon:

```
repeat 6 [fd 50 rt 60]
```

and finally, you complete the procedure by typing `end`. The full procedure looks like this:

```
to hexagon
repeat 6 [fd 50 rt 60]
end
```

Logo now recognises the word hexagon as if it was one of its own primitives. Try clearing the screen and typing `hexagon`.

Great! The hexagon procedure works well, but to draw a dodecahedron you would have to define a new procedures. Wouldn't it be more convenient to define a general-purpose procedure which would draw any polygon (many-sided figure)? In the same way as you type `fd 50` to move 50 units, how about typing `polygon 12 50` draw a 12-sided polygon, 50 units along each side.

Again this is simple to do in Logo, but it introduces another concept, the *parameter*. The amount you tell the `fd` command to move by, and the two numbers you need to add to the `polygon` command are *handed over* to the procedure, which then uses them within its commands.

The parameters you need to use in a universal polygon program are: the number of sides the figure will have, and how long each side will be. The parameters are treated as *local* variables; they only exist within the procedure that uses them. This means that many procedures may use the same variable names, without any confusion. The only version of a given variable which exists at any one time is the one used within the procedure being executed.

The first line of the polygon procedure may be written as:

```
to polygon :sides :length
```

The format is similar to that of the hexagon procedure, except that the procedure name is now followed by two parameter names. These are the variables to which the two amounts will be passed when we use the new polygon command. Each is preceded by a colon, which is the way Logo distinguishes variables from commands. If you find it hard to remember the colon, think of how you distinguish a string variable in BASIC. In Logo, the colon is used to prefix the variable name in the same way that '$' is used as the string variable suffix.

The second line, which is the 'guts' of the procedure, is:

```
repeat :sides [fd :length rt 360/:sides]
```

The format is the same as the original repeat statement used to draw the hexagon, but rather than using numbers to specify the amount of movement and turn, the values of the two parameter variables are used instead.

The number of times the commands are repeated is now governed by the number of sides of the required figure, and the amount of movement is effectively the length of each side. The amount of turn is 360 degrees (a full rotation) divided by the number of sides in the figure.

The last line is again `end`, so the complete new procedure is:

```
to polygon :sides :length
repeat :sides [fd :length rt 360/:sides]
end
```

You should be able to see that the finished procedure is very readable. Try it out by clearing the screen and typing, say, `polygon 12 50`. This will draw a dodecahedron of side 50 units.

What happens if you increase the number of sides? If, for instance, you used polygon to draw a 36-sided figure, what would be the result. In effect, when you increaee the number of sides you approach a circle. Because the computer display is made up of a number of discrete dots, or *pixels*, you don't have to specify many sides on your polygon before it appears circular.

You now have two procedures defined in your Logo system. The hexagon procedure is unaffected by the polygon one – type `hexagon` again, if you want to check. To catalogue all the procedures you've defined, type `pots`. This memorable little abbreviation stands for 'print.out.titles'. You should see:

```
to hexagon
to polygon :sides :length
```

You can now see one of the fundamental differences between BASIC and Logo. In BASIC, each part of your program is aimed at solving the same problem. This may be broken down into more manageable parts, but the language is essentially problem-oriented. Logo, on the other hand, allows you to define procedures to do many things and hold them all in memory as extensions to its own set of primitives. These procedures can all be made to work together to solve one problem, or they can just as easily be discrete building blocks. These building blocks may be put together in a number of different ways, or left to stand as individual tools.

Turtle Graphics – Home Sweet Home

To draw a picture mentioned earlier, showing, for example, a house, a couple of trees and the sun, you could start by writing down how you would construct the picture in general terms. Your description might run along these lines:

```
move turtle
draw frame
move turtle
draw tree
move turtle
draw house
move turtle
draw tree
move turtle
draw sun
display title
```

Each of these actions can form a separate procedure, which you can then bring together to draw the finished picture. Starting from the top down, as

it were, you might type:

```
to picture
cs
move
frame
move
tree
move
house
move
tree
move
sun
move
title
end
```

This is a valid Logo procedure, although as yet the procedures `move`, `frame`, `tree`, `house`, `sun`, and `title` are undefined. It's a good idea to clear the screen, within the procedure, using `cs`. Type in the full text of the procedure, as above, and you can edit it as you develop the various procedures within it.

The move procedure

The procedure `move` has to move the turtle from one place to another, without drawing a line. A suitable procedure would be:

```
to move :xy pu setpos :xy pd end
```

The `pu` and `pd` commands stand for 'pen up' and 'pen down', and are two more Logo primitives. They stop and start the turtle drawing lines; `setpos` is another primitive, which moves the turtle to the co-ordinates given as a list after the command name. You'll need to put values in square backets after each `move` command, and these will be passed to the internal `setpos` command. Suitable co-ordinates for the first `move` command in the `picture` procedure are `[-150 -120]`.

All the figures in this picture are going to be constructed using the `polygon` procedure already defined. The trees and house will be made up of triangles and squares, the sun will be a 'circle' and the frame will be another, larger, square.

To define the procedure `frame`, you could type:

```
to frame
polygon 4 350
pu fd 12.5 lt 90 fd 12.5 rt 90 pd
polygon 4 325
end
```

This will draw two large squares, one inside the other. The middle line of the procedure moves the turtle from the outside square to the starting point of the inside one, without drawing a line between the two.

Type in this procedure too, and then edit the `picture` procedure to put in the co-ordinates for the first `move` command.

Logo has a very easy-to-use, full-screen editor which you can enter by typing `ed` followed by the name of the procedure. When you're referring to an object (in this case the procedure `picture`) you must precede its name with a '"'. Thus, to edit `picture` you type:

```
ed "picture
```

The screen will clear and the procedure will be listed down the screen, with a horizontal line and the word `edit` at the bottom. You can now use the cursor arrow keys to move the cursor around the listing, you can delete with the <DEL> and <CLR> keys, and insert text where you need to. If you decide to abandon the changes you've made to your procedure, pressing <ESC> will restore the old version. To incorporate your changes into the procedure, leave the editor by pressing <COPY>. Logo will confirm that the procedure is defined.

When you've altered `picture` to show the co-ordinates of the first `move` command, the first three lines of `picture` should look like this:

```
to picture
move [-150  -120]
frame
```

Type `picture` and the frame will be drawn. Logo will then issue an error message:

```
Not enough inputs to move in move: move
```

This means that there are no parameters for the second `move` procedure call. You haven't put any in yet — it depends where you want to draw your first tree! Try `[-100  -100]`.

When you re-enter the editor, you'll find that the cursor is already positioned at the point where Logo encountered the error. This is convenient as you can now enter the co-ordinate list `[-100 -100]`. Press <COPY> to leave the editor.

You can't put off defining the tree procedure any longer, but you might like to use the screen editor for writing it, rather than the command editor (which would be called by typing `to tree`. Type `ed "tree` instead. The

screen editor will be called with just the two lines:

```
to tree
end
```

showing.

Edit the procedure until it shows:

```
to tree :size
if :size = 0 [stop]
polygon 4 :size * 2
fd :size * 2 lt 90 fd :size * 4 rt 120
polygon 3 :size *10
fd :size *8 lt 30
tree :size - 1
end
```

You can see that `tree` too has a variable passed to it. `:size` determines how high the tree is by the number of times the routine repeats itself. It's worth running through the procedure, as it introduces a couple of new ideas, one of which is fundamental to Logo.

The first line of the procedure deals with stopping it. Each layer of the tree as it's built up is one unit smaller than the previous one, until, when the size variable reaches 0, the procedure stops.

The next four lines deal with drawing the two sections of each layer of the tree. Each layer is composed of a small square trunk section and a larger, triangular, branches section. The size of each section, and the length of movements between the two is again governed by the `:size` variable.

The final line before the end of the procedure, `tree :size - 1` calls the tree procedure from within itself, with the size variable reduced by one.

This form of working is called *recursion*, and is one of the main ways of programming used by Logo. You will also find recursion used extensively in Pascal and C. Locomotive BASIC allows subroutines to recurse, but little emphasis is placed on this.

Many computer users fight shy of recursion, because they feel the technique is uncontrollable; that calling a routine from within itself is like a vacuum cleaner sucking itself into its own bag! This fear is really unfounded, as any loop, whether iterative (the normal kind used in WHILE or repeat structures) or recursive, only needs a proper conditional statement to control how it stops. Most people who use BASIC have at some time created an *infinite loop*, and have had to press <ESC> to regain control of the machine!

Once you've created the tree procedure, leave the editor by pressing <COPY> and re-enter it to edit `picture`. Here you'll need to add a parameter to the first tree procedure — use `5`, for five layers of branches. Now try running `picture` again. This time, as well as the frame, a tree (of sorts) will be drawn, before the routine ends with the same error message as before. This time it refers to the third `move` procedure, and you'll need to put in the co-ordinates to move the turtle to the start position for drawing the house. Edit `picture` again and add the `move` co-ordinates `[-50 -100].`

The house, like the tree, is rather rudamentary, but it serves to show how a simple procedure like `polygon` can be called into service in a number of different ways.

The `house` procedure can be typed in using the screen editor, by starting with `ed "house`. Then add the following lines:

```
to house
polygon 4 100
rt 90 fd 35 lt 90
polygon 4 30
pu lt 90 fd 25 rt 90 fd 10 pd
repeat 4 [polygon 4 25 pu fd 80 rt 90 pd]
pu fd 90 lt 90 fd 10 rt 120 pd
polygon 3 100 lt 30
end
```

In many ways the structure of the `house` procedure is similar to the structure of the `tree` procedure. The calls to `polygon` draw either squares or triangles, and the other lines are commands to move the turtle from one point to the next. The repeat loop, four lines from the end of the procedure, is used to draw the four windows of the house in one operation, and shows again the power of this construction.

Leave the editor again, and try `picture`. A tree and a house will be drawn within the frame.

The next two instructions within the `picture` procedure are another `move` and a second `tree`. The `move` parameters should be `[110 -70]`, and , for a bit of variety, try a parameter of `4` for the other tree — this will make it shorter.

Now the picture's nearly there. The final `move` can be to `[-100 130]`. The sun procedure is just a neat way of encapsulating another call to the `polygon` procedure, using `20` for the sides parameter so that it comes out pretty circular. It looks like this:

```
to sun
polygon 20 8
end
```

If you call picture again at this stage, the two trees, house and sun will be drawn, but the error message this time will be:

```
I don't know how to title in picture: title
```

This is the standard Logo error message for a command it doesn't recognise. You can see that all Logo's error messages are very clear and helpful. Part of the Logo philosophy is that error reporting should be an integral part of the language, rather than an abbreviated afterthought.

The final procedure, `title`, prints a title to the picture underneath it. Logo has several ways of printing out characters or words, but `pr`, for print, will probably be the most familiar to BASIC users.

The interesting part of this procedure is the way you get Logo to print spaces in front of a word. Because the space character is use as a separator between commands and variables, Logo has to be told that, in this case, spaces should just be printed out on the screen. You can do this by using the \ character to precede each space you want printed. Thus the `title` procedure becomes:

```
to title
print [\\\\\\\\\\\\ Home Sweet Home]
end
```

The only thing that remains is to call `picture` again and see the complete picture built up.

Although this particular example worked towards a single task, producing a picture, each of the procedures used is an individual unit and could be called up for use in another task.

You can, in fact save all the procedures you have defined on disc, by typing `save` followed by " and a suitable file name. You might save the current set of procedures as `"picture`. You can also read the directory of the disc using the `dir` command. Only Logo files will be reported.

This example has introduced quite a few Logo concepts and commands, but it has dealt almost entirely with turtle graphics. As was said at the beginning of this section, Logo is a lot more than turtle graphics. In fact, the list processing abilities of Logo are as important, and in many ways more useful, that the language's graphics facilities. The second example will look at some of the list processing commands and build up a rudamentary 'expert system'.

List processing – car diagnostics

An expert system is a computer simulation of the kind of dialogue you might have with a human 'expert', whether it be a doctor, teacher or motor

mechanic. The system usually works by asking you a series of questions about your particular problem. With a doctor these questions would obviously be related to how you feel, and with a motor mechanic, related to how your car feels!

The answers to these questions will lead the program down a particular path, at the end of which will, hopefully, be some form of diagnosis. If the system is well designed, a basic assessment of the problem can be made by the computer, although human intervention would obviously be necessary to check the system's findings, particularly when dealing with health problems!

The basic structure for many expert systems is a hierarchical *tree*. If this sounds devastatingly complex, don't worry; it simply means that the result of one question will lead down to the next level, where another question will further refine the problem and lead on to increasingly more detailed queries. A simple tree structure is illustrated in Figure 5.1

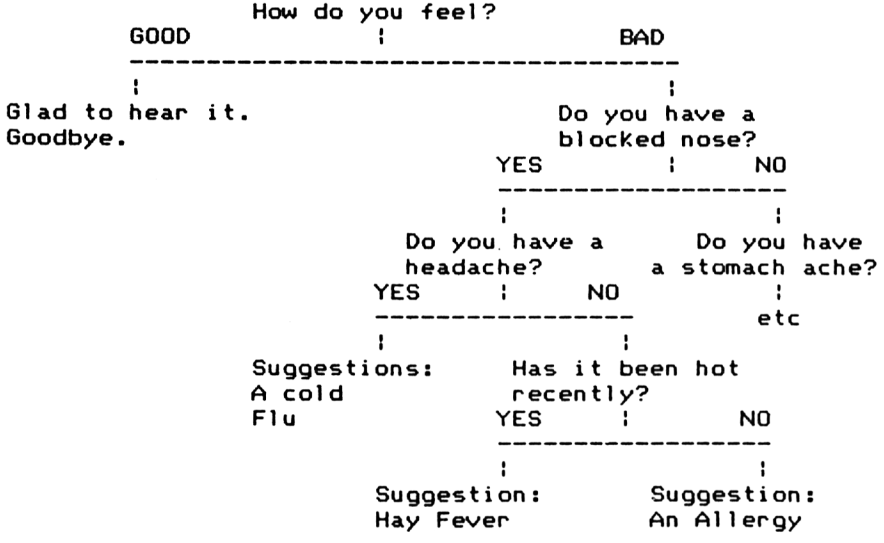


Figure 5.1 - A simple tree structure

This example is very trivial, but you should get the idea of how a system of this kind will work. The tree for a working system would obviously need to be much larger.

How would you write a program to handle this data structure in BASIC? You would probably use a large array and flip from cell to cell according to the answers given to the questions. This would work, but would use a lot of memory.

Logo is not the most economical of languages in the first place, and there's only about 8K of workspace available on the 464/664 version of DR Logo, but if there's one thing Logo is particularly good at, it's handling lists. And lists of lists. And lists of lists of lists....

What it boils down to is that Logo can handle a tree structure like this very easily, and the following short set of procedures is all that's necessary to cope with whatever tree structure you care to create. The only constraints for these particular routines are that the questions must have a simple 'yes' or 'no' answer, and that the number of 'levels' in the tree must not exceed the number of levels of recursion allowed by DR Logo.

Although it is quite possible to write similar procedures to deal with multiple choice answers, the simple 'yes/no' will serve to demonstrate the main points.

Each question in the tree is both a set of characters as in 'Do you have a blocked nose?', and also *the name of a list* in logo. The list which it names consists, in each case, of the two further questions that the 'yes' and 'no' answers to the current question will lead to.

So, in the car diagnostics system used in the following example, the list with the name `Does.engine.turn.over?` consists of `[Does.engine.fire? Power.from.battery?]`. These two questions follow from the 'yes' and 'no' answers to `Does.engine.turn.over?`. Each, in turn, is the name of a list in its own right.

This system carries on until you reach the end of one of the branches of the tree. At this stage you are told what to replace, or to refer your vehicle to a garage for further attention! The system then finishes.

The questions/list names are separated by `.`s rather than spaces because Logo treats spaces as separators. If spaces were used eg `"Does engine turn over?`, Logo would treat `"Does` as a variable name, but would look for procedures `engine`, `turn` and `over?`. Also, `[Does engine fire? Power from battery?]` would look to Logo like a list of six elements, rather than the two elements intended.

The following set of procedures will handle any general tree structure:

```
to car
ct
make "question "Does.engine.turn.over?
ask :question
end
to ask :question
type :question
if emptyp thing :question [stop]
y-or-n
if :input = "y [make "question item 1 thing :question]
if :input = "n [make "question item 2 thing :question]
ask :question
end

to thing :variable
op gprop :variable ".APV
end

to y-or-n
pr [\ (y or n)]
make "input rc
if (or (:input = "y) (:input = "n)) [pr :input stop]
y-or-n
end
```

If you chose to handle a different system from the car diagnostics, the only procedure that would need changing would be the car routine; and this would only need a different title and different initial question.

If we run through each of the four procedures above, you should be able to see how Logo handles the structure in a very simple and elegant way.

The first procedure, `car` really just sets up the screen and an initial variable `question`, and then hands over to `ask :question`. The command `ct` (clear text) is different from `cs` (clear screen), as it doesn't split the screen or display the turtle. It's used to start the system with a clear text screen.

Procedure `ask :question` is where most of the work is done. It's worth discussing it line by line:

`to ask :question` The parameter, `:question`, holds the pertinent question at each stage of the diagnosis. This is updated in line with the answer supplied to it, further on in the procedure.

`type :question` This is one of the alternative commands to

`print`. It does the same thing, but doesn't add a carriage return to the end of the object that's printed. This means that the next thing displayed, in this case the prompt `(y or n)` from the `y-or-n` procedure, continues on the same line.

Remember it's the *contents* of the variable `question` which is displayed by this line of the procedure.

`if emptyp thing :question [stop]` This is quite a complex line and deals with stopping the system when the end of one of the branches is reached. `emptyp` looks for an empty list (one containing no elements) and returns a logical value *true* if its empty and *false* if it isn't.

`thing :question` finds the contents of the variable whose name is the contents of the variable `question`. This is a tricky concept to grasp . In the present example, if `question` holds the value `Does.engine.turn.over?`, then this function looks at the contents of `'Does.engine.turn.over?'`, to see if there are any further questions. If `Does.engine.turn.over?` is empty, the whole system stops, as instructed by `[stop]`.

`y-or-n` is the procedure which accepts a keyboard input, and checks it's either `y` or `n`.

`if :input = "y [make "question item 1 thing :question]` This, and the following line, change the current question for the one contained in the list of current question names. This line deals with the `yes` response and makes the first item in the list the new question. The next line deals with the `no` response and takes the second item in the list.

The `make` command in Logo is like BASIC's LET assignment — it assigns a value to a variable. `item1 thing :question` takes the first item in the list pointed to and put it in `:question`.

`ask :question` is a recursive call to this procedure. This is why there musn't be more hierarchical levels in the tree than there are levels of recursion in the system.

That's all there is to the main body of the tree control structure. The other two procedures are convenient blocks to pull away from the main body to perform specific tasks.

`thing` is normally a Logo primitive in DR Logo, but because of the CPC 464/664's small memory area, some of the primitives have had to be omitted from the Amstrad version of the language. This short procedure is a custom-built version of `thing`, which extracts the value of a variable. When combined with `:`, as in `thing :question`, it outputs the *value of the value* of the named variable!

`y-or-n` prints the question prompt and takes in a character from the keyboard, using the `rc` (read character) command. This is assigned to the variable `input`. The `if` clause checks that the character read is either `y` or `n`. If so, the character is displayed and the procedure stops. If not, a recursive call is made to `y-or-n`.

The `if` clause may look a little alien to those used to BASIC. Logo uses prefix operators for most expressions, rather than the infix notation usually used by BASIC. This means that rather than saying 'IF INPUT$="y" OR INPUT$="n" THEN...', as you would in BASIC, Logo requires you to put the `or` first, and to enclose the two conditions in curved brackets. Once you get used to the idea, it comes quite naturally. Prefix notation is also used on a number of calculators, and in the language *Forth*.

Having set up a series of Logo procedures to handle this 'expert system', how do you set up the tree itself, which can be thought of as a small database. There's no way round typing in each relationship individually, but it's worth remembering that once you've typed it in, you can `save` it to disc as part of the Logo workspace.

Each relationship in the tree may be set up with an expression of the form:

```
make "Does.engine.turn.over? [Does.engine.fire
Power.from.battery?]
```

The `"` before `Does.etc` means you are assigning to the variable whose names is `Does.etc`, and the square brackets around the two questions tell Logo you're defining a list.

If you want to type in a set of questions for a car diagnostics system, a set follows. Bear in mind that these are my ideas, and that I've never been a motor mechanic! You should, however, be able to see the idea behind *expert systems* from this elementary example.

```
YES                                                    NO
                    Does.engine.turn.over?
Does.engine.fire?                          Power.from.battery?
                Does.engine.fire?
Does.it.fire.regularly?                 Does.HT.produce.spark?
                Does.it.fire.regularly?
Does.it.peter.out?                      Cleaned.&.gapped.plugs?
                Does.it.peter.out?
Overrich.mixture                    There.shouldn+t.be.a.problem
                Cleaned.&.gapped.plugs?
Cleaned.carburettor?                                       Do.it
                Cleaned.carburettor?
Plenty.of.petrol?                                          Do.it
```

```
                        Plenty.of.petrol?
Checked.fuel.lines?                                              Do.it
                      Checked.fuel.lines?
Obscure.fault...refer.to.garage                                  Do.it
                    Does.HT.produce.spark?
Replace.plugs                                          Checked.coil?
                        Checked.coil?
Checked.HT.leads?                                                Do.it
                      Checked.HT.leads?
Checked.contact.breakers                                         Do.it
                    Checked.contact.breakers?
Does.distributor.shaft.rotate?                                   Do.it
                  Does.distributor.shaft.rotate?
Obscure.fault.refer.to.garage.                     Refer.to.garage
                      Power.from.battery?
Does.starter.motor.engage?                          Checked.leads?
                    Does.starter.motor.engage?
Does.starter.motor.turn?                         Replace.solenoid
                     Does.starter.motor.turn?
Major.fault.refer.to.garage                Replace.starter.motor
                        Checked.leads?
Replace.battery                                                  Do.it
```

Each of the final statements in the tree (Do it, Replace plugs etc.) should be given an empty list, [ ], as its contents. For instance, `make "Do.it [ ]`.

This brief taster for Logo will, I hope, have made you want to explore the language further. There are many other serious applications for Logo which we haven't room to explore here, and also much scope for 'playing around', either with turtle graphics or Logo's list processing functions. The language was designed to encourage play, at all levels, and is, above all, great fun to use.

Pascal

In the 30-year history of commercial computing, a number of programming languages have been developed for specific purposes. One of the earliest of these was Fortran, which started its life in the laboratories of IBM, and was mainly concerned with number-crunching, or dealing with scientific calculations. It was well suited to this purpose, but was felt in some quarters (mainly academic ones), to be too free-and-easy in its approach to instill the necessary discipline on the program, and ensure a good structured programming style. (Fortran was one of the immediate predecessors of BASIC).

An alternative language, ALGOL, was therefore developed by a committee of computer scientists from several different organisations. Although not very successful in its own right, ALGOL had influenced a number of more recent languages, one of which was Pascal.

Pascal was designed by one man, Nicklaus Wirth, who set out to overcome the faults that he saw in ALGOL, and design a computer language which would have a solid structure and tight syntax (the way commands are put together). To this extent Pascal is rather different from BASIC, which is primarily easy to learn. Although Pascal, too, is designed to make learning a computer language simple, this is done by imposing a rigid set of programming disciplines, rather than letting the would-be programmer 'get stuck in' and do the job however he or she can.

Someone dabbling their toes in the Pascal pond might at first find the water rather icy, but once they take the plunge, they'll probably find swimming quite invigorating! There is not really space in this book for much more than a quick dip, but, hopefully, most of the main philosophies of Pascal can be demonstrated.

Pascal comes in several versions, two notable ones being UCSD, which was developed at the University of California at San Diego, and ISO, which is a 'standard' version produced by the International Standards Organisation. The version most easily available for the Amstrad micros comes from Hisoft — who also produce *Devpac 80*, which was covered in Chapter 4.

The company produces two versions of the language, substantially the same, to run under AMSDOS and CP/M. The version used here is Pascal 80, which is designed to run on any CP/M-based, 8 bit micro. It consists of a language *compiler*, and an editing program which is, in fact, ED80, the editor supplied with Devpac 80.

The idea of a *compiled* language may be new to you, as it is a fundamentally different concept to *interpreted* languages, like BASIC and Logo.

All high-level languages (anything other than assembler) have to be translated from the words they use to represent their commands (`PRINT`, `FOR`, `fd`, `rt` and the like) to machine code, which is at the base level that the computer can understand directly. This translation is carried out by either an *interpreter* or a *compiler*.

An interpreter, as used by BASIC and Logo (among others), translates each line of program *as the program is run*. Thus, in the BASIC line:

```
FOR N%=1 TO 10:PRINT "Hello":NEXT N%
```

each section of the line will be read and translated 10 times by the interpreter. Even when this process is carried out quickly, as by the Locomotive interpreter used on the Amstrad machines, it is still a much

slower process than running a compiled program.

A compiled language, such as Pascal, uses a different system. After you've written and debugged your program in Pascal (the source program), you submit it once and for all to the compiler, which translates it to a machine code (object program) equivalent, which can then be run in its own right. The machine code program obviously runs much faster than an interpreted program, since the translation work has already been done. It won't be as fast as custom-designed machine code, because the compiler is a 'computerised' translator, but the machine code program will run typically 40–50 times faster than BASIC.

Where the compiled language loses out to the interpreted one, however, is in the debugging or modification process. An interpreted BASIC or Logo program is held in its original form in the memory of the computer, and if you need to make changes to the program you can list it and alter the wording directly.

With a compiled program, however, you have to discard the flawed machine code program, reload the untranslated original, make your alterations to that and recompile it. This is a much more long-winded procedure and is the penalty you pay for the extra speed your program will eventually have.

Several of the control structures in Pascal will be familar to those who've programmed in Locomotive BASIC. The FOR loop is there, as are WHILE, IF...THEN...ELSE and GOTO, although each structure is more powerful in Pascal than in BASIC. Pascal also adds several new structures. You now have REPEAT...UNTIL and CASE...OF, as well as the chance to use multi-line procedures and functions not available in BASIC.

Pascal is a procedure-oriented language, which means it strongly encourages you to break your program up into manageable blocks, and to write these as independent procedures. The cynic starting to program in Pascal might say that breaking a program into small blocks is the only way you'd ever get any of it past Pascal's fussy syntax checking!

A procedure can be loosely thought of as a subroutine, although it is called by name and is a lot more versatile to use than a typical BASIC subroutine. A Pascal *function* is similar to a procedure, except that it returns a value to another part of the program at the end of its working.

The shortest Hisoft Pascal program you can write, and one which does nothing, is:

```
PROGRAM name;
BEGIN
END.
```

Every Pascal program must have these four essential parts — more useful

bits can also be added at your discretion! The program must be identified by name, must have a beginning and end statement, and must finish with a full stop (and only one). The semicolon at the end of the first line is important, as is all punctuation in Pascal. The semicolon is used to mark the end of a program section.

Most versions of Pascal allow both upper and lower case letters to be used in keywords and variables, but Hisoft expects keywords to be in capitals. To make the example programs in this section easier to read, all variable names will be printed in mixed case.

To illustrate some of the syntax and structures used within Pascal, we'll develop a short program to calculate and display the Exclusive OR product of all the numbers from 0 to 15. If that sounds rather esoteric and not a lot of use, then bear in mind that when you try to animate graphics on your Amstrad's screen, the most efficient way is to XOR the pixels of your graphic character with those already on the display. Many commercial programs use this technique, and when you're trying to work out the colours you need to use to get the right effects, a table of XOR products is extremely useful.

The main problem with writing a Pascal program to draw up this table is the fact that Pascal doesn't have an Exclusive OR function, like XOR in Locomotive BASIC. In fact, none of the Boolean functions that BASIC offers will return anything other than true or false values in Pascal. This means a custom-made function will have to be written.

The basic idea of the program will be to run two, nested loops, each from 0 to 15, to convert each pair of numbers to their binary equivalents in an array, and to Exclusive Or each binary bit of the two numbers together, to produce a third number. This is then reconverted from binary to decimal and forms one of the numbers in the table.

This structure leads naturally to four main program parts:

1. A decimal to binary conversion
2. An Exclusive OR of two binary numbers
3. A binary to decimal conversion
4. A controlling section which will also display the table on the screen

The first two of these tasks will be programmed as procedures, the third will be a function, and the last will form the main body of the program. The two conversion routines will be written first, and tested separately before combining them with the rest of the program.

This kind of modular programming and testing is encouraged by Pascal, and is ideally suited to a team of programmers working on a large involved program. Even on a program of this length, it helps to split the problem

down into a number of easily managed parts.

One of the main differences between BASIC and Pascal is that Pascal demands you declare all the variables and constants you're going to use, *before* you start assigning values to them. This is partly due to its philosophy of making the program self-explanatory, but it also makes life a lot easier for the language's compiler.

A Hisoft Pascal program is easily written using their ED80 full-screen editor, supplied with the package, although any CP/M editor or word processor could be used (ED or *Wordstar*, for instance). ED80 has already been described in some detail in Chapter 4.

When you type in a Pascal program, it is useful to indent each loop within the program, so that the structure can be more clearly seen. The sample program used here has each loop indented by two characters. These are typed in by hand on ED80, but you could alternatively define Tab stops to do it for you. The Pascal compiler ignores these indentations, so you're not obliged to use them.

Each Pascal program, procedure and function has to have a heading, which includes the title, **PROGRAM**, **PROCEDURE** or **FUNCTION**, the name of that program or section, and, in the case of procedures and functions, any parameters which will be passed to them.

The concept of a parameter may be new to those who have only been used to BASIC until now. The idea is used in Logo and was discussed in that section, but is more developed in Pascal. Users of BASIC will, in fact, have used parameters already, without really thinking about it. A lot of BASIC's built-in functions take parameters; think of MODE, PEN, LEFT$, PRINT as a few examples. Each of these requires a number, variable or piece of text on which to act. These parameters are passed to the section of code which performs the function, so they've got something to work on.

In Pascal, you can define your own procedures and functions, and can also define the number and type of parameters you can pass to them. When you want to call a procedure or function within the program, you simply state its name, and supply the appropriate parameters. You must supply the same number of parameters as you defined for it, and they must be of the same type as the procedure needs. If your procedure is defined as:

```
PROCEDURE test(i:integer);
```

it's no use calling it with the line:

```
test('a');
```

Test expects an integer number as its parameter, and won't be able to deal with a character such as 'a'. In fact, an error like this would never get past the

compiler, which would signal it and stop the compilation.

Having covered a few of the preliminaries, here is the first procedure in the XOR program, which converts a decimal number into a pseudo-binary one:

```
PROCEDURE DecToBin(Index,Number:INTEGER);

VAR Test,Pointer:INTEGER;

BEGIN
  Test:=128;
  Pointer:=0;
  WHILE Test>0 DO
  BEGIN
    IF Number>=Test THEN
    BEGIN
      Array[Pointer,Index]:=1;
      Number:=Number-Test;
    END
    ELSE Array[Pointer,Index]:=0;
    Test:=Test DIV 2;
    Pointer:=Pointer,1; END
  END;
```

The procedure works by comparing a number, `Number`, with a series of values held in `Test`. `Test` is divided by two on each pass through the `WHILE` loop, and if `Number` is greater than `Test` a 1 is put in an element of the array `Array`. At the end of eight passes round the loop, the array will hold a series of 1s and 0s, representing the 8-bit binary equivalent of `Number`. This representation might be called a pseudo-binary number.

This procedure may look rather confusing to you, but we'll run through each line and explain what it means.

The heading gives the procedure the name `DecToBin`, which describes what it does. Note that you can use upper or lower case characters in procedure and function names. The two parameters which will be passed to the procedure will be put in the variables `Index` and `Number`. Both parameters will be integers.

Every time you declare a variable in Pascal, you have to specify its type. Hisoft Pascal supplies four main pre-defined variable types: CHARACTER, BOOLEAN, INTEGER and REAL. A character type may contain any ASCII characters, similarly to a string in BASIC. Boolean variables can only contain the logical values *true* and *false*. Integers are whole numbers only, and reals are floating point numbers.

Although there isn't space here to discuss the idea fully, Pascal allows you to define you own types of variables to supplement these four. You could, for

instance, define the type POSINT, which could only contain positive integers, or the type DAYSOFTHEWEEK, which could only contain 'Monday', 'Tuesday', 'Wednesday' etc. This facility, which may be hard to grasp at first, is very useful for limiting the values that variables may contain.

As well as the two variables which will have values passed as parameters to `DecToBin`, the procedure will also use two other variables, `Test` and `Pointer` (again both integers), within it. All four of these variables are *local* to the procedure. This means that they only have values while the procedure is working. Once the program returns to its main section, or passes to another procedure or function, their names and values are forgotten.

This facility is valuable to both the programmer and the language. From your point of view, as the programmer, you don't need to worry about clashes between variable names. If you define variables `Index` or `Test` in this procedure, or include then in the parameter list, they are *not* the same as identically named variables in any other part of the program. You can define variables to be used in any part of the program by including them at the beginning, under the `PROGRAM` header. The array `Array`, used in this procedure, is defined in this way. A variable or array available to the whole program is known as *global.*

From the program's point of view, the use of local variables means that it doesn't have to allocate space for every variable used in the program. As local variables can be forgotten outside the environment of their defining procedure or function, the same space can be re-used for the next set of variables. The only variables which have to be allocated space for the duration of the program are the global ones.

The `BEGIN` and `END` markers are used to divide up discrete blocks of a Pascal program. They mark the beginning and end of the main block of this procedure, and the loop and multi-statement `IF...THEN` clause. They are important to the syntax of Pascal, and help the layout of the program, so you can see more easily what's going on.

The two variables `Test` and `Pointer` are assigned values using the `:=` symbol. This is used by Pascal to distinguish between mathematical equivalence, and the assignment of a value to a variable. The main use of `=` on its own is in `IF...THEN` statements, such as `IF Number>=Test THEN`, later in this procedure.

The `WHILE...DO` statement is the only loop in the procedure. It will execute eight times, as `Test` will reach zero before the ninth loop. A `FOR...TO...DO` loop could have been used instead.

The main part of the procedure consists of an `IF...THEN...ELSE` clause, and it's important to note that the statements following `THEN` are separate statements. In BASIC all the statements following a similar

IF...THEN clause would have to be on the same line, separated by colons. The limit in BASIC is therefore the length of a single BASIC line, normally 255 characters.

In Pascal, however, because of the use of BEGIN and END, the THEN clause may be as long as needed. Control of the program will still pass to the statement after END, if the IF condition is not met, or to an ELSE clause, if there's one there.

The assignment statements within the IF...THEN...ELSE clause are fairly straightforward, using the := operator. An array in Pascal is marked out with square brackets rather than round ones. As all the variables are integers, DIV must be used rather than /.

Semicolons must again be used to separate each discrete clause. Pascal allows you to lay out your program as you like - you could type the whole thing as one long line - but the compiler needs to know where one statement ends and the next begins.

Pascal ensures this by being fussy about the use of punctuation, and particularly the semicolon. If you forget any in your Pascal program, the compiler will probably throw up all kinds of odd errors, which at first don't appear to relate to the program you're compiling. You will often find, however, that Pascal has been trying to read two lines of program as one, because you've forgotten to tell it to stop by putting in a semicolon.

The function BinToDec does the reverse of DecToBin. It takes a pseudo-binary number in an array and adds multiples of two to a decimal number, depending on each pseudo-bit value, finally reaching a decimal equivalent. It works in a similar way to the procedure just described. The listing looks like this:

```
FUNCTION BinToDec(Index:INTEGER):INTEGER;

VAR Test,Number,Pointer:INTEGER;
BEGIN
  Test:=128;
  Number:=0;
  FOR Pointer:=0 TO 7 DO
  BEGIN IF Array[Pointer,Index]=1 THEN
    Number:=Number+Test;
  Test:=Test DIV 2
  END;
  BinToDec:=Number;
END;
```

Much of the function is similar to the procedure DecToBin. The differences are mainly the differences between a Pascal procedure and a function.

The fundamental difference between a procedure and a function is that a function returns a value. So, in the function header, the parameters are defined as before (`Index` in this case), but the result of the function — the value passed back to the part of the program which called it — must also be given a type. The function `BinToDec` takes in a parameter `Index`, which is an integer, and passes back a value which is also an integer. The first line of the function defines both of these in a simple and explicit way.

The penultimate line is the command which passes the result of the function back to the calling program segment. The name of the function is assigned the value of the variable which holds this result.

Before progressing to the rest of the program, it's a good idea to check the procedure and function already defined, by contriving a test program. This will contain a program header and statements to accept entry from the keyboard and output results to the screen. Suitable program segments look like this:

```
PROGRAM Check;

VAR Number, Index:INTEGER;
  Array:ARRAY[0..7,0..2] OF INTEGER;
```

(The procedure `DecToBin` and function `BinToDec` are inserted here.)

```
BEGIN
  WRITE('Enter number (0-255) for conversion: ');
  READ(Number);
  DecToBin(0); FOR Index:=0 TO 7 DO
  WRITE(Array[Index,0]);
  WRITELN;
  WRITELN('Press any key to reconvert');
  REPEAT
  UNTIL INCH<>CHR(0);
  Number:=BinToDec(0);
  WRITELN(Number);
END.
```

Procedures and functions in Pascal are included before the main part of the program, so that, when the program is compiled, the compiler knows what to do when procedures are called from the main program. There are some circumstances when it isn't possible to define all procedures and functions before they're called, and it's possible to tell the compiler to allow a forward reference to a procedure not yet defined. This is done by using the `FORWARD` keyword.

The three global variables defined in the program `Check` are `Number`, `Index` and `Array`. `Array` is a two – dimensional array, containing integers. Note how the minimum and maximum array subscripts are

defined. You can specify the values you want them to run from and to. Note also that the variables `Number` and `Index` are *not* the same as those used in `DecToBin` and `BinToDec`.

The program itself contains four statements not yet described, `WRITE`, `READ`, `REPEAT...UNTIL` and `INCH`.

`WRITE` in Pascal is very similar to BASIC's PRINT statement. It transfers the following text or variables enclosed in brackets, to the screen. `WRITELN`, which is short for 'write line', is very similar, except that it prints a line feed/carriage return at the end of the text. Items inside the `WRITE` statement's brackets are separated by commas.

`READ` is similar to BASIC's INPUT statement, but it doesn't issue a '?' prompt. The input from the keyboard will be assigned to the variable name inside the brackets, and must be of the right type.

The `REPEAT...UNTIL` loop is similar to the `WHILE...DO` loop, except that the test for completing the loop is at the end, rather than the beginning. This means that the loop will always execute at least once. In the program above the loop is empty, and it's used simply to scan the keyboard repeatedly, until a key is pressed.

The `INCH` function is special to Hisoft Pascal, and is a pre-defined library function which scans the keyboard. If a key is pressed, the associated ASCII character is returned, otherwise CHR(0) is returned instead.

The line `Number:=BinToDec(0);` calls the function `BinToDec` and assigns the result to the global variable `Number`.

To compile the program to test `DecToBin` and `BinToDec`, you type the full listing into ED80, then leave the editor using <CTRL>K X. You will be asked to specify a filename, and this can be anything you like, as long as the filetype is `.PAS`. The Hisoft compiler won't recognise a file with any other filetype.

You call the compiler by typing `HP80 filename;option` at the `A>` prompt. If you are happy with the compiler's default options, you can just type the filename. The compiler will then be loaded and start compiling the program whose filename you've specified.

If all goes well, a listing of the program will appear, with the end address of the machine code generated by the compiler displayed at the bottom of the listing. Another file will have been created on your disc, using the same filename as your original program, but with the filetype `.COM`. You can run this machine code program by typing the filename, as with any other CP/M `.COM` file.

If you've made a syntax error in your program, which is much more likely in Pascal than in BASIC, the compiler will stop the listing at the statement after the mistake has been spotted, and report the error number. One error in a Pascal program tends to generate a 'snowball' effect, and a number of other, sometimes false, errors will be signalled. Correcting the original mistake will often remove several others from the compilation.

As already mentioned, Pascal checks the construction of your program very exactly, and will signal most errors at compilation time. As further evidence of this, Hisoft Pascal lists 73 errors which may show at compilation, but only 13 that can occur when you run the program.

If your program has errors in it, you'll have to call ED80 again with the original `.PAS` file, and correct them there. This process of compilation, correction and recompilation goes on until the program is free of compilation errors. This can be a slow process, particularly when you first start to use Pascal, but when you finally get an error-free compilation, the program is far more likely to run correctly first time than an equivalent BASIC program.

There are a number of options available within the compiler, and these can be used to modify the resulting machine-code. Three that are worth mentioning are `P`, `R`and `Y`. Each is added after the filename, separated by commas. `P` directs the listing produced by the compiler to the printer, `R` prevents the routines handling REAL numbers being added to the machine-code program (which makes the program shorter), and `Y` deletes any existing `.COM` file with the same name. (If you're recompiling, then you should use this option, otherwise the compiler will stop when it finds the existing file on the disc.)

When you run the program `Check`, you will be asked to enter a number, which will then be converted to a string of binary characters. When you next press a key, the binary number will be reconverted and your original number will be displayed. A typical run looks like this:

```
A>CHECK
Enter number (Ø-255) for conversion: 57
ØØ111ØØ1 Press any key to reconvert
57

A>
```

The final procedure in the XOR program, exclusively ORs two pseudo-binary numbers from `Array`, and puts the result in a third section of the array.

When you exclusively Or two bits, the result is only '1' if one of the bits is '1' and the other isn't. In other words, if the two bits are different from each other the result is '1', otherwise it's '0'. This makes it very easy to devise a

procedure to do the job:

```
PROCEDURE ExOr(Number:INTEGER);

VAR Pointer:INTEGER;

BEGIN
  FOR Pointer:=Ø TO 7 DO
  BEGIN
    IF Array[Number,Ø]=Array[Number,1]
      THEN Array[Number,2]:=Ø
    ELSE Array[Number,2]:=1
  END;
END;
```

The main part of the program deals with the formatting of the table on the screen, and calls the other three parts of the program to do the work. The complete listing of the 'XOR' program is given in Figure 5.2. The only new command is another Hisoft predefined procedure, **PAGE**, which clears the screen.

```
PROGRAM XOR;

VAR Xpos,Ypos,Number:INTEGER;
    Array:ARRAY[0..7,0..2] OF INTEGER;


PROCEDURE DecToBin(Index,Number:INTEGER);

VAR Test,Pointer:INTEGER;

BEGIN
  Test:=128;
  Pointer:=0;
  WHILE Test>0 DO
  BEGIN
    IF Number>=Test THEN
      BEGIN
        Array[Pointer,Index]:=1;
        Number:=Number-Test;
      END
    ELSE Array[Pointer,Index]:=0;
    Test:=Test DIV 2;
    Pointer:=Pointer+1;
  END
END;
```

```
FUNCTION BinToDec(Index:INTEGER):INTEGER;

VAR Test,Number,Pointer:INTEGER;

BEGIN
  Test:=128;
  Number:=0;
  FOR Pointer:=0 TO 7 DO
  BEGIN
    IF Array[Pointer,Index]=1
      THEN Number:=Number+Test;
    Test:=Test DIV 2;
  END;
  BinToDec:=Number
END;


PROCEDURE ExOr(Number:INTEGER);

VAR Pointer:INTEGER;

BEGIN
  FOR Pointer:=0 TO 7 DO
  BEGIN
    IF Array[Pointer,0]=Array[Pointer,1]
      THEN Array[Pointer,2]:=0
    ELSE Array[Pointer,2]:=1;
  END;
END;

BEGIN
  PAGE;
  WRITELN('     Table of EXCLUSIVE OR values from 0 to 15');
  WRITELN;
  WRITELN('     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15');
  FOR Ypos:=0 TO 15 DO
  BEGIN
    WRITELN;
    IF Ypos<10 THEN WRITE(' ');
    WRITE(Ypos,' ');
    DecToBin(0,Ypos);
    FOR Xpos:=0 TO 15 DO
    BEGIN
      DecToBin(1,Xpos);
      ExOr(Xpos);
      Number:=BinToDec(2);
      IF Number<10 THEN WRITE(' ');
      WRITE(Number);
    END;
  END;
END.
```

Figure 5.2 - Complete listing of sample Pascal XOR program.

The table that the XOR program produces is shown in Figure 5.3. If you don't feel the urge to program in Pascal, I hope, at least, you might find the table of use.

```
Table of EXCLUSIVE OR values from 0 to 15

    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

 0  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 1  1  0  3  2  5  4  7  6  9  8 11 10 13 12 15 14
 2  2  3  0  1  6  7  4  5 10 11  8  9 14 15 12 13
 3  3  2  1  0  7  6  5  4 11 10  9  8 15 14 13 12
 4  4  5  6  7  0  1  2  3 12 13 14 15  8  9 10 11
 5  5  4  7  6  1  0  3  2 13 12 15 14  9  8 11 10
 6  6  7  4  5  2  3  0  1 14 15 12 13 10 11  8  9
 7  7  6  5  4  3  2  1  0 15 14 13 12 11 10  9  8
 8  8  9 10 11 12 13 14 15  0  1  2  3  4  5  6  7
 9  9  8 11 10 13 12 15 14  1  0  3  2  5  4  7  6
10 10 11  8  9 14 15 12 13  2  3  0  1  6  7  4  5
11 11 10  9  8 15 14 13 12  3  2  1  0  7  6  5  4
12 12 13 14 15  8  9 10 11  4  5  6  7  0  1  2  3
13 13 12 15 14  9  8 11 10  5  4  7  6  1  0  3  2
14 14 15 12 13 10 11  8  9  6  7  4  5  2  3  0  1
15 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

Figure 5.3 - Table of Exclusive or values produced by XOR program.

The purpose of the discussion has been to give a flavour of the Pascal language, using a well-provisioned, modern version of the language currently available for the Amstrad micros. There's no doubt that Pascal can produce fast, efficient programs, and that it's strict demands on the layout of the program can lead to clear and easy-to-read listings. For someone used to the laid-back demands of BASIC, though, Pascal may seem excessively fussy. It's very much a question of horses for courses, and using the right language for the right application.

A third option, which allows some of the freedom of BASIC while retaining some of the structure of Pascal, is 'C'. There is now a version of this language available under AMSDOS, and we'll have a brief look at this in the next section.

C

At first glance, C seems to embody many of the features of languages like Pascal, but, in fact, their design philosophies are quite different. Where Pascal tries to regulate the construction of a program by enforcing strict checks on syntax and form, C will allow you to do virtually anything you like. If anything, it is more 'free form' than BASIC.

There is a penalty to pay for this freedom, however. Because very little checking is performed by the C compiler, any bugs in the program are your responsibility — and bugs in C can become very obscure.

Nonetheless, C has become a very popular language among systems and application programmers. A lot of modern microcomputer software is written in C, because it offers the versatility and speed they need. In many cases, applications that would once have been programmed in assembler are now written in C. Although not as fast as 'hand-written' machine code, C is sufficiently speedy to do many jobs, particularly with the steadily increasing speed of microprocessors.

The only commercial version of C that will run happily on the Amstrad micros is Hisoft C. This implementation has been written to work under AMSDOS, rather than CP/M. It includes most of the features of a full version of C, the main exception being the lack of floating-point arithmetic. This means that the current version of the language will only deal with integers (whole numbers). For many applications, however, this won't be a problem.

C consists of a 'hard core' of functions, which is supplemented by a set, or sets, of library functions, which can be as general or specific as are needed. Hisoft C comes with three libraries of functions, a basic set common to most versions of the language, and two further sets specific to the Amstrad micros. Both of these sets mimic functions available in Locomotive BASIC.

The first library deals with general functions and sound commands, while the second covers graphics. Whenever you want to include one of these library routines in a Hisoft C program, you simply call it by name, and tell the compiler to include the appropriate library file when it compiles the program.

When you look at a C program for the first time, you are quite likely to find it full of odd symbols, scattered freely through the code. C uses a number of different symbols as shorthand for operators which are given names in other languages. For example, `!` means NOT, `&` means AND, and `:` means OR. These operators can be combined with comparison markers, so, in C, you could write:

```
if x!=1 & y!<2 : z=3
```

which would translate as `if x is not equal to 1 and y is not greater than 2 or z equals 3`. The C version is certainly shorter!

As well as shortening logical operators, C offers more efficient ways of assigning values to variables. Although you're quite at liberty to write `x=x+1` in C, you can also write `x+=1`, or even `x++`. All three of these statements will add one to the value of x. The difference between them is in the way the compiler treats them. The two alternative ways of writing `x=x + 1` are progressively more efficient, because they're closer to the way the operation would be carried out in machine code, and it's into machine code, of course, that a C program is compiled.

C uses several of the structures common to BASIC and Pascal. It has the `if...else` clause for conditional statements, and the `while` and `for` loops. There's even a `goto` statement, but this transfers control to a defined label rather than a line number. In addition to these structures, C has the `do` loop, which puts the condition at the end, in a similar way to Pascal's `repeat...until`.

There's also the `switch` statement which transfers control to any of several sections of the program, depending on the value of a `case label`. Finally, the `break` statement will close a `while`, `do`, `for` or `switch` statement prematurely. If these loops are nested, `break` will only close the loop most immediate to the `break` statement.

As an example of a short C program, Figure 5.4 shows a mathematical routine to calculate prime numbers, using the 'Sieve of Eratosthenes' algorithm. In its original form, it was published in the USA by 'Byte' magazine, as a benchmark (speed test) for different versions of C.

```
#define SIZE 8190
#define FALSE 0
#define TRUE 1

char flag[SIZE+1];

main()

  static int i,j,count,prime;

  count=0;
  for (i=0;i<=SIZE;i++)
    flag[i]=TRUE;
  for (i=0;i<=SIZE;i++)

    if (flag[i]==TRUE)

      prime=i+i+3;
      for (j=i+prime;j<=SIZE;j+=prime)
        flag[j]=FALSE;
      count++;


  printf("%d primes. n",count);
```

Figure 5.4 C program for 'Sieve of Eratosthenes'

The program uses a number of `for` loops to 'sieve out' all the numbers which are not divisable by any but themselves (and one), and so are prime. The program as it stands will calculate all the primes between 3 and 16384, a total of 1899 numbers. It does this in 6.2 seconds. A similar program, written in Locomotive BASIC, is shown in Figure 5.5.

```
10 ntimes=10
20 size=8190
30 false=0
40 true=1
50 DIM flag (size+1)
60 DEFINT c,i,j,p
70 count=0
80 FOR i=0 TO size
90 flag(i)=true
100 NEXT i
110 FOR i=0 TO size
120 IF flag(i)=true THEN prime=i+i+3:FOR
 j=i+prime TO size STEP prime:flag(j)=fa
lse:NEXT j:count=count+1
130 NEXT i
140 PRINT count;" primes."
150 END
```

Figure 5.5 BASIC program for 'Sieve of Eratosthenes'

You can see that the structure of the two programs is almost identical. The BASIC program, however, takes 174.0 seconds to find the same 1899 numbers. The comparison is perhaps a little unfair, as there is very little screen activity, which slows all programs down and is often the governing factor in determining the overall speed. Nevertheless, the comparison does serve to show the kind of speed increase that is possible when using a compiled language.

A few of the other features of C can be seen be examining the program in Figure 5.4.

A C program consists of a series of functions, which may or may not take parameters or return a value. Any parameters are enclosed in round brackets, and if none are needed, the brackets are still used, but left empty.

The simplest form of a C program looks like this:

```
main()
{
}
```

where `main` is actually a function name. There must always be a function called `main` in a C program. This is the controlling function for the program, from which other functions may be called.

The two curly brackets `{` and `}`, known as *braces*, represent the start and finish of the body of the function, and are equivalent to the keywords BEGIN and END in Pascal. As with other things in C, though, they are abbreviated to save typing and to keep the program concise.

The first three lines of the C Sieve program define constants to be used later in the program. The `#define` command is, in fact, a directive to the compiler to store the values '8190', '0' and '1' and replace the three words `SIZE`, `FALSE` and `TRUE` with them, whenever they're encountered in the source program.

The next line defines an array. Arrays in C are rather different from similar structures in BASIC or Pascal. In C, an array is an area of memory set aside by the compiler, with enough bytes to hold a specified number of single-byte characters, or two-byte integers.

A character array, which is defined by preceding the dimensions of the array with the word `char`, will contain one less byte than the number used to specify it. In this case the number is `SIZE+1`, so there will in fact be 8190 bytes reserved in the array `flag`. The last byte in an array is always set to 0, to mark the end of it. As in Pascal, arrays in C are marked out with square brackets, and statements are completed with semicolons.

The next two lines define the beginning of the program, which in this case is made up of only one function, `main`. The first line of `main` defines four integer variables, `i`, `j`, `count` and `prime;`. `count` is set to zero in the next line. This variable counts the number of primes found by the program.

A loop is set up in the next two lines which sets all the elements of `flag` to true (1). A `for` loop in C consists of the word `for` and three elements, enclosed in brackets, representing the start, finish and increment values. This loop uses `i` as a counter, and `i` is initially set to 0. The counter is incremented by one on each loop `(++i)`, until `i` is greater than `SIZE`. This loop is directly equivalent to the FOR loop in line 110 of the BASIC program in Figure 5.5.

An identical loop is set up in the next line, and within this loop is a multi-line `if` clause, which does most of the work of the program. Note the `==` sign used in the conditional test. C uses `=` for assignment, and `==` for comparison. It's easy to get confused when you start using the language.

The `for` loop within this `if` statement has an increment of `prime`, rather than 1, and this is indicated as `j+=prime`. The equivalent BASIC is 'STEP prime'.

The `printf` statement at the end of the loops is typical of the way C handles display. The *literal* part of the statement is put first, inside inverted commas (""). Any variable included in the line is represented by a `%`, followed by a letter indicating the number base in which you want it printed. C will print a numeric variable in octal, decimal, or hex, using the letters `o`, `d` or `x` after the `%`. Other characters are used to indicate characters and strings.

After the literal or control part of the `printf` statement come the names

of the variables, in the order you want them printed.

The Hisoft C editor mimics the operation of the Locomotive BASIC editor, so it's easy to work with. A number of single letter commands control its operation. `i` allows you to insert lines of program, `d` will delete them, `l` will list them, `p` puts them to disc (saves them), `g` gets them back again (loads them) and `w` writes them to your printer. In addition, the `f` and `s` commands will find a given string of characters, and substitute another string, respectively.

The `e` command enters the line editor, and you can then move forwards and backwards through a line of your program, inserting or overwriting the existing text. You can delete characters to the left of or directly under the cursor. Pressing <ENTER> incorporates the changes into your program, while <ESC> abandons the current alterations and retains the original line.

If you have a CP/M-based text editor or word processor, you can use this to edit a Hisoft C program, as long as you remember to save it with the filetype `.C`, and to switch back to AMSDOS before running the compiler. The advantage of working with a full-screen editor may be outweighed by the added complication of switching from CP/M to AMSDOS for each compilation.

The editor uses line numbers in a similar way to BASIC while you're entering or editing your program, but they disappear during compilation, and are only there for your convenience. The `n` command will renumber your program for you.

As well as including library routines in your program, you can combine functions of your own at compilation time, and build up a program from individual modules, developed and saved separately.

Although C is more forgiving than Pascal in its syntax, there are still many compilation errors which may be reported when you compile a program. As with Pascal, some errors will confuse the compiler into producing further 'false' reports. The best technique is to tackle each error in turn, and see if correcting one will remove others further on in the program.

When you have successfully compiled a C program, you tell the compiler you've finished by typing <CTRL>Z at the keyboard, and you can then run the compiled program.

You can convert your Hisoft C program to run on any Amstrad micro, even one without C running on it. This opens the opportunity of writing programs for sale to other users.

6128 Special

All three of the languages decribed so far will run on the CPC 6128 using CP/M 2.2, as you would expect. HiSoft C runs under AMSDOS anyway, so will run in exactly the same way under either system. Logo and Pascal, though, are sufficiently different to warrant some extra notes on their use under CP/M Plus.

Dr Logo

As mentioned earlier in this chapter, Dr Logo as supplied with the CPC 464/664 is a rather 'cut down' version of the full language. With the extra memory available on the CPC 6128, a full version of the language is supplied. This has many more Logo commands available, and the extra ones are shown below:

arctan - outputs the arctangent, in degrees, of the input number

changef - renames a Logo disc file

copyoff - stops sending text to the printer

copyon - starts sending text to the printer

cursor - outputs a list containing the cursor position

define - makes the input list, the definition of the given procedure name

defaultd - outputs the name of the current default drive

dirpic - outputs a list of picture files from the specified drive

dotc - outputs the colour number of the specified co-ordinate list

edall - loads all the variables and procedures in Logo's workspace into the screen editor

edf - loads the named disc file into the screen editor

erall - erases all current procedures and variables from the workspace

fill - fills an area of the screen in the current pen colour, starting at the position of the turtle

home - moves the turtle to the centre of the graphic screen, heading north

last - outputs the last element of the input object

lc - outputs the input word in lower case

listp - outputs TRUE if the input object is a list

loadpic - loads and displays the named picture file

lput - adds the first input object to the end of the second input object, and outputs the result

memberp - outputs TRUE if the first input object is included in the second input object

namep - outputs TRUE if the input word is the name of a defined variable

noformat - removes procedure formatting, and comments to free workspace

notrace - turns off the procedure trace monitor

nowatch - turns off the procedure watch monitor

numberp - outputs TRUE if the input object is a number

piece - outputs an object containing given elements of the input object

poall - displays the definitions of all defined procedures and variables

pons - displays the names and values of all global variables

pops - displays the names and definitions of all procedures

pps - displays the 'non-standard property pairs' of all defined objects

quotient - outputs the division of the two integer input numbers

remainder - outputs the integer remainder of the division of the two integer input numbers

rerandom - makes the random number function 'random' repeat a previous set of random numbers

round - outputs the input number rounded to the nearest integer

savepic - saves the current graphics design in the named disc file

setbg - sets the graphics background to the colour given by the input number

setcursor - moves the cursor to the point specified by the input co-ordinate list

setd - makes the given drive the default drive

setscrunch - sets the aspect ratio of the graphic screen to the input number

setx - moves the turtle horizontally to the x co-ordinate given

sety - moves the turtle vertically to the y co-ordinate given

shuffle - outputs a list containing the elements of the input list, in random order

shuffle - outputs a list containing the elements of the input list, in random order

thing - outputs the value of the variable whose name is the input

text - outputs the definition of the given procedure

towards - outputs a heading to make the turtle face towards the given x,y position

trace - turns on the procedure trace monitor

uc - outputs the input word in upper case

watch - turns on the procedure watch monitor

where	-	outputs the start position of the included elements from the most recent memberp expression
.in	-	fetches the value of the port with a given number
.out	-	sends the given value to the port with a given number
.ENL	-	end of a procedure line, broken by a <RETURN> character or spaces
.EMT	-	beginning of a procedure line, broken by a <RETURN> character or spaces

Quite an impressive list!

You can see from the above list that Logo3 has a lot more scope, particularly for handling lists, that the earlier Logo2. The graphics and sound commands added for the Amstrad version of the language haven't been sacrificed, either.

The other major difference between the two systems is the amount of memory space available under Logo3. Under the earlier version of Logo you had 2105 nodes (10525 bytes) available as workspace for Logo procedures. Logo3 nearly doubles this to 3761 nodes (18805 bytes). This allows much longer programs to be developed.

The two programs discussed in the earlier sections of this chapter will run under the full Logo, but a couple of modifications need to be made. To start with, though, you'll need to prepare a working disc for Logo3, as the CP/M Plus version is called. Do this using Disckit3, and then type:

```
SUBMIT LOGO3
```

rather than the normal CP/M technique of typing the program name alone. There is a SUB file already included on the disc, and this loads a set of customised key definitions from the file KEYS.DRL. These definitions set up the cursor keys, and some others, to speed editing.

The procedure 'picture' will run as before. Type in the procedures 'picture', 'move', 'frame', 'tree', 'house' and 'title' as shown and you can run the program by typing:

```
picture
```

The car procedure, though, needs one of its sub-procedures removed. 'thing' is a Logo primitive in Logo3, and the procedure that was defined to mimic it in the CP/M 2.2 Logo is no longer needed. In fact, if you include it, Logo3 will confuse it with the primitive 'thing' and display an error message. To erase the procedure, if you've already entered the program under CP/M 2.2, type:

```
er "thing
```

This will erase the procedure 'thing'. If you are doing this, you'll have to do it under the 2.2 version of Logo, as Logo3 will refuse to load the 'car' file, because of the clash of the 'things'!

Pascal 80

The HP80 compiler works under CP/M Plus with no modifications. The XOR program will compile and run without modification, as will any program which is free of screen control codes. If you do need to manipulate the screen, remember that the CPC 6128 uses a terminal emulator and you will have to use the Zenith Z19/Z29 codes rather than those available under CP/M 2.2. You can find a list of these on page 7.49 of the *User Guide*.

If you're upgrading from a CPC 464/664 to the CPC 6128, and already have a copy of Pascal 80, you may also have a copy of ED80 for editing your source programs. If you bought this before the release of the CPC 6128, you will probably find that ED80 will not run on the newer machine, and won't respond to your attempts to re-install it. You can return the program to HiSoft, together with £5.00, and they will return an updated disc, containing copies of the editor installed for both systems. New copies of Pascal 80 include both versions of ED80.

As with Logo3, Pascal 80 (and indeed all CP/M-based languages) enjoy the extra memory space offered by the CPC 6128 and allow you to develop larger programs.

HiSoft C

HiSoft C as described in this book runs under AMSDOS, and is therefore unaffected by the change of CP/M versions. Just as this book is being completed, however, HiSoft are about to announce a true CP/M version of C with many enhancements. Among these are routines to allow true random-access filing, which makes the language very suitable for writing database-type application programs.

Other Languages

The CPC 6128 is a much more 'standard' CP/M micro than the CPC 464/664, and has a larger TPA than many. Because of this, most language and application software which found life hard on the earlier Amstrad micros are more than happy with the bigger machine. There are plenty of languages available under CP/M, and a few that you might care to consider are Microsoft BASIC (for which you can buy a compiler), Turbo Pascal, which has become something of an industry 'standard' in the US, BDS C, which is a full CP/M version of the language, and BBC BASIC, which offers several useful extra structures.

The three languages described in this chapter open many possibilities for the adventurous programmer. BASIC is an extremely comfortable language, which is easy to use and, sometimes, easy to read. It's main disadvantage, though, is it's speed of execution. For many applications an interpreted language like BASIC is simply not fast enough. There are such things as BASIC compilers, but because of the way the language is structured (or unstructured), these don't usually give the same kind of speed increase as can be gained by switching to a language such as Pascal or C.

Whether you want to learn another language is you own decision, but you should now be aware that there are other wonders to be explored, as long as you're prepared to plan the expedition, and reconnoitre the land well ahead of you.

If you're more interested in the finished product, however, and particularly concerned with using your disc-based Amstrad micro in your business, then read on. The next chapter discusses the main business applications, with examples of each of the 'Big Three' programs.

Chapter 6

Business Software

Since the advent of CP/M on microcomputers, the largest single area of application has been in business. Many programs have been written to exploit the speed and convenience of micros, and although there are plenty of specialist programs designed to fill almost any business need, there are three main applications that have proved the most popular: the word processor, the spreadsheet and the database.

There are examples of each of these types of program available on the Amstrad micros and in this chapter each application will be described with the aid of a two programs available for the Amstrad machines.

Although CP/M is well implemented on the Amstrad micros, both the CPC464 and CPC664 suffer from the small size of memory available to application programs (the *transient program area*, as it's called); in the region of 39.5K. This leaves some CP/M software struggling, as it's used to more room. Some programs are available in special Amstrad versions, while others have been specially written for CP/M within the Amstrad machine.

Three specially written packages will be looked at in this chapter, the word processor Tasword 464D from Tasman Software, the spreadsheet Microspread from Saxon Computing, and Masterfile 464, the database from Campbell Software Design. Despite their names, all the programs will work on the CPC664 and CPC 6128 as well. All three programs are conveniently available, and perform the majority of functions you would expect to find in programs of their type. There are plenty of others, though, and no particular recommendation is implied by the inclusion of these three in this discussion. The 6128 Special Section at the end of this chapter looks at three 'genuine' CP/M programs, which will run happily in the larger memory of this machine.

THE WORD PROCESSOR

One of the most vaunted applications for microcomputers is the word processor. Much more than a computerised typewriter, the word processor can change your methods of writing, can speed your typing (even if you're a one-fingered typist) and can organise your correspondence to save time when producing standard or similar letters.

The fundamental difference between a word processor and any type of mechanical or electro-mechanical typewriter is that the word processor separates typing from printing on paper. As soon as these two stages in producing a document are detached from each other, several changes follow automatically.

The onus on you to produce error-free text is immediately removed. Now the document may be typed as a free translation of your thoughts, without the worry of having to correct mistakes with correction fluid, or perhaps retype a 'final' copy later on. If, having completed the document, you find a number of typing errors, these may be corrected long before the text ever reaches the paper. You are only changing an electronic copy of the document, which has little physical being at all.

Not only can you correct typographical errors, but phrases or sentences can be restructured just as easily by moving through the text and altering the offending sections. Characters may be inserted, deleted or overtyped with as little effort as adding new text to the end of the document.

The documents created with a word processor no longer have to be stored as paper copies in folders or filing cabinets. Although you can produce a paper copy at any stage, the finished text can be stored in magnetic form on the surface of a disc. Many documents can be stored on each disc, dramatically reducing the overall storage space.

There is an additional benefit for those who have never managed to bring more than two fingers into play on the keyboard at once (among whose numbers I include myself). The touch of a modern computer keyboard (like those on the CPC Amstrad machines) is so light that you can expect to type a lot faster than with a manual or electric typewriter.

If it were not for these benefits, I for one would never attempt to type my own manuscript for a book of this length.

There are a number of other features of word processors which should endear them to anyone who spends much of their time typing. These are concerned with the format of the typed document on the page. Most good word processors will have the features as a matter of course. The terminology is fairly straightforward, but it's as well to run through these terms before starting a specific description of Tasword 464D.

When you have typed a document, you may find that some of the things you've written seem to be in the wrong order. Perhaps a paragraph should be repositioned or deleted altogether. A word processor allows for this kind of movement, for copying and for deleting blocks of text.

The way these blocks are handled is by 'marking' the start and finish of the section, usually by typing some special key sequence, and then using other sequences to move, copy or delete the text. In the case of copying text, the

original marked block is left unaltered.

One of the main reasons you have to watch the carriage while typing on a conventional typewriter, is so you don't type off the right-hand side of the paper. On a word processor, you don't need to worry about this, as a word that would run off the end of one line is automatically moved to the start of the next. This is known as *word-wrap*, and means that you only need to type <ENTER> when you want to stop a line short, as, for instance, when you complete a paragraph.

A normal business letter is typed so that the right-hand edges of lines are left uneven, and only the left-hand edges line up down the page. The right-hand edges are said to be *unjustified.* The pages of this book, however, are lined up on both margins, and are justified. Most word processors will carry out this justification automatically, if you select the appropriate option.

You can also choose to centre a line on the page. You don't have to count the number of characters in the line and range it in yourself. The word processor will do it all for you.

Sometimes you will want the line spacing set to something other than one line. The word processor, like the typewriter, allows you several spacings, but has the advantage that these spacings only need to show on the printed document. While working with it on-screen, you can see the text single-spaced, which means more will fit on the screen.

A lot of business documents need a heading at the top of each page and often chapter or page details at the bottom. These can again be handled automatically by a word processor.

They are normally referred to as *headers* and *footers.* You can enter whatever material you like into each section, and the word processor will print it at the top and bottom of each page. You can also adjust the amount of space at top and bottom of the page, and where within these spaces the header and footer will be printed. Most word processors also update the page number, so it's incremented on each fresh page.

If you're writing a long document, and towards the end of it find you have consistently misspelt a word, it can take a long while to check through the text by eye, correcting each misspelling. Again, most word processors can help you with this, via their *'search and replace'* functions. As the name suggests, these routines will search through your text for a given word, and replace it with another.

Most programs will offer you the chance of a *global* or *selective* replacement. A global replacement automatically changes all occurrences of your selected word, whereas a selective replacement will show you each occurrence in context, and ask for confirmation before replacing it.

A search and replace routine will actually take the string of characters you enter, rather than treating it as a discrete word. This means that if you're trying to change 'import' for 'export' throughout your text, the word processor will also change 'importing' for 'exporting' and 'importation' for 'exportation'. This may be OK, but you'll have to check the routine doesn't make any 'exportant' mistakes.

As well as correcting spelling mistakes, the search and replace routine can also be used to expand on a deliberate abbreviation within the text. For example, if you know that the word 'computer' will appear frequently in your document, you can shorten it to 'c' while typing your text, and then expand 'c' to 'computer' in one operation at the end. You'll have to be sure the language 'C' doesn't appear in the same document, though!

Tasword 464D

Tasword 464D offers all the features just mentioned, and saves and loads your documents from disc. Tasword runs within AMSDOS, rather than CP/M, but from your point of view as a user, there will be little apparent difference.

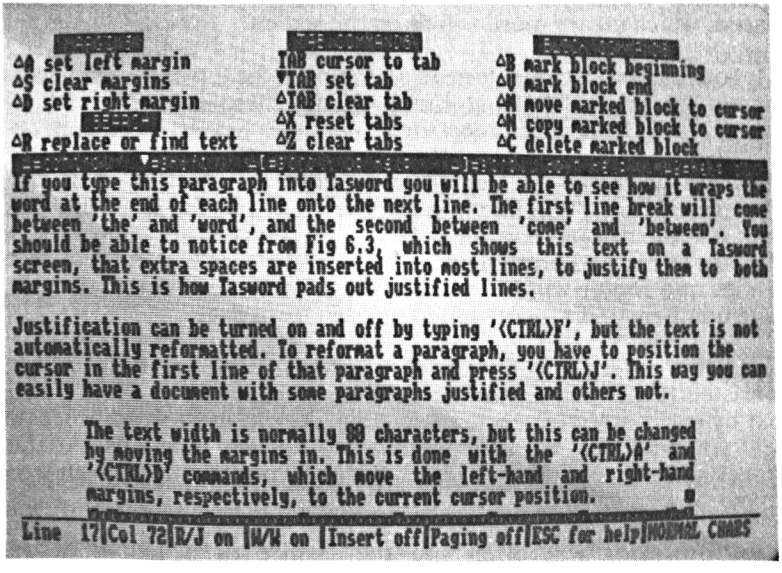


Figure 6.1 Tasword 464D screen

When you load the program, the first screen you'll see is the text screen. This is where you enter all your text, and from where you control most of the facilities of the program. It is shown in Figure 6.1, complete with a couple of paragraphs of sample text.

As you can see, the screen is divided into three parts. The top section offers some guidance to Tasword's controls, which are all <SHIFT> or <CTRL> sequences. There are actually a lot more controls than those shown here, and it is possible to display other sets of instructions in this top section, or to display them all together, covering the whole screen. The full set is shown in Figure 6.2. If you want to see more of your document in one go, you can also remove the normal top section completely.

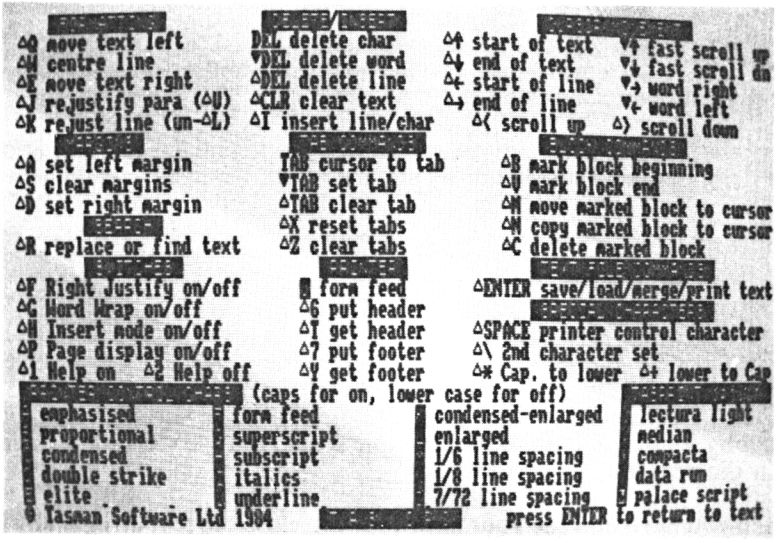


Figure 6.2 Tasword 464D 'help' screen

The centre section, which is initially blank, is where you enter your text. It acts as a *window* on your document, and may be moved up and down through the text.

The bottom section provides a *ruler*, showing your current margin and tabulation settings, and several pieces of *status* information. This information includes whether your text is being justified, whether the breaks

between pages are shown on-screen, and the current position of the cursor, as row and column numbers.

If you type this paragraph (Figure 6.1) into Tasword you will be able to see how it wraps the word at the end of each line onto the next line. The first line break will come between *the* and *word*, and the second between *come* and *between*. You should also be able to see that extra spaces have been inserted into most lines to justify them to both margins.

Justification can be turned on and off by typing <CTRL>F, but the text is not automatically reformatted. To reformat a paragraph, you have to position the cursor in the first line of that paragraph and press <CTRL>J. This way you can easily have a document with some paragraphs justified and others not.

The text width is normally 80 characters, but this can be changed by moving the margins in. This is done with the <CTRL>A and <CTRL>D commands, which move the left-hand and right-hand margins, respectively, to the current cursor position.

Once the text has been entered, you may want to make changes. Insertions and deletions are simply made by moving the cursor to the right point in the text and pressing <DEL> or <CLR> to delete left or right, or <CTRL>I to start an insertion.

The way Tasword handles insertions is rather unusual. When you press <CTRL>I, a blank space is inserted into the text. If you want to add a single character, then you can type it immediately, but if you want to insert several words, you press <CTRL>I again. Tasword will then move all the text that is to the right of the cursor on the current line, onto the next line. You can then enter your new text up to the end of the line. If the text you're inserting is still longer than the available space, you use <CTRL>I again to clear the next line.

When you have finished your insertion, you close the text up again by using <CTRL>J, to reformat the paragraph. (Most word processors use a simpler method for inserting text, whereby each character typed moves all the following text to the right).

On completing your document, you may want to do two things: print it and save it on disc. These functions, and a number of others, are selected from Tasword's main menu. Before you switch to the menu, though, you may want to title and number the pages of your document.

To enter a header into Tasword, you simply type the heading you want into the first line of your document, and then type <CTRL>6, to tell the program that the line is a header. The line will be taken into an area of memory assigned to headers, and will disappear from the text on-screen. It can be called back using a different <CTRL> sequence, and edited at any

time.

A footer is set up in exactly the same way, except that you use <CTRL>7 to indentify it. It may be recalled and edited like a header.

To display Tasword's main menu, you type <CTRL><ENTER>. The menu is as in Figure 6.3. You print your document using the P option from this menu.

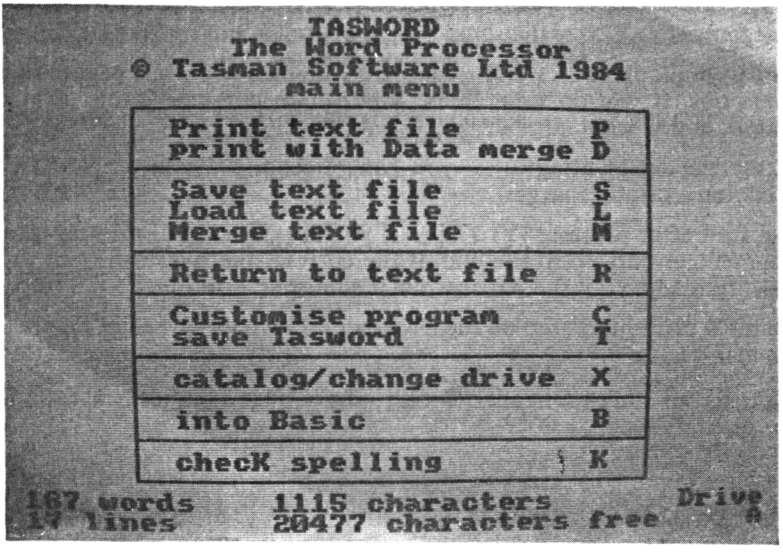


Figure 6.3 Taswords main menu

When you select P, you are presented with another menu, offering a number of different print options. These options define the final layout of the text on the page. The numbers and letters in brackets at the end of each option are the default values that will be used if you don't change them. Running through them briefly:

start at line (1) you can choose to start the printout from any point in the text — useful for printing subsections of documents.

finish at line (last) you can also finish the printout at any point you specify.

number of copies (1) normally you will only want one copy of your text, but you can specify any number and Tasword will print them out in one go.

line spacing (1) you can specify line spacing as any whole number.

continuous or single sheet (C) C/S if printing on continuous stationary (roll or *fanfold* paper), Tasword won't stop between pages. If you're using single sheets, though, you will need the printer to stop, so you can insert a fresh sheet.

form feed at page breaks (N) Y/N again, if you're using single sheets of paper, it saves time if you make Tasword feed each sheet out of the printer, once it's printed it.

print header (Y) Y/N just because you've defined a header, it doesn't mean you have to use it. This option selects whether or not to print it.

print footer (Y) Y/N the same option as above, but for footers.

print page numbers (Y) Y/N some documents will need page numbers, others not. If you do want them, you can specify where they should be printed, and at what number they should start.

left margin on printing(0) you can offset the whole of your text by specifying a printer margin.

form feed after printing (Y) Y/N if you are thinking of printing another document to butt up to the end of the current one, you will want to stop Tasword feeding out the paper when it's finished printing.

A sample printout, based on the document shown on-screen in Figure 6.1 is shown in Figure 6.4.

The printout shows a header, footer, and a page number centered at the bottom of the page.

To save a text file to disc, you select the **S** option from the main menu. The disc directory is then displayed, and you can type in a filename and save your text.

You may be interested in the other options from the main menu. These are:

Merge text file this allows you to add a text file from disc to one already held in memory, presuming that there is room in memory to hold both together.

Return to text file returns to the edit screen and displays your document.

Customise program allows you to specify several display options for the program. You can, for instance, set the display colours and how the cursor appears. You can also set up control sequences to be sent to your printer for special effects, such as underlining and **'bold'** print.

save Tasword will save a 'customised' version of the program to another disc. You should always work with a copy of you programs, rather than the master.

catalog/change drive shows what's on your disc, or changes from one to the other on a twin drive system.

A Sample Document prepared on Tasword 464D Date:8.7.85

If you type this paragraph into Tasword you will be able to see how it wraps the word at the end of each line onto the next line. The first line break will come between 'the' and 'word', and the second between 'come' and 'between'. You should be able to notice from Fig 6.3, which shows this text on a Tasword screen, that extra spaces are inserted into most lines, to justify them to both margins. This is how Tasword pads out justified lines.

Justification can be turned on and off by typing '<CTRL>F', but the text is not automatically reformatted. To reformat a paragraph, you have to position the cursor in the first line of that paragraph and press '<CTRL>J'. This way you can easily have a document with some paragraphs justified and others not.

The text width is normally 80 characters, but this can be changed by moving the margins in. This is done with the '<CTRL>A' and '<CTRL>D' commands, which move the left-hand and right-hand margins, respectively, to the current cursor position.

Chapter 6 The Amstrad Disc Companion

- 1 -

Figure 6.4 - Printed document from Figure 6.1

into Basic leaves Tasword and returns you to Locomotive BASIC.
checK spelling Tasman software, the publishers of Tasword, also produce a spelling checker. This program, which carries its own dictionary on disc, will run through your text, reporting each spelling it can't trace. If you're a bad speller, this utility can save you a lot of time.

The observant reader will have noticed that there's one option not covered on the above list. That is `print with Data merge`. The facility to merge data from other files with a text document is offered by a number of good wordprocessors, and can be very useful.

Personalised business letters

A good example of the way the feature works, is in producing 'personalised' business letters, using a list of names and addresses held as a separate text file, or on a database. A typical database, *Masterfile 464* will be described in the next section of this chapter, and this program is quite suitable for use in conjunction with Tasword to produce this kind of letter.

Assume for the moment, though, that you have a list of names and addresses set up as a separate Tasword text file. To merge these names and addresses into a series of letters produced on Tasword, you could set up your letter as in Figure 6.5.

```
&N
&A

                                        8.7.85

Dear &N,

This is just a short note to illustrate how a series
of personalised letters can be produced on
Tasword, buy merging one text file into another.

Yours sincerely,
```

Fiugure 6.5 Personalised business letter.

The text of the letter will be printed out as normal, but each time Tasword comes across the **&** character, it 'pulls in' the corresponding section of the name and address file, and prints it out in the appropriate place in the letter.

An example of a suitable name and address — part of the name and address file — would be:

```
&NA.M. Strad
&A16, Valley Road,
Bletchley,
Bucks.
```

The single character after the & symbol is used to identify the right part of the record to be inserted into the text. You can use any letter of the alphabet, in upper or lower case, giving you up to 52 possible insertions in any document.

If you print a number of letters using this technique, then the first name and address will be used in the first letter, the second in the second letter, and so on. You can also arrange to display each letter, including its insertions, on-screen before printing it. If you choose not to print, the program will automatically skip to the next name and address. In this way you can select which names on your list to send letters to.

A more sophisticated method of selection, also provided by Tasword, is to print documents subject to pre-set conditions. For instance, you can choose to merge files and only print out letters for those people whose names begin with 'A', or who live in Devon.

A word processor is probably the first program you should think of buying if you want to make use of your Amstrad micro in business. Of all the business applications, word processing is the easiest to introduce, and will show the most immediate benefits.

While you can store your name and address list as a separate Tasword text file, you may find it a lot easier to use a purpose-made filing program, *a database*, for this kind of work. The next section will look at the uses of a database, and describe one such program for the Amstrad CPC464/664 micros.

THE DATABASE

If you run your own business, are secretary of a local club, or even just own a lot of gramophone records, the chances are you'll keep some kind of file. This may be a notebook with all the details of your records, a card index with a card for each member of your club, or a filing cabinet containing details of the products you sell or the people you employ.

As an example, say you are a electrical dealer, and you keep a card index of all your customers, detailing their names and addresses, what they've bought, when they bought it, and when the guarantees on their equipment run out. It's your policy to write to customers a couple of weeks before their guarantees expire, to offer them maintainance contracts. The only way you can do this, though, is to go through the cards selecting by hand the relevant customers for the coming month.

This, on its own, is probably not too laborious — you only have to check the cards once a month. You're also having a sale soon, however, and it would make sense to advertise this to your past customers — but only those who live fairly close to your shop. And next month, you're thinking of running a special offer on cassette tapes. How about sending literature to all those who've bought cassette recorders from you in the past? There are plenty of good ways to use the information on your cards, but each application means you have to thumb through them all by hand.

A computer database can hold the information on your index cards in a very similar format to the card itself, but with all the advantages of computer data storage.

The three main terms used in describing a database reflect the similarity between the physical filing system and the computerised equivalent.

A *record* holds the information that would be held on a single card in a card index system. All the data referring to one person in a personnel system or one product in a stock control system would appear on a single computer record.

Each piece of information; a name, address, part number, book title or whatever, will be stored and displayed in a small section of a record. Each of these small sections is known as a *field*. Thus, on a Name and Address record you might have the following fields:

Title (Mr/Mrs/Miss/Ms)
Initials
Surname
Position
Organisation
House
Street
Town/City
County
Post Code
Telephone No.

Each of these items would have its own field on a record.

All the records together make up a *file*. You've already come across this term when first formatting a disc in Chapter 1, and have used it a lot since then; and a data file is simply all the records of data used by your database.

There is nothing to stop you storing several different files of data on your discs, all of which may be used with the same database program, but covering different applications. It is only the data you save, and the way in which you configure the database program, which distinguishes between one data file and another.

One of the main advantages of a computerised database is the speed and sophistication of searching the data. You can search through a computerised database and find not only those customers who bought goods from you twelve months ago, but also all those who live in your town, bought cassette recorders, paid cash and have maintainance contracts.

You can search through a database simply by stating the rule, or rules, you want the records to match. The program does all the work and displays or prints a list of the records it finds.

If you hold your information on a computer database, you will probably want to hold it in some particular order. In the example just quoted, it would be useful to have all your 'cards' in alphabetical order of customer name. If you are trying to keep a tally of the number of cookers or TVs you've sold, though, you might equally well want to categorise your sales by product type. There's no reason why you can't do both. You can sort your database by product type and then, within each type, you can sort by customer name.

Besides holding the information on disc, you will need to print out reports, like the list of 'all your customers who own cassette recorders'. Most computer databases will be only too happy to oblige, and will offer a number of different formats for the reports and lists they produce.

As well as all these advantages of a computer database over a manual card index, some database programs allow records to be 'related' together, so that you can associate a piece of data from one record with a different piece from another. In use, this means that you can set up a record which has a link with another, and which can refer you to the second record for further information. The relationship is often referred to as a *parent-child* link.

As an example, if you were using a database to catalogue books in a reference library, you would most probably want to index the books by both author and subject. While you could do this by printing lists sorted by these two catagories, a better solution would be to have two related files; an author file and book file.

You could then set up a link between each author record and the corresponding book records so that when you look up an author, all his/her books are referenced with their details, taken from the book record file.

The main advantage here is a saving of space within the database. The link between the two files can be a series of one or two-letter codes, taking up little extra space but appearing to the user to provide a lot of extra information.

There are two main ways of storing the data which a database program manipulates. You can either load the whole file into memory from disc (or tape) at the start of each session, or load and save records on disc as you come to use them within the database.

The memory-based database program has the advantage of speed. Everything you do on your file deals with data in memory, and the program will probably not have to wait for a disc access until you store your file back to disc at the end of the session.

The database that holds all its records on disc is limited only by the size of the disc, in Amstrad's case about 170K. The problem here is, of course, that each time you want to manipulate your records, the database program has to shuffle them about on the disc. Each disc access takes an appreciable time, meaning that the whole system will appear a lot slower than the database working in RAM.

A more extensive discussion of the different ways of storing data is given in chapter 7.

Database programs vary rather more than word processing ones in the way they set about their jobs. All of them consist of two parts, sometimes handled by two distinct programs. One part will help you to set up your database, defining how the information will be presented on the screen and how the records will be set up within the file. The other part handles entering and amending your data, and displaying and printing it out.

You will normally only need to set up your database once. Although you may need to modify a screen or print layout later on, you should plan to get as much right as possible from the start. A lot of database programs make it awkward for you to alter the structure of your file, once you've started to enter your data into it.

The best way to start is with pencil and paper. Plan out your screen and printer layouts and which pieces of data you want to store. Make sure the database program you choose can handle the amount of data you have, and that it can provide you with all the reports and lists you need.

Perhaps the best way of illustrating how a database may be set up, filled with data, and made to organise the information is to discuss the workings of a specific program. As with Tasword 464, no particular recommendation is given to Masterfile, other than the fact that it's already available, and is a good example of how a microcomputer database.

Masterfile 464

This program, written for the CPC464, works equally well on the 664 and is a sophisticated, memory-based, relational database. A separate version, Masterfile 128, is available for the CPC 6128. It will cope with just about anything you care to do with your data, and is limited only by the available memory of the CPC464/664, which leaves the program with about 32.5K for data storage.

The program is easy to control, using a series of menus from which you select an option by pressing a single key. At first sight, Masterfile appears rather daunting. There are an awful lot of menus, and many options on each of these, but with a little perseverance you should find the organisation of the program is actually very straightforward. Although a complex program, each stage of creating and using your database file leads logically on from the last.

When you load Masterfile, you are confronted with its main menu, which is illustraded in Figure 6.6.

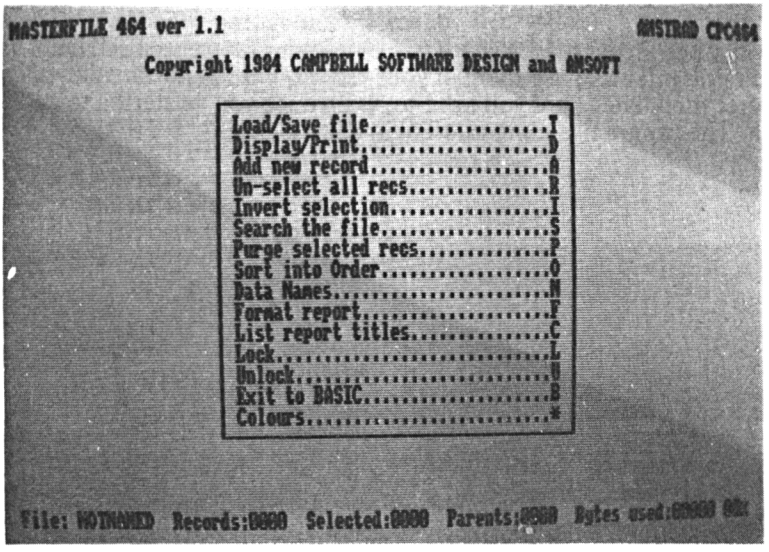


Figure 6.5 Masterfile main menu

Beneath the 15 options of the main menu are five pieces of information about the data file itself. These are the filename (which is initially `NOTNAMED`, to remind you to give it a name), the number of records you've entered, the number currently selected, the number of 'parent' records on the file and the amount of memory used, in bytes and as a percentage of the total.

The selection of records is connected with Masterfile's search facility, which will be discussed a bit later, as will the idea of `parent` and `child`

records, which form the basis of the program's relational links. The byte count is useful to keep an eye on how much room you have left in memory. This line of statistics is maintained all the time you're using Masterfile.

The menu is arranged in the right order for use once you've defined your file format. Before you can use the program for data storage you will at least need to define the names of the fields on your records. This is done by selecting option **N** from the main menu.

Each field in your record has two pieces of data attached to it, a *data reference*, which is a single letter or number, and a *field name*. The field name is used as a prompt when you enter your data, and the data reference is used by the program to select a given field — it is also a lot easier to type **P** than **Part No**, each time you want to select the 'part number' field.

It will make life easier for you if you enter your references and names in the order you intend to use them in your record, but Masterfile does allow you to alter them later on if you have to, even once you've started to enter your data. The program is quite unusual in allowing this.

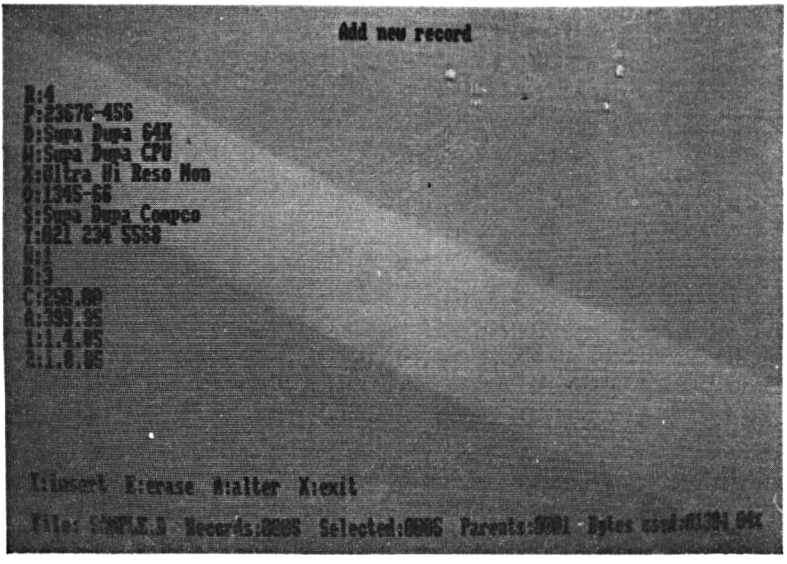


Figure 6.7 - Raw format of data entry in Masterfile

Once you've entered all the references and names, you can return to the main menu and, if you wish, start entering your data. Masterfile doesn't require you to define your record format (the way your data will be shown on the screen) before entering data, although you'll only be able to see the data in its 'raw' form, with each piece lined up against the field name and data reference. Figure 6.7 shows a typical screen in this raw format.

To design the record format, you select option F from the main menu, and you're immediately presented with another menu, giving you options to create a new format, or to review, erase or copy an existing one.

If you select the 'New format' option, the first of the screen design menus is displayed. This one deals with the overall size and shape of the record, and is referred to as the 'Report geometry' menu. The options shown here are as follows. The figures initially displayed are the default values, which will be used if you don't overwrite them:

```
Heading area depth.......01
Record area depth........02
Forms depth.............. 66
Forms margin............. 00
Border colour............ 23
Pen colour............... 00
Paper colour............. 23
Title.................. (title)
```

A further, short-form, menu appears at the bottom of the screen. This single-line menu is used on all the record design screens:

```
A:alter E:erase I:insert item N:next item D:display X:exit
```

The options here are fairly self-evident:

A:alter is used to change any of the parameters in the current menu,

E:erase will wipe out the current record format-item (a heading, data entry or drawn line),

I:insert item allows you to insert a new format-item,

N:next item displays the next format-item in sequence,

D:display shows the current record on the screen so you can see how it looks at any stage,

E:exit takes you back to the main menu.

The options on the geometry menu cover the margin and length of the printed report, the colours of the screen, and the size of the heading and data areas on the display.

The heading area is where you put the title of the file (this is automatically centred by Masterfile in the first line of the record), and any column titles you might want to use in the record.

The data area is where the information from the record will be displayed, but you may also include headings for the data fields here.

To alter any of the displayed default values you first press **A**, and a small diamond-shaped cursor appears to the left of the first line of the current menu. A prompt, **Enter new data if required:** appears at the bottom of the screen, and you can overtype the default value on the current line of the format menu. Pressing <ENTER> will move the cursor to the next line.

After the geometry menu, you can start to enter the headings, data fields and ruled lines which make up the format of your record. You do this by selecting **I** from the single-line menu, to insert a format-item. Another single-line menu at the bottom of the screen then allows you to select between headings, data and drawn lines.

The heading format menu looks like this:

```
Hdg(0) or Rec(1).....00
Column.....01
Line.....01
Inverse.....N
Text.....(heading)
```

You can put a heading in either the heading or data file areas of the record. The first option on this menu selects which. The column and line numbers dictate where the heading will be placed on the record screen. The record may use the full 80 columns of the screen, but only 21 lines of the 25 available on the Amstrad. The text of the heading will be reproduced exactly as it's typed in, including leading and trailing spaces. You may select to display the heading in inverse video if you want.

The data field format menu looks like this:

```
Data reference.....?:Un-named data...
Hdg(0) or Rec(1) ..... 01
Column ..... 01
Line ..... 01
Width ..... 40
Depth ..... 01
Inverse ..... N
Right-justify ..... N
Numeric ..... N
Column total ..... N
Two dec.places ..... N
Thousands commas ..... N
Filler if absent ..... -
Leading symbol .....
```

When you type the data reference letter, Masterfile displays the letter and associated data name on the top line of the menu.

Again you can display the field in either the heading or data areas of the screen, although it would normally be put in the data area.

You can position the field by defining the column and line numbers, but in addition you specify the length and depth of the field. The data field can be displayed in inverse video, and may also be right-justified, which is useful for lining up columns of figures (amounts of money, for instance).

The field may hold either character or numeric data. If it's numeric, you may also specify commas every third figure (to mark out thousands, millions, etc), decimal place fixed to two figures, and a leading symbol, such as '£' or '$'.

You can define the symbol that will be used to fill the field if the data is absent on any record. Lastly, you can define a data field to be the sum of a column of other fields. The calculation will be done automatically by Masterfile.

The last format-item menu is for creating drawn lines on your record. These can be used to divide up and frame different parts of the screen, to highlight important fields or notes within the file.

The menu looks like this:

```
Across (Y) or Down (N).....Y
Mid-char ends.............N
Double thickness..........N
Start column..............01
Start line................02
Length.....................79
How many...................02
Interval...................20
```

By defining whether the line should be drawn across or down, where it should start, what it's length should be, how many should be drawn, and at what interval, you can set up a wide variety of frames and dividers very quickly and easily. You can emphasise lines by printing them double thickness, and run them to the centre of character positions, to ensure that vertical and horizontal lines meet.

At any stage during the design of your record, you can see what it looks like by switching to Masterfile's display mode. You can then switch straight back to the format menu to refine or add to your record format.

An example of the kind of record that may be designed with Masterfile is shown in Fig 6.8

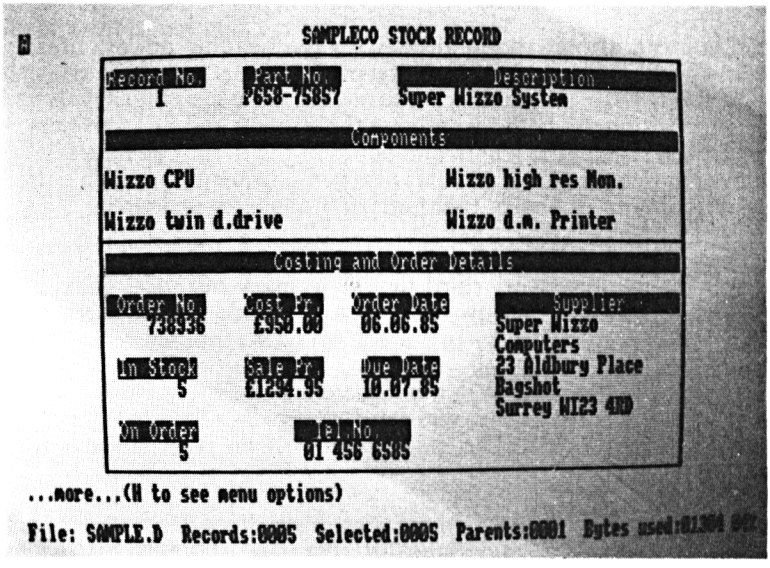


Figure 6.8 Sample record screen designed with Masterfile

This screen shows a simple stock record that might be useful to to a computer dealer. It shows the current stock level of each product stocked, as well as any outstanding orders, the cost and sale prices of the item and the address and telephone number of its supplier.

In addition, the record shows what components, if any, the product contains. This would allow for 'packages' assembled by the dealer, containing a number of different products.

The format shown in Fig6.8 is only one of many you can design, all of which can be used on the same file. You can design one format for a detailed record, and another which displays a limited amount of information, but on several records at once. You can also specify different formats for screen and print-out, so you can get the most from your printer.

Masterfile offers two types of record, the parent and child, which may be linked together to form a relational file. A parent record may be linked to a child record, so that the data on the child record is automatically called up when the parent record is displayed. A record may even have a link up to a parent and another down to its own child. In Masterfile, a parent record may have more than one child, but a child may have only one parent. This won't usually be a limitation.

The way these links are created in Masterfile is by using two special data references, '>' and '^'. The '>' reference is used to reference a field in the parent record, while '^' is used in the child record to tie up the other end of the link. You can then give the field a suitable name to show it holds the link to another record. What you have, in effect, is the ability to build two files in one, each of which has limited access to the other data.

To sort your Masterfile file, you select option 'O' from the main menu, and are then asked for the data reference of one of the fields in your record. This field will be the 'key' for the sort. On the sample file from which the record in Fig 6.8 comes, you would probably want to sort on part number or description. The two options Masterfile gives you for sorting are an ascending or descending and a character or numeric sort.

An ascending sort will put records into alphabetical order from 0 to 9 and from A to Z, with the numbers first. A descending sort will do the opposite; Z to A and 9 to 0, with letters first.

In most cases a character sort, which works using the ASCII code values for each letter and number, will give the right result, even when sorting numbers. A problem arises, however, when sorting numbers with a different number of digits in them.

If you do an ascending character sort on records with, for example, the numbers 1000 and 30 on two separate records, a character sort will put 1000 first. this is because a sort routine based on ASCII characters will compare each character pair in turn and stop when it finds one character code greater than the other. When it checks the '1' in 1000 against the '3' in 30, it will know the '1' is smaller, and not check any further.

To overcome this problem, a numeric sort treats all numbers as their actual values, and will sort 30 and 1000 into their proper order.

Masterfile marks each record on your file as 'selected' or 'unselected'. When you first set up your file, all the records will be selected, but if you search through the file for specific records, then the ones you don't want will be unselected by the program.

This technique is very versatile, because you can reselect other records according to another rule you set, or invert the selection, to see only the records which didn't match your original rule. You can superimpose one rule on the selected records left after your first search, and keep doing this to refine your search so that the selected records eventually match a complex set of rules.

When you select the search option, 'S', from the main menu, you are offered the option to select or unselect records. This means you can either extract records from those already selected, or include further records from the set currently unselected. Once you've decided which to do, a further set of

options are offered:

```
Parents.....P
Children.....C
Orphans.....O
Childless Parents.....B
Data compare.....D
Prev menu.....X
```

The first four options refer to the relational records described earlier. They give you the options to select all the parent records, all the child records, only the child records with no parents, or only the parent records without children. The orphans option is mainly for debugging a database, as you won't normally have child records without linked parent records.

The data compare option first asks you to declare the data reference of the field you want to compare, and then offers a further menu, to allow you to specify direct comparisons in your search:

```
Equal.....E
Less.....L
Greater.....G
Unequal.....U
Present.....P
Absent.....A
Non-numeric.....N
Scan.....S
Prev menu.....X
```

The first four options are fairly obvious, although it's worth mentioning that these comparisons can be used on character fields as well as numeric ones. The field will be compared against the search string, refered to by Masterfile as the 'argument' of the search, and will be found equal if it contains the same characters. Even if the contents of the field is longer than the search string, the two will be found equal if all the characters of the search string are matched.

The present and absent options will search for any contents in the named field of the records, and select accordinglyu. The non-numeric option is used to check that a field named as 'numeric' really only contains numbers.

The scan comparison is a verstatile option, as it will find any occurence of the search string within the named field. Thus, if you were running a scan search for 'on' in your file, Masterfile would find 'ONUS', 'visions', one two three' and 'NIGHT ON A BARE MOUNTAIN'.

Masterfile is a comprehensive database which will provide most of the facilities you are likely to need. As said before, it's only real limitation is the size of file you can hold in the memory of your CPC464. As a guide, records

similar to that shown in Fig 6.8 take less than 250 bytes each, and so you could expect to fit about 120 into your file. Obviously, the bigger the record and the more information held on each, the fewer of them you will be able to fit into memory.

THE SPREADSHEET

The last of the 'big three' business application programs is the spreadsheet processor. A spreadsheet is very like a large electronic sheet of squared paper. With a sheet of paper, you would normally enter a single character in each square, but with a spreadsheet, you can enter a complete number or work.

You can arrange the words and numbers as you like within squares, or 'cells' of the spreadsheet, so building up a forecast, a balance sheet or any number of other 'models'. The sheet can be a lot larger than your computer's screen, and the display acts as a 'window' onto the full sheet.

Spreasheets consisting of several thousand cells are not uncommon. Most models, though, will only need a fraction of this number.

Although it's very useful to be able to set up a model, laid out as you need it on the computer screen, the real power of the spreadsheet is its ability to hold relationships between the numbers in different cells. These relationships are set up as formulae, which reference the cells co-ordinates. If you look at Figure 6.9, you will see the top left hand corner of a fairly typical spreadsheet display.

```
  <  A  ><  B  ><  C  ><  D  ><  E  ><  F  ><  G  ><  H  >
 1                      SPREADSHEET EXAMPLE
 2
 3                 100
 4                                24                [     ]
 5
 6                           3
 7
 8                                          800
 9
10            8                                          100
```

Figure 6.9 - Section of a typical computer spreadsheet

As you can see, the top row of the display is divided up into column headings, which are labelled from A to H. The rows down the left-hand side are numbered, from 1 to 10. Each cell is defined as the area of the sheet where a particular row and column meet. Thus the cell marked out with the two square brackets, [and], has the cell reference H4.

The two brackets represent the spreadsheet cursor, which would normally

be shown as a complete cell in *inverse video*, that is with the pen and paper colours reversed. This highlights the cursor's position.

The cursor can be moved around the spreadsheet, usually by using the cursor arrow keys. If the cursor is moved to a row or column at the limit of the display (in the above example, column 'H' or row '10') then any attempt to move the cursor further will 'nudge' the window over the spreadsheet, so that column 'A' or row '1' disappears off the screen and the new area appears. You can continue to move the window over the spreadsheet by repeatedly nudging it around.

The words and numbers shown in Figure 6.9 are the kind of entries you might make on a spreadsheet, although they would usually be arranged in a more logical order, and the values would be given labels to explain their meanings.

To enter a number or word, you simply move the cursor to the cell you want to fill and type the digits or letters which make it up. To type **SPREADSHEET EXAMPLE** in Figure 6.9, for instance, you would move the cursor to cell D1 and type **SPREADS**, move to 'E1' and type **HEET EX'**, and move to F1 and type **AMPLE**. Some spreadsheets allow you to type across cell boundaries, if you're entering text rather than numbers. This makes it a lot easier to type banner headings.

The numbers are entered in a similar way, except that when you finish the entry, by typing <ENTER> or moving to a different cell with one of the cursor keys, the numbers are butted up against the right-hand boundary of the cell. This is known as right-justification, and ensures that columns of figures will line up neatly.

The first three numbers on the example spreadsheet, '100', '24' and '3', are entered directly as numbers, but the next three are derived from these by calculation. The calculations are specified in formulae, which you allocate to the cells in which you want the results to appear. A formula only usually shows on the screen when you move the cursor over the cell which contains it. It is often displayed in a separate area of the screen, reserved for the purpose.

The formulae you use within a spreadsheet may be simple additions of two numbers, or a complex relationship between the contents of several cells and a number of constants. As well as the normal arithmetic operations, many spreadsheets off other financial or statistical funtcions. In the example spradsheet the formula for cell F8 could be **C3*E4/D6**, and that for H10 could be **F8/B10**.

The power of the computer spreadsheet shows when you change one of the numbers in a cell which is tied to another by a formula. In the example spreadsheet, if you changed the value in C3 from **100** to **200**, the corresponding values in F8 and H10 would also change — to **1600** and

200 respectively. Although this example is rather trivial, you should be able to see that with an involved model this automatic recalculation facility can save you a lot of work.

Say, for example, you have built up a spreadsheet containing a list of the products you sell. Against each of them you have entered their cost price, and a selling price calculated as the cost price multiplied by a constant mark-up. If you want to change the selling price by altering the mark-up, then all you need do is overwrite the number in the cell containing the mark-up and all the sale prices will be automatically updated.

You can use this type of readjustment to answer all sorts of questions which begin 'What if...?'. This kind of playing with figures to assess the outcome of a particular change in finance or stock level can be a great help in business decision-making.

When you're setting up a complex model on a spreadsheet, you will often find that the formulae for several of the cells are of identical form, and differ only in the cells that they reference. To have to enter a formula for every cell becomes extremely tedious, and the kind of job you ought to be able to get your micro to do for you — and of course you can.

It's possible to get the spreadsheet program to copy, or 'replicate' a formula, or formulae, from one area of a spreadsheet to another. This saves a lot of typing.

When you're replicating formulae within your spreadsheet, there are two types of replication that you'll need to consider; absolute and relative.

If you replicate a cell reference absolutely, then that cell reference will be used unaltered in all the replicated formulae. So, if the formula for cell G4 is F3 + F4 and you replicate the F3 reference absolutely into cells G5 to G10, each of these cells will contain F3 as the first reference in their formulae.

If, however, you replicate the F4 reference relatively, then instead of replicating F4 into each of the cells G5 to G10, the program will look at the relative position of F4 to G4. In this case, the relative position is one cell to the right, so the new formulae in cells G5 to G10 will each contain the reference of the cell one to the right of the cell holding the formula. Thus, the formula in G5 will be F3+F5, that in G6 will be F3+F6, G7 will contain F3+F7, and so on.

If you find this explanation hard to follow, then you should find the next section, which describes the setting up of a simple forecast on a specific spreadsheet program, will clear up any difficulty.

MICROSPREAD

This is a spreadsheet processor supplied by Amsoft and written by Saxon Computing. It runs under CP/M and works in a similar way to the hypothetical spreadsheet just discribed. It normally assumes a spreadsheet size of 28 rows by 20 columns, giving a total of 560 cells. There's no reason to change this default for the example spreadsheet we'll develop here.

The example will be an income and expenditure forecast for a small business. While the actual model has been greatly simplified, mainly to fit it all in one screen window, it serves to show the essential features of Microspread. A single screen window in Microspread shows eight columns by 15 rows, and when you first load the program from disc, the screen as in Figure 6.10.

The columns are labelled AA, AB, AC, etc, and the rows are numbered 1 – 15. A single message at the bottom of the screen reminds you how to display the *Help* menu, which shows all the control sequences Microspread recognises. Three other reminders show the filename of the current spreadsheet, `FILENAME` until you save the sheet to disc, the display *mode*, which is either `TEXT` or `FORM`, and whether you have `LOCKed` any cells in position.

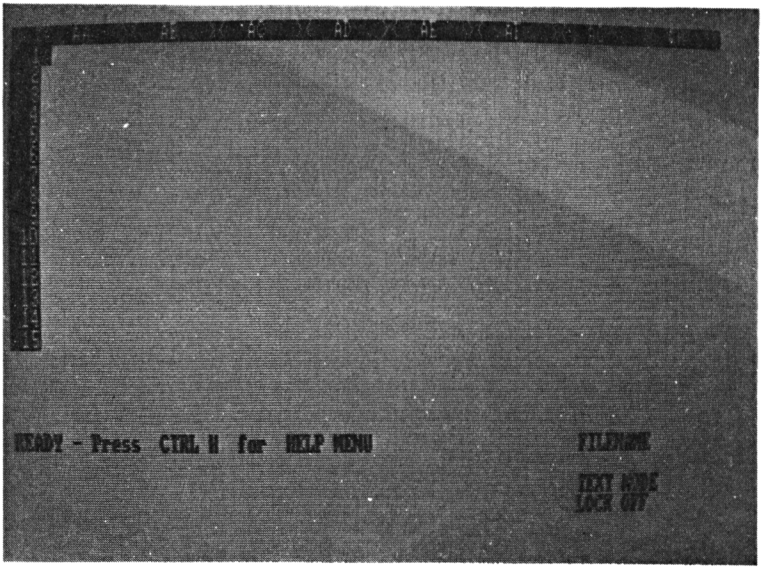


Figure 6.10 Microspread screen

The filename reminder is pretty obvious, but the other two deserve some explanation.

As was said earlier in the chapter, most spreadsheets don't display the formulas assigned to specific cells in the spreadsheet, but show the resulting values instead. Microspread gives you the option of showing either, by switching from text to formula mode. The way this spreadsheet applies formulae to cells is slightly different from other programs of this type, and will be explained a bit later.

The facility to lock particular cells can be useful on large spreadsheets, where you may want to keep certain headings visible in the screen window all the time. For instance, if you have set up a sheet of figures covering a twelve month period, and you have put the column headings 'January' to 'December' at the top of the sheet, then you may find it useful to lock these, so you can scan down the columns without the headings scrolling off the screen.

When you start to move the cursor around a Microspread spreadsheet, you'll notice that it's only one character big and that it only moves by one

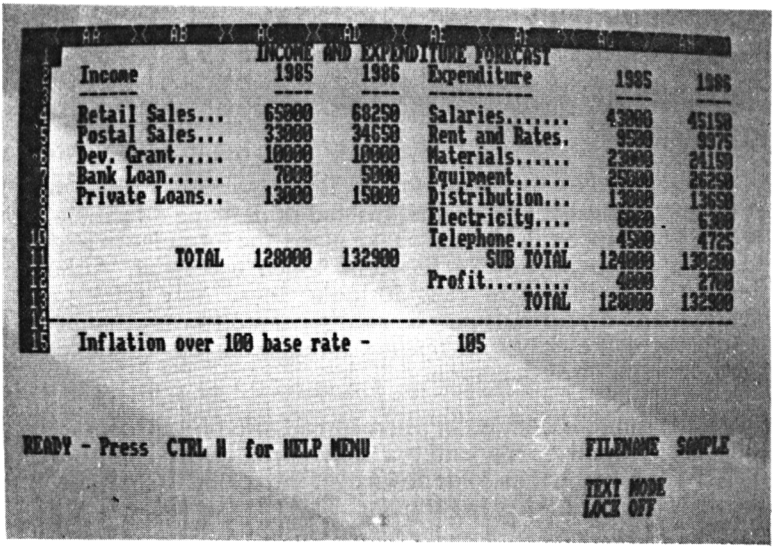


Figure 6.11 Sample forecast screen on Microspread.

character or line each time you press a cursor key. This makes it rather slow to move along a row of your spreadsheet, but fortunately, by pressing the left or right arrows together with <SHIFT>, you can move by a cell at a time as well.

Typing text into Microspread is very straightforward, as the program allows you to cross cell boundaries with long headings, as long as you don't try and put a number into a cell which already contains text. The completed sample forecast sheet is shown in Fig 6.11, and you can see the title and headings which were typed in this way.

Once you've entered all the text, including the year headings (which are actually taken to be numbers by Microspread), you can enter the figures for the 1985 income and expenditure categories. The figures in this example are all fictitious, but are probably not too preposterous!

To produce the income total and expenditure sub-total, you'll need to define the sheet's first formula. A complete list of the formulae used in this example is:

```
FORMULA 1 AC11=AC4 Sum AC10
FORMULA 2 AG13=AC11 , [AA1]
FORMULA 3 AG12=AC11 - AG11
FORMULA 4 AD4=AC4 Rate [AE15]
```

The cell references shown in square brackets are absolute references. All other references are relative.

Microspread handles the definition of formulae rather differently from many other spreadsheet programs. Instead of defining a formula for each cell where you want a calculation to take place, the formulae in Microspread are defined independently of the cells which use them. Although cell references are included in each formula, these only refer to its first application, and the formula may be used repeatedly in any number of cells. Although there are only four distinct formulae in this example, 15 cells derive their values by calculation.

The way a formula is defined in Microspread also differs from some other spreadsheet programs. To enter a new formula, you start by typing <CTRL>F'. A menu then folds down over the screen to show the different operators, available for use in formulae. These are as follows:

```
,ADD            -SUBTRACT       *MULTIPLY
/DIVIDE         $SUM            AANALYSE
CArcCos         SArcSin         TArcTan
XTrend          HHi (max)       LLo (min)
RRange          MMean
%Box1*Box 2/100 PBox 1/Box 2*100
```

The first four need little explanation; the four arithmetical operators act on the contents of two cells in the way you'd expect. The three trigonometrical functions work as expected, as well. In addition to these seven operators, though, Microspread offers another nine. This is what they do:

$ - calculates the total of a range of cells.

A ANALYSE – makes one cell equal to another, depending on the value of a third. This gives you the option of creating 'cut-off' points in your calculations.

X Trend – tries to predict the next value in a series comprising the entries from a range of cells. For instance, if you take the *trend* of cells with contents 1, 2, 3 and 4, the prediction for the next number will be 5.

H Hi (max) – extracts the highest value from the cells in the given range.

L Lo (min) – extracts the lowest value from the cells in the given range.

R Range – calculates the difference between the highest and lowest values in the given range.

M Mean – calculates the average of the values in the given range.

% Box1*Box2/100 – calculates **Box2** percent of **Box1** (Microspread refers to cells as boxes).
P Box1/Box2*100 – calculates the fraction **Box 1 / Box 2**, as a percentage.

For the first formula you would choose Sum from the operator menu. Microspread then asks you to **FIX RESULTS BOX**. To do this you move the cursor to the required cell and press <ENTER>. This method is a little slower than typing the cell's co-ordinates, but you're less prone to key in the wrong references.

You're then asked for two further cell references. If the operator is **+, -, * or /**, the contents of these two cells will be operated upon, otherwise the cell references represent the start and finish positions of a range of cells.

Microspread will ask you to specify each cell reference in a formula as either absolute or relative. In **FORMULA 1**, in this example, both references are relative. Having completed the first formula, you can force Microspread to recalculate the spreadsheet by pressing <CTRL>C. The value **128000** should appear in cell **AC11**.

Although **FORMULA 1** refers to cells **AC4** and **AC10**, these references are made relatively. This means you can apply the same formula to other cells. You may, in fact, have wondered why the range of cells was **AC4** to **AC10**, when cells **AC9** and **AC10** are empty. The reason is that the same formula can be used to calculate the expenditure sub-total in **AG11**, and the range of cells will then need to be **AG4** to **AG10**, to ensure the figures for 'electricity' and 'telephone' are included.

To allocate FORMULA 1 to cell AG11 you type <CTRL>B A. <CTRL>B' selects the commands which act on a Block of cells, and A is the option to allocate a formula. This time you're asked to specify the top-left and bottom-right cells which mark out the block in which you're interested. You can allocate a formula to many cells at once, but in this case both the top left and bottom-right cells are AG11. If you recalculate after allocating this formula, the figure 124000 should appear in cell AG11.

Dual account book keeping requires that your income and expenditure should balance, so it would be useful if you could make the expenditure total mimic the income total. This is quite easily done, by defining a formula which adds '0' to the contents of cell AC11. FORMULA 2 does this by adding the contents of cell AA1 (which, being empty, is zero) to AC11. The AC11 reference is relative, so you can use the formula again, but the AA1 reference is absolute, as any cell using this formula will need to use cell AA1.

The profit figure is derived as the total income less the total expenditure, and FORMULA 3 performs this subtraction. It is important that the income total is taken from cell AC11, rather than the copy just created for cell AG13. This is due to the way the values in a spreadsheet are calculated.

All spreadsheets calculate starting from the top left-hand corner and work out each row in turn down the sheet, or each column, progressing from left to right. Either way, a cell to the left or above another will be calculated first. In this example the profit cell will be calculated *before* the expenditure total, but *after* the income total, hence it's the income total figure which must be used to calculate the profit.

If you recalculate the sheet at this stage, all the figures for the 1985 forecast should now look as shown in Figure 6.11.

The 1986 figures are entered into the sheet in one of three ways. The figures for Development Grant, Bank Loan and Private Loans are straight predictions, and are typed in as numbers. The totals, sub-total and profit figures are calculated in the same way as the 1985 figures. All you do is to assign the relevant formula to the corresponding 1986 cells.

The remaining figures, for Retail and Postal sales, and for all the expenditure categories, are calculated as a percentage increase due to inflation. The inflation rate is entered into cell AE15, and FORMULA 4 is defined to calculate the percentage increase of cell AD4 over AC4. This formula is then assigned to cell AD5 and to all the expenditure categories in cells AH4 to AH10. This last assignment is made just as before, using the block assignment function, and entering the top-left and bottom-right cells as AH4 and AH10, respectively.

Another recalculation will complete the forecast sheet, which should now be the same as Fig 6.12.

The interesting thing at this stage is to change the inflation rate held in *AE15* and watch how it affects the profit figure. In fact, changing any of the figures and recalculating will automatically update any of the other cells whose values depend on them.

Microspread allows you to print out the sheet, or to save it on disc. Both these functions are called by typing <CTRL> sequences from the keyboard. The print function allows you to specify your printer's width and the area of the spreadsheet you wish to print, by specifying its top left and bottom right cells. Two print-outs from the sample sheet, showing different values for the inflation index, are shown in Fig 6.12.

Microspread will handle many small to medium-sized spreadsheets, but may be struggling if you need to set up a really large model. Here again, the constraint is not so much the program, but the amount of available memory on the Amstrad micros.

INCOME AND EXPENDITURE FORECAST

Income	1985	1986	Expenditure	1985	1986
Retail Sales...	65000	68250	Salaries.....	43000	45150
Postal Sales...	33000	34650	Rent and Rates.	9500	9975
Dev. Grant.....	10000	10000	Materials.....	23000	24150
Bank Loan.....	7000	5000	Equipment.....	25000	26250
Private Loans..	13000	15000	Distribution...	13000	13650
			Electricity....	6000	6300
			Telephone.....	4500	4725
TOTAL	128000	132900	SUB TOTAL	124000	130200
			Profit.....	4000	2700
			TOTAL	128000	132900

Inflation over 100 base rate - 105

INCOME AND EXPENDITURE FORECAST

Income	1985	1986	Expenditure	1985	1986
Retail Sales...	65000	71500	Salaries.....	43000	47300
Postal Sales...	33000	36300	Rent and Rates.	9500	10450
Dev. Grant.....	10000	10000	Materials.....	23000	25300
Bank Loan.....	7000	5000	Equipment.....	25000	27500
Private Loans..	13000	15000	Distribution...	13000	14300
			Electricity....	6000	6600
			Telephone.....	4500	4950
TOTAL	128000	137800	SUB TOTAL	124000	136400
			Profit.....	4000	1400
			TOTAL	128000	137800

Inflation over 100 base rate - 110

Figure 6.12 - Two print-outs from sample spreadsheet.

6128 Special

The main advantage of the CPC 6128, as has been emphasised in previous chapters, is the increase in available memory. Because CP/M Plus can handle the extra 64K bank as extra text or data space, it effectively gives the machine a 61K tpa. This is more than many 'proper' CP/M programs expect, and the CPC 6128 will therefore run most of them without much adaptation.

Amstrad, of course, are keen to encourage this, as the more well-known software packages the machine will run, the more people are likely to consider it as a 'serious' machine. There is one big drawback with this approach, however. The majority of well-known CP/M business software is still expensive by home computer standards. It's not unusual to find packages costing considerably more than the price of the CPC 6128 itself! Clearly, if the availability of this type of software is to be used to boost sales of the computer, the price is going to need reducing.

For this reason, Amstrad has entered negotiations with some of the major CP/M software publishers to produce 'special' versions of their programs, which can be sold at more reasonable prices. The more enlightened of these publishers have seized the opportunity and have made available full versions of their programs at between £50 and £120.

The three programs to be described here are examples of these low-cost Amstrad versions. The word processor is WordStar, the program which has become the 'industry standard' against which other word processors are judged. The database is Cardbox Plus, a newer program, but one which has proved extremely popular. The spreadsheet is Supercalc 2, another program with huge sales, probably only bettered by the original electronic spreadsheet, Visicalc.

POCKET WORDSTAR

Although the name of this version of WordStar suggests it is in some way 'cut down', it is, to all intents and purposes, a full version of the program, and comes complete with 'Mailmerge'. Mailmerge is a separate program which handles the kind of mailing list application discussed in the section on Tasword 464D.

WordStar opens up with its 'OPENING MENU', which is similar in some of its functions to the main menu of Tasword, and is shown in Figure 6.13.

The WordStar menu offers a few more options, though, and the menu screen also carries a disc directory, sorted into alphabetical order. The menu is broken down into five categories: preliminary commands, commands to open a file, file commands, system commands and WordStar options. Most of these are obvious, but one or two might need explanation.

The help level, which you set by pressing 'H' from the opening menu, governs how many of WordStar's menus are displayed while using the program. Level 1 shows none of them, while level 3 shows them all. You can adjust the level as you become more familiar with the commands.

WordStar can deal with 'document' files, in which it will insert its formatting control codes, and 'non-document' files, which might, for instance, be the source code for a Pascal or C program. You can select which of these types to edit, by pressing 'D' or 'N' from the opening menu.

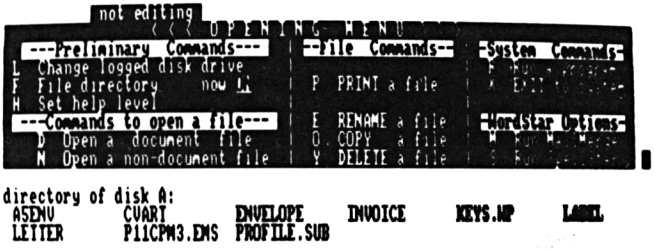


Figure 6.13 Wordstar opening menu

Unlike Tasword, WordStar allows you to run another program from within it, and then to exit from that program back into the word processor. At first sight that may not appear very useful, but you may want to run simple utilities such as DIR.COM to display a full directory, and you can do this by selecting the 'R' option from the opening menu.

To move from the opening menu to the editing screen you must open a document file by pressing 'D' and typing a file-name when prompted. Wordstar filenames must be compatible with CP/M, but you can use any filetype you like to help distinguish different types of document. You might, for instance, choose .LET for letters, .REP for reports or .QUO for quotes.

The editing screen displays a single status line at the top, giving the name of the file, the current position of the cursor as page, line and column numbers, and whether the insert mode is on or off. The 'MAIN MENU' sits below the status line, and shows a list of WordStar's more basic functions, and reminders of the other five menus which may be called in to replace it. Directly below this menu is the ruler line, showing margins and tabulations, and the rest of the screen is taken up by the text of your document. A typical WordStar editing screen is shown in Figure 6.14

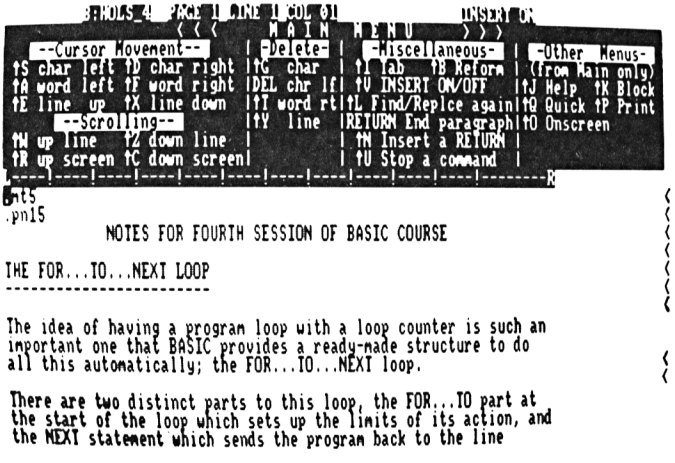


Figure 6.14 WordStar editing screen

As you can see from the main menu, all WordStar's commands are called up as <CTRL> sequences. This was a deliberate ploy by Micropro, the program's publisher. When WordStar was first written, the QWERTY keys were about the only ones guarenteed to be on the keyboards of all computers. By combining them with the <CTRL> key, also common to most machines, the program was made as 'transportable' as possible. Although the same is not true today, with nearly all micro's having cursor keys, for example, the old commands have remained to ensure compatibility with earlier versions of the program.

This reliance on <CTRL> sequences does mean that the newcomer to WordStar is faced with what seems to be a mass of incomprehensible command codes which have to be learnt before the word processor can be used. Contrary to popular belief, there is some logic in the choice of

sequences used, but it still comes down to a lot of learning by rote. To help you out, though, you can set up a PROFILE.SUB file on your WordStar work disc, with the instructions:

```
SETKEYS KEYS.WP
LANGUAGE 3
WS
```

on it. Make sure the files **SUBMIT.COM**, **SETKEYS.COM**, **KEYS.WP** and **LANGUAGE.COM** are also on the disc. The PROFILE file will automatically set the cursor, <DEL> and <CLR> keys up to produce the correct WordStar codes. You can then use the keys within WordStar.

WordStar doesn't limit the length of document to the available memory of your CPC 6128, but will move parts of the text on and off disc, quite automatically, as you come to edit them. This means that you may have a single document up to 160K long. Mind you, it would take quite a while to page through a file this long to edit something in the middle! The program shows the breaks between pages on the screen, so you can check for paragraphs or tables broken in awkward places.

As well as <CTRL> commands which normally affect the text immediately you use them, WordStar uses a series of 'dot' commands to embed instructions into the text. A dot command is a simple two letter command, preceded by a full stop (the dot), and sometimes followed by a numeric parameter. These commands affect the format of the text, which will often only be seen when you print it out. They cover such things as line spacing, top and bottom margins and the printing of page numbers. The codes again have to be learnt, but do generally suggest their functions.

WordStar can handle text up to 240 characters wide, and the screen will scroll horizontally to display it section by section. This facility won't be much use to you, of course, unless you have a printer capable of printing that many columns across the paper.

When you've finished editing a document, you can leave it by typing <CTRL>KD, which will first save the file to disc before returning you to the opening menu. WordStar automatically saves the document before allowing you to print it or leave the program. This 'safety first' policy makes it hard to lose text accidentally, and another built-in feature of the program keeps a back-up of the edited text, on a file with the same name as the main file, but with a .BAK filetype. This is also useful, but on Amstrad's low-capacity discs, you may soon find you run out of text space.

Printing your file is a straightforward business. Wordstar prompts you with a series of options, such as starting and stopping at given page numbers, and pausing between pages for a paper change. The codes for different printers may be installed into the program, so that commands within the text to underline, print boldface or shadow print are correctly translated and sent to your printer.

Although WordStar was written several years ago, and there are now many more 'user-friendly' word processors around, it refuses to be superceded, and as a de facto standard, is still one of the most comprehensive programs of its type available on any micro.

CARDBOX PLUS

This is a full-blown CP/M database program which tries to mimic a manual box of record cards as closely as possible. While you can't perform calculations on the data in Cardbox Plus records, the program's main strength is in it's searching facilities.

You can easily extract records which obey complex search rules, and create sub-sets of the data in the complete file. So, for instance, if you have a list of customer's addresses on a Cardbox Plus file, you can produce a list (or a set of address labels) of only those customers who live in a particular area, who have bought goods from you in the last year and who paid cash.

Before you can set about this kind of search, though, you'll need to set up the file 'cards' and enter your data onto the file. It's also worth while setting up a PROFILE.SUB file with the appropriate CP/M utilities included on your work disc, so you can start up the disc and select the right colour scheme, language ('£' for English) and key definitions. If you use the KEYS.WP file, you'll find that the cursor keys are correctly set up for Cardbox Plus.

When you first start Cardbox Plus the main menu screen is display. This is shown in Figure 6.15.

You select a primary and secondary function by pressing `P` and `S` respectively, until the arrows in the top half of the screen point to the functions you want.

To start with, you'll need to create a format for your record card, so you should select `Format definition` and `Create`. You then press `F` and type in a filename for the format file. Cardbox Plus stores the format of your records in a separate file from the actual data. If it had to store all the data in the format you define, this would take a lot more disc space.

You then press <ESC> followed by `G`, to perform the function you've selected. The screen clears to show some status information on the top line, and a number of menu options in the bottom four lines. The rest of the screen is left blank, for you to create your record card.

To create or edit a field, you select `F` and can then specify the length, position, name and caption for the field. The caption is the message that appears on the screen next to the field (Name, Address, Telephone No etc), while the name is a two character reference which Cardbox Plus ties to the field. You can define up to 26 fields on a card, which may be up to 100

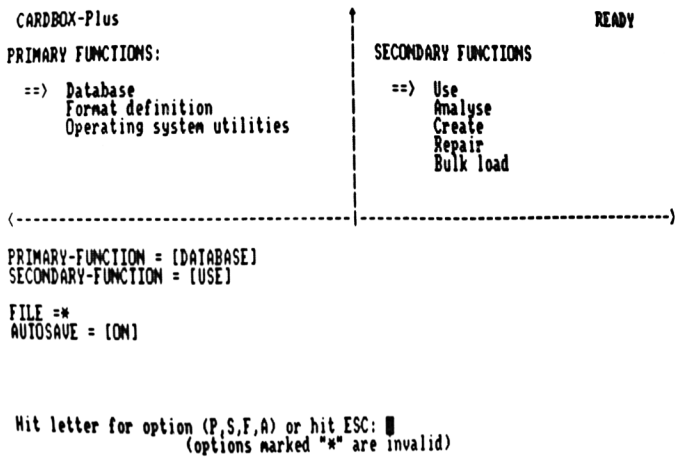


Figure 6.15 Cardbox Plus main menu

columns wide by 20 rows deep; a large area.

As well as the captions which refer to individual fields on each record, you can type in record titles and other 'background' text which will appear on every card. You can also use limited graphics to produce borders and separating lines on your card. As Cardbox Plus works under CP/M, though, there is no access to the built-in graphics of the CPC 6128.

Once you've finished designing the record card, you save the format and return to the main menu. This time you select **Database** and **Use**, and Cardbox Plus will present you with a blank card, identical to the one you designed (except that the fields are blank) ready for you to enter your own data. A Cardbox Plus data entry screen is shown in Figure 6.16.

This database has been set up as a customer address list; the kind of list which might be used by a sales representative dealing with a number of potential or actual customers for his/her company's products. On the left-hand half of the card is the contact's name, position, company address and telephone number, while on the right are details of successful sales, and notes and reminders. Cardbox Plus allows you to set up a field covering more than one line for text entry and will even wrap words around from one line to the next, like a word processor. This feature is used in the 'Item Bought' and 'Notes' fields.

A list of valid commands is shown at the bottom of the screen, and each of these may be entered by typing the first two letters of its name, or as a <CTRL> sequence. The four functions <CTRL>R, <CTRL>C, <CTRL>A and <CTRL>F control movement through the file, while the two-letter commands operate on the current record or the whole file.

```
 CARDBOX-Plus    FILE = A:SAMPLE.FIL     READY
Level  0 - RECORD 20 OF 35
--------------------------------------------------------------------------------
+                              CUSTOMER ADDRESS LIST                           +
--------------------------------------------------------------------------------
+                                      +                                       +
+   Forename: John                     +   Item Bought: Oil resistant          +
+    Surname: Drew                     +    tubing                             +
+   Position: Purchase Clerk           +                                       +
+                                      +   Price Paid: £ 2187.87               +
+    Company: Brosco Pumping Systems   +                                       +
+  Address 1: Brosco House             +   Purchase Date: 15/4/85              +
+  Address 2: 165 High Street          +                                       +
+  Address 3: Bravscombe               +   Payment Method: Cheque              +
+  City/Cnty: Cornwall                 +                                       +
+   Postcode: TR12 14JH                +                                       +
+                                      +                                       +
+  Telephone: 0375 678054              +                                       +
+                                      +                                       +
--------------------------------------------------------------------------------
Enter command:
  MAsk; SElect, INclude, EXclude; HIstory, BAck, CLear; TAg; KEep;
 ADd, DUplicate, EDit, DElete;  REad, WRite; FOrmat, PRint; SAve, QUit
LIST:  †R=1st †C=last †A=back †F=fwd TAB=tag  ENTRY:  ESC=erase †H=backspace
```

Figure 6.16 Cardbox Plus data screen, showing typical record 'card'.

At the top of the screen are details of the file you're using and the number of records on the current 'level'. Cardbox Plus uses the concept of levels to hold sub-sets, which you may extract from the file according to given rules or 'search criteria'. The complete file is always held on level 0.

The quickets way of selecting records from a file is to use fields which you've previously 'indexed'. When you're setting up your record card, you can tell Cardbox Plus to index certain fields (or all of them if you like). The program will then keep extra information about these fields, to speed up searches. You can also search on non-indexed fields, although more slowly.

In the customer address list example, it would probably be useful to have the `Surname, Position, City/Cnty, Item Bought` and `Payment Method` fields indexed. To select records with a certain content in one of these fields, you type `SE` (for SElect), the two-character field name and the word or number you want to search for. Cardbox Plus will then 'pull out' all the records which match your selection and put them on the next level. You can then scan through these and, if you want, apply another rule to select a smaller sub-set from these records.

You can go on doing this until you've isolated just the records you want. Cardbox Plus also provides functions to `undo` a single search, or to remove all selections and return you to the complete file on level 0. You can also

display a history of the selections you've made, to check you've used the rules you intended.

As well as selecting records from the complete file or a sub-set, you can also include or exclude records from other levels. By combining all these facilities, you can select the records you need very quickly.

Although Cardbox Plus doesn't offer all of the facilities of Masterfile, described earlier in this chapter, its ease of operation and exhaustive searching functions make it very useful for 'record card' type applications. Where it really outshines Masterfile, which even in the CPC 6128 version is based in memory, is that Cardbox can handle a file as big as a disc can hold (170K if you use twin drives). This removes any restriction on large files for most applications. If, some day soon, someone produces a hard disc for the CPC 6128, Cardbox will be able to handle a file of up to 8 Megabytes!

SUPERCALC 2

This is a 'full-blown' spreadsheet, which will allow up to 16000 cells. This should be sufficient for all home or small business uses, although in practice you're likely to run out of memory space before filling anywhere near this number of cells.

The master disc, supplied by Amsoft, contains routines to install a working copy on a separate disc, so you don't need to devise a PROFILE file for yourself. The working copy, once prepared, may be run by typing:

```
SC2
```

The main spreadsheet screen, which is shown in Figure 6.17, offers 20 rows by 8 columns. The rows are numbered 1 to 20, and the columns are lettered from A to H. At the bottom of the screen are three status lines which show a variety of information.

The top line shows the current cell co-ordinates, and displays prompts when you're entering commands. Supercalc 2 is very helpful in this department, and offers assistance with nearly everything you do. Nearly all commands are given with just a couple of characters, and there are no <CTRL> sequences to learn. This line also shows the contents of the current cell (text, a number or a formula) and whether the cell is protected or not. You can protect any cell so that its contents can't be inadvertantly wiped.

The middle line shows the width of the current cell, the amount of memory remaining for the sheet (with no sheet loaded, this is about 31K) the last row and column co-ordinates of the current sheet, and a reminder of how to display HELP information.

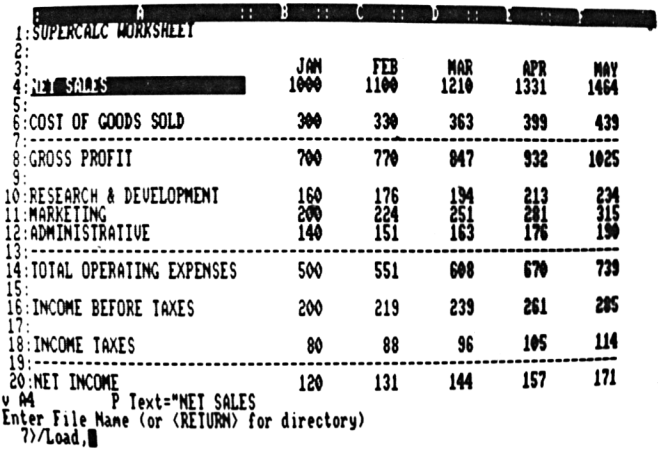


Figure 6.17 - Supercalc 2 main spreadsheet display

The bottom line echoes the commands you type at the keyboard. These are fairly easy to learn. To enter numbers, you move the main cursor to the required cell and type them in; it's as simple as that. Text input has to be preceded by a '"', to let the program know what's coming. Any other character is assumed to be the start of a formula, defining a relationship between cells.

Supercalc 2 offers a wide variety of mathematical functions. As well as the obligatory **+**, **-**, ***** and **/**, there are trigonometrical operators, some statistical ones and a LOOKUP function. This last feature allows you to create a look-up table within your spreadsheet and calculate any value depending on one from another cell, within the table.

When you've entered all the numbers and text into the sheet, you will probably need to manipulate the values to complete your 'model'. To do this, you call on Supercalc's picturesquely named 'slash' commands. You can call up a help screen to display all these commands, and this is shown in Figure 16.18.

All the slash commands start with the '/' (slash) character. Some of them are obvious from the HELP screen: /B for blanking a cell, /C, /D, /E and /I for copying, deleting, editing and inserting cells, for example. Others, perhaps

```
SuperCalc2 1  AnswerScreen  Slash Commands:
   A(rrange)--- Sorts cells in ascending or descending order.
   B(lank)----- Removes (empties) contents of cells.
   C(opy)------ Duplicates contents and display format of cells.
   D(elete)---- Erases entire rows or columns.
   E(dit)------ Allows editing of cell contents.
   F(ormat)---- Sets display format at Entry, Row, Column, or Global levels.
   G(lobal)---- Changes global display or calculation options.
   I(nsert)---- Adds empty rows or columns.
   L(oad)------ Reads spreadsheet (or portion) from disk into the workspace.
   M(ove)------ Inserts existing rows or columns at new positions.
   O(utput)---- Sends display or cell contents to printer, screen or disk.
   P(rotect)--- Prevents future alteration of cells.
   Q(uit)------ Ends the SuperCalc2 program.
   R(eplicate)- Reproduces contents of partial rows or columns.
   S(ave)------ Stores the current spreadsheet on disk.
   T(itle)----- Locks upper rows or left-hand columns from scrolling.
   U(nprotect)- Allows alteration of protected cells.
   W(indow)---- Splits the screen display.
   X(eXecute)-- Accepts commands and data from an .XQT file.
   Z(ap)------- Erases spreadsheet and format settings from workspace.

    Press any key to continue
```

Figure 16.18 - Supercalc 2 '/' command HELP screen

need a bit more explanation.

/R, the replicate function, copies formulae from one cell to another, or to a range of cells. .This greatly speeds up the entry of similar formulae into a number of cells. At each stage of the process, Supercalc 2 prompts you for the next piece of information.

Imagine you have two rows of figures in your spreadsheet, and you want to add each pair up and put their sum in the cell below:

```
   :    A    ::    B    ::    C    ::    D    ::    E    ::
1:      32.56      47.34      23.45      10.87      27.61
2:      23.98      65.89      14.67      28.65      69.11
3:[     56.54]

> A3                    Form=A1+A2
```

You first move to the cell which will take the result of the first addition. In the above example, this is A3. Type in:

```
A1,A2 <RETURN>
```

and the result of the addition will immediately appear in cell A3. The formula you typed in will be shown in the top status line.

To copy the formula into the cells 'B3' to 'E3' you press /R and the prompt line will show:

```
From? (Enter Range)
```

Here you enter the range of cells you want to copy from. In this case, it's the single cell A3, so just enter this cell reference and a comma. The prompt will then change to:

```
To? (Enter Range), then Return; or "," for Options
```

You want to enter the range B3 to E3, and you do this by entering the first and last cell reference, separated by a colon, ":". If you press <RETURN> at this stage, the formula 'A1+A2' will be copied into cells B3, C3, D3 and E3.

What is wanted in this case, however, is that each cell should take the sum of the two cells above it. The two cells added together should not be A1 and A2, but the two cells in the same *relative* positions as A1 and A2 are to A3. To do this, first type a comma after the 'To' range.

The prompt now changes again, to:

```
N(o Adjust), A(sk for Adjust), V(alues), +,-,*,/
```

To adjust the cell references (make them relative rather than absolute), press A. The next prompt is:

```
Source cell A3, Adjust A1 (Y or N)
```

This message means 'the cell being copyied from is A3. Do you want to make the reference to A1 relative?'. Type 'Y' here. The question is then repeated for A2, and when you press 'Y' again, the correct sums are calculated and displayed in cells B3 to E3. The finished screen segment looks like this:

```
   :    A    ::    B    ::    C    ::    D    ::    E    ::
 1:     32.56      47.34      23.45      10.87      27.61
 2:     23.98      65.89      14.67      28.65      69.11
 3:[    56.54]    113.23      38.12      39.52      96.72


> A3                    Form=A1+A2
```

The replication of formulae is one feature that makes Supercalc 2 such a powerful program. Another is the ability to split the screen and display two sections of a spreadsheet at once. This is particularly useful if you have a large sheet and want to compare figures in cells normally remote from each other.

To split the screen, you first move the cursor to the row or column along which you want to split the screen. Then, by typing /W, you can choose to split the screen horizontally or vertically, and to synchronise the two halves together. If you split the screen and synchronise the halves, then moving the window over the spreadsheet in one half of the screen will move it by a corresponding amount in the other half. This is useful if you have to rows or columns of figures which are remote from each other, but which you want to bring onto the screen and scan through simultaneously.

If you choose not to synchronise the two halves of a split screen, then moving the window in one half will have no affect on the other.

Supercalc 2 has many other features which together earn it the title of 'professional' software. There is only room here to highlight a couple of the more important features. It is hard to think of a more effective program for answering 'What if...?' questions on the CPC 6128.

This chapter has looked at three of the main business applications of a disc-based micro, and examined six specific programs to illustrate how these applications are implemented on the Amstrad micros. When using discs, however, there is another option available to those interested in storing data; write the program yourself. Some of the techniques involved in doing this will be examined in the next chapter.

Chapter 7

File Handling

As well as being a much faster medium for storing programs than cassette, the main advantage of discs is the fast data storage they provide. With discs it is quite possible to develop databases, and other applications, where the data is not all held in memory at once, but is transferred to and from disc as it's needed.

The most obvious application of this fast data storage is in the database, and this type of program will be used to illustrate the principles involved in handling a data file.

Techniques

There are three main techniques for handling data on a micro.

The most obvious, and most often used on cassette-based systems, is to load the data into the computer's memory in one go, either into an array or directly into an area of memory which has been 'partitioned off' for the purpose. The data is then manipulated by referring to the co-ordinates of the array, or by updating a 'pointer' variable, which holds the address of a given record in memory.

This method is very fast, because you're only ever dealing with information held in memory, but the size of the file is governed by the size of the micro's RAM. In the case of the CPC 464 and CPC 664 machines this is about 40K, less the size of the database program itself. A database program written within these restrictions can still be of considerable use, however. Witness Masterfile 464, which uses this technique as part of a powerful heirarchical database.

Later in this chapter, a complete small database program will be described in detail. It stores all its records in memory, allows records of over 200 characters each, and can hold 100 of these. In addition, it can manipulate 26 of these 100 record files, offering, in effect, a capacity of 2600 records.

The second technique for handling data is the *sequential* file. This is a file of records, recorded on disc, which are numbered sequentially. Each record is known to the controlling program by its number and when the data in a particular record is needed, all the preceding records are loaded into memory, and discarded, so that the required record can be loaded, and its data extracted. While this is a fairly simple arrangement to set up, it is limited by

its speed; if you have a long file, it will obviously take an appreciable time to load all the unwanted records.

The third method, which is the most efficient way of handling data from disc, is the *random access* file. As the name suggests, with this scheme any record on the file may be accessed individually, without affecting any other record. If you want to, you can select records completely at random.

The way this is achieved is similar to the first method described, when the whole database is in memory. With a random access file a pointer is again maintained, but this time it points to a particular point on the *disc*. When a record is called, the program uses this pointer to go straight to the right place and transfer the data.

It's all very well to describe a random access program as maintaining a pointer, but in programming terms how is this achieved? The easiest way is to fix the length of each record on the disc. Say, for instance, that each record in a random access file is 250 bytes long. Now, if you need to read record 27 in your file, your program only has to add 26*250=6500 to the address of the start of your file on disc, to know where to start reading the required record.

The obvious disadvantage of this method is that you may waste a lot of space on your disc. If you have an address list file where each record has a length of 100 characters, then you won't waste much room on the record for:

```
Peregrine Winstanley-Ffoulkes
Brizenorton Manor
Billowing-on-Slurry
nr Worcester
Worcestershire
```

but quite a bit on:

```
Jim Nutt
3 Fore Street
Lynn
Hants
```

The alternative method of working is to store each record as its actual length. This would allot 96 bytes on disc for Peregrine, but only 34 for Jim. This method has its own problems, though. Your trouble now lies in modifying records — what happens if Jim moves to:

```
Stanbottom Priory
Nunachuck Lane
Stanbottom-by-Mud
nr Moretonhampstead
Devon
```

You obviously can't write the new address where the old one was on the disc, because you'll overwrite the following record. The only thing you can do is to move all the other records. In fact, the way this is usually handled is to add a new record to the end of the file, with Jim's new address in it, and mark the old record as 'deleted'. Then, periodically, your program goes through your disc, shuffling all the undeleted records up together, and releasing space at the end of the file.

Having said that random access files are probably the best way of handling disc-based data, it may be a little disappointing to learn that Locomotive BASIC doesn't support random access. Some dialects of the language, BBC BASIC for instance, provide a PTR keyword which may be used to point to any part of a disc file. This facility, combined with the useful EOF function, which tells you when you've reached the End Of a File, make writing a random access program fairly straightforward.

A filing system under AMSDOS and Locomotive BASIC has to use sequential files, and, although there are ways of getting round the shortcomings of this type of file, you will need to resort to other operating systems and languages to put together a random access database. CP/M, for example, provides for random access files, and it is possible to use some CP/M-based languages to produce this type this type of file.

A Small Database Program

To show what can be done with BASIC and memory-based data, a simple, easy-to-use database program is now included.

The program is set up as a straightforward address file, which you can use to hold names, addresses and telephone numbers, but it would be very simple to convert it for use as stock records, a patient file or estate agent's register, to name but three applications.

The database allows you to enter and delete records, sort them into any order, search through them for a particular entry, print out a list or set of labels and to save the complete file on disc.

In fact, you can save up to 26 separate files, labelled 'A' to 'Z', each of which may hold up to 100 records. All in all, therefore, you can handle 2600 records with the system. In addition, you can store the results of a search (that is, all the records matching a given rule) in a separate file, which itself may be reloaded into the micro. This file may then be searched again, using a different rule, and the sub-set of the file saved back to disc, or printed out.

The emphasis throughout the program is on ease-of-use, and you are provided with instructions or information at most points in its operation. The main functions of the program are called by pressing keys on the numeric keypad, and movement about each record, and from one record to

another, makes use of the four cursor keys.

The program only works as listed on the CPC 464/664. A special version of the program, which uses the extra 64K of banked memory, is included in the 6128 Special section, at the end of this chapter

In Use

You run the program by typing:

```
run "address
```

or whatever other filename you give the listing when you save it. The operating screen is then displayed, and this is shown in Figure 7.1. The screen is divided into three main sections.

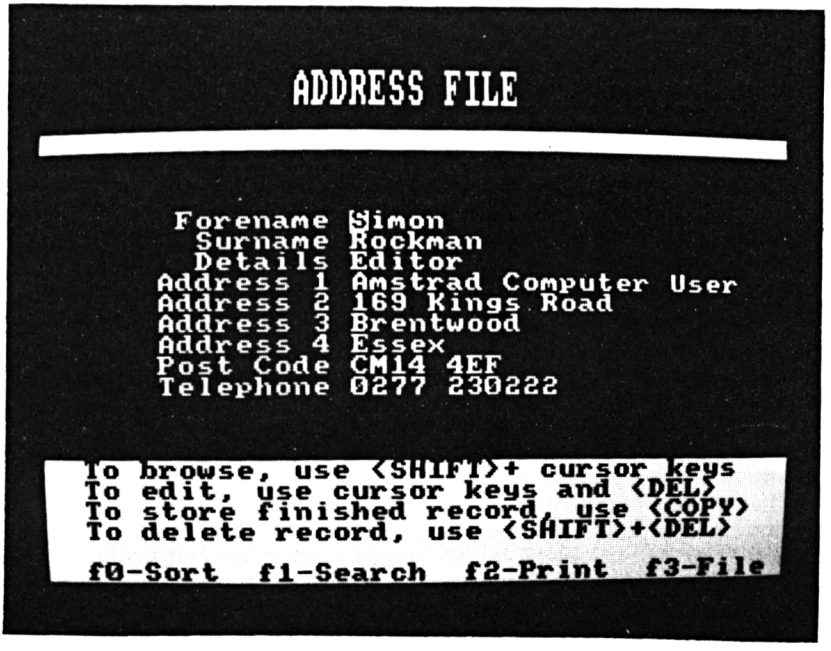


Figure 7.1 Main screen display, ram database

At the top, directly below the title, is a single white strip. This is where all the prompts and comments from the program are displayed.

Below this is a central area which shows the field titles and the contents of a

single record. Each new record which is displayed will overwrite the right-hand side of this section of the screen.

When you run the program you will either need to load a data file or create a new one by typing in names and addresses. Any of the letters A to Z may be used, when loading or saving files. In addition, the special file, @ , may be loaded, saved and displayed. Your selection is confirmed as the file is loaded. Once this is done, the first record on the file is displayed.

You may edit the record by overtyping what is displayed. The cursor keys move the cursor around the record, and <ENTER> and <DEL> work as you'd expect. Your edited record is not entered into the database, however, until you press <COPY>. The record is then read directly from the screen and replaces the old record in memory. The next record in sequence is then displayed.

To move from one record to another, you use the cursor keys, together with <SHIFT>. The left and right arrows will display the previous and next records respectively, while the up and down arrows will move you straight to the start or end of the file. When moving to the end of the file, a blank record will be displayed (assuming the file isn't full), so you can enter new data.

Once you have entered a few records, you will probably want to sort them into order. The program will do this for you, using its own sorting routine.

You select the sort option by pressing **0** on the numeric keypad. A column of figures will then appear down the left-hand side of the field titles. You select the field by which you want the file sorted by pressing the corresponding number key from the main keyboard. The file will then be sorted. This will only take a few seconds, even when the file is nearly full. This is because all the data is held in memory, and because a fast sorting algorithm is used.

When you have a lot of records in your file, you will find that it takes you a while to reach a given record by cycling through them with the cursor keys. A faster way of locating records is to use the program's built-in search function. This is called by pressing **1** on the numeric keypad.

After selecting the key field, as with the sort routine, you will need to type in the word or phrase you want searched for. This may be anything from 1 — 24 characters long (the length of any field in the record). You don't have to type in the complete word, just enough letters to distinguish it from any other entry.

Although the routine doesn't have to be given the complete key word, it does take note of the case of the characters used. If you're searching for **Winstanley-Ffoukes**, therefore, **Winsta** would probably find it, but **WINSTANLEY-FFOUKES** wouldn't.

Having typed in the search phrase, you are given the option to either display the results of the search, or to save them in the special file '@'. If you choose to display them, each matching record will be displayed in turn, with a prompt if there are more matches on file.

If you want to do a multiple search, however, such as finding all the people called **Nutt** who live in **London**, you can do this by directing the results of the first search into the '@' file.When the first search has been completed, finding all the **Nutts** on file, you can reload file '@' and search again, this time for London in the Address 4 field. The results of this search may again be displayed or re-stored. You can continue refining your search rules, to extract just the records you want. The program reports the number of matching records found at each stage.

Two different kinds of printout are available from the program, and these may be selected by calling the print option, which is key **2** on the keypad.

The two types of printout are a list and a set of labels. The first option is to select which of these you want. If you select a document, then the program will print out a list of every record on the file, heading the list with the file letter.

The labels option will either print a complete set of labels, spaced at a pitch of nine print lines (1.5 "), or allow you to select which to print. It does this by letting you browse through the records using the left and right arrow keys, together with <SHIFT>. Pressing <COPY> will print a label from the record on display. <ENTER> will complete the option and return you to the main editing screen.

The final facility available from the program is to load or save files on disc. There is no requirement to save a file loaded as, say, file 'A', back to disc as file 'A'. This leaves you completely free to manipulate the data between files as you wish. You might want to set up each file by its alphabetical contents; all surnames beginning with 'A' in file A, all 'B's in B etc. Alternatively, you could use file B for business contacts, file F for friends and R for relatives. The choice is yours.

The database program is modularised into a main control section and a number of subroutines, some of which call subroutines of their own. A brief description of how the program works should provide you with plenty of ideas for adapting it to your own uses, or writing your own.

The program loads data from disc into a two-dimensional string array, **array$**. This is a pretty efficient form of memory storage, since the byte overheads for strings are smaller than for numeric variables. If the database is used as an address list, there is no need for numeric variables, anyway.

The other main variables are `temp$`, a one dimensional array which is used for temporary storage of a single record, `recno%`, an integer variable which acts as a pointer to the record number being referenced, and `total%`, the current total number of records held in the file. The other variables quoted in the program are generally used as counters and flags.

The main section of the program runs from lines 10 to 350. The first section of this, lines 10 to 90, load up a section of machine code. This comprises two separate routines, one to provide double-height characters for the screen title, and the other to make a call to the Amstrad's operating system, to read a character from the screen.

```
10 MEMORY &A53B
20 FOR x=HIMEM+1 TO HIMEM+59
30 READ in$:POKE x,VAL("&"+in$):NEXT
40 DATA cd,60,bb,21,44,a5,77,c9,20,cd,06
50 DATA b9,f5,3e,00,cd,a5,bb,dd,21,32,90
60 DATA 06,08,7e,dd,77,00,dd,77,01,23,dd
70 DATA 23,dd,23,10,f2,f1,cd,0c,b9,3e,fe
80 DATA 21,32,90,cd,a8,bb,3e,ff,21,3a,90
90 DATA cd,a8,bb,c9
100 MODE 1:recno%=0:lastrec%=0:DIM temp$(8),array$(8,99)
110 KEY DEF 5,1,0:FOR n%=13 TO 15:KEY DEF n%,1,0:NEXT
120 INK 0,2:INK 2,26:BORDER 0
130 PEN 1:PAPER 0:CLS
140 x$="ADDRESS FILE":LOCATE 14,1:FOR i=1 TO LEN(x$):POKE &A54A,ASC(MID$(x$,i,1)
):CALL &A545
150 PRINT CHR$(254);CHR$(10);CHR$(8);CHR$(255);CHR$(11);:NEXT
160 WINDOW 7,15,8,17
170 PRINT " Forename  Surname  DetailsAddress 1Address 2Address 3Address 4Post C
odeTelephone";
180 PEN 0:PAPER 2:WINDOW 1,40,20,25:CLS
190 PRINT "  To browse, use <SHIFT>+ cursor keys      To edit, use cursor keys ar
d <DEL>       To store finished record, use <COPY>"
200 PRINT "  To delete record, use <SHIFT>+<COPY>"+CHR$(13)+CHR$(10)+CHR$(10)+"
 f0-Sort  f1-Search  f2-Print  f3-File";
210 WINDOW 17,40,8,18:PEN 2:PAPER 0:WINDOW#1,1,40,4,4:PEN#1,3:PAPER#1,2:CLS#1:WI
NDOW#2,5,5,8,17:PEN#2,3:PAPER#2,0
220 WHILE 1: CALL &BB81
230 IF INKEY(0)=32 THEN IF recno%>0 THEN recno%=0:GOSUB 21000
240 IF INKEY(2)=32 THEN IF recno%<=lastrec% THEN recno%=lastrec%+1:GOSUB 21000
250 IF INKEY(8)=32 THEN IF recno%>0 THEN recno%=recno%-1:GOSUB 21000
260 IF INKEY(1)=32 THEN IF recno%<=lastrec% THEN recno%=recno%+1:GOSUB 21000
270 IF INKEY(9)=32 THEN IF recno%<=lastrec% THEN FOR y%=recno% TO lastrec%:FOR x
%=0 TO 8:array$(x%,y%)=array$(x%,y%+1):NEXT:NEXT:lastrec%=lastrec%-1:GOSUB 2100C
280 IF INKEY(15)=0 THEN flag%=1:GOSUB 1000
290 IF INKEY(13)=0 THEN flag%=0:GOSUB 1000
300 IF INKEY(14)=0 THEN GOSUB 2000
310 IF INKEY(5)=0 THEN GOSUB 3000
320 IF lastrec%>98 THEN CLS#1:LOCATE 15,1:PRINT#1,"FILE  FULL" ELSE IF INKEY(9)=
0 THEN GOSUB 4000
330 in$=INKEY$:IF in$<>"" THEN GOSUB 5000
340 WEND
350 END
```

```
1000 REM To select sort/search of file
1010 PRINT#2,"123456789";
1020 CLS#1:PRINT#1,"    Select key field by number (1-9)"
1030 in$="":WHILE in$<"1" OR in$>"9":in$=UPPER$(INKEY$):WEND:i%=VAL(in$)-1
1040 CLS#2:LOCATE#2,1,i%+1:PRINT#2,CHR$(243)
1050 IF flag% THEN GOSUB 10000 ELSE GOSUB 11000
1060 CLS#1:CLS#2
1070 RETURN
2000 REM To print file as list
2010 text$="Print Document or Labels? (D/L)":GOSUB 22000
2020 IF in$="L" THEN GOSUB 13000:RETURN
2030 text$="   Set up printer and press <SPACE>    ":GOSUB 22000
2040 CLS#1:LOCATE#1,14,1:PRINT#1,"Printing file"
2050 PRINT#8,CHR$(13);"File ";file$;CHR$(10);CHR$(10):WIDTH 80
2060 FOR y%=0 TO lastrec%
2070 i$=array$(0,y%):GOSUB 24000
2080 PRINT#8,i$;" ";array$(1,y%)
2090 i$=array$(2,y%):GOSUB 24000:IF i$>"" THEN PRINT#8,i$
2100 FOR x%=3 TO 8
2110 i$=array$(x%,y%):GOSUB 24000
2120 IF i$>"" THEN PRINT#8,;i$;:IF x%<8 THEN PRINT#8,;", "; ELSE PRINT#8,CHR$(1
)
2130 NEXT:NEXT
2140 CLS#1
2150 RETURN
3000 REM To load/save file from/to disc
3010 text$="Load or Save file? (L/S)":GOSUB 22000
3020 IF in$="S" THEN GOSUB 15000:RETURN
3030 text$="Save current file first? (Y/N)":GOSUB 22000
3040 IF in$="Y" THEN GOSUB 15000
3050 ERASE array$:DIM array$(8,99):CLS
3060 GOSUB 14000
3070 RETURN
4000 REM To save record in memory
4010 LOCATE 1,1
4020 FOR x%=1 TO 9:in$="":n%=1:flag%=0
4030 WHILE flag%=0
4040 CALL &A53C:i%=PEEK(&A544):IF i%=44 THEN i%=128
4050 IF i%=32 THEN PRINT CHR$(9);:CALL &A53C:PRINT CHR$(8);:IF PEEK(&A544)=32 T
EN flag%=1:PRINT CHR$(13);CHR$(10);
4060 IF flag%=0 THEN in$=in$+CHR$(i%):PRINT" ";
4070 n%=n%+1:IF n%=25 THEN flag%=1
4080 WEND
4090 array$(x%-1,recno%)=in$
4100 NEXT:IF recno%>lastrec% THEN lastrec%=recno%
4110 recno%=recno%+1:GOSUB 21000
4120 RETURN
5000 REM To edit record on screen
5010 in%=ASC(in$)
5020 IF in%=240 THEN IF VPOS(#0)>1 THEN PRINT CHR$(11);
5030 IF in%=241 THEN IF VPOS(#0)<9 THEN PRINT CHR$(10);
5040 IF in%=242 THEN IF VPOS(#0)>1 OR POS(#0)>1 THEN PRINT CHR$(8);
5050 IF in%=243 THEN IF VPOS(#0)<9 OR POS(#0)<24 THEN PRINT CHR$(9);
5060 IF in%=13 THEN IF VPOS(#0)<9 THEN PRINT CHR$(13)+CHR$(10);
5070 IF in%=127 THEN IF VPOS(#0)>1 OR POS(#0)>1 THEN PRINT CHR$(8)+CHR$(16);
5080 IF in%>31 THEN IF in%<127 THEN PRINT CHR$(in%);:IF VPOS(#0)>9 THEN LOCATE
,1
5090 RETURN
```

```
10000 REM To sort file in memory
10010 CLS#1:LOCATE#1,14,1:PRINT#1,"Sorting file"
10020 FOR pntr1%=0 TO lastrec%-1
10030 FOR pntr2%=pntr1%+1 TO lastrec%
10040 IF array$(i%,pntr2%)<array$(i%,pntr1%) THEN FOR n%=0 TO 8:temp$(n%)=array$
(n%,pntr2%):array$(n%,pntr2%)=array$(n%,pntr1%):array$(n%,pntr1%)=temp$(n%):NEXT
10050 NEXT:NEXT
10060 recno%=0:GOSUB 21000
10070 RETURN
11000 REM To search file and display
11010 CLS#1:PRINT#1," Search for? >                              <"
11020 search$="":LOCATE#1,15,1
11030 WHILE LEN(search$)<25 AND INKEY(18)<>0
11040 in$=INKEY$
11050 IF in$=CHR$(127) THEN IF search$>"" THEN search$=LEFT$(search$,LEN(search$
)-1):PRINT#1,CHR$(8);CHR$(16);
11060 IF in$>=" " AND in$<"{" THEN search$=search$+in$:PRINT#1,in$;
11070 WEND
11080 text$="Display or Store in file '@'? (D/S)":GOSUB 22000
11090 IF in$="S" THEN GOSUB 12000:RETURN
11100 recno%=0:m%=-1
11110 WHILE recno%<=lastrec%
11120 IF search$=LEFT$(array$(i%,recno%),LEN(search$)) THEN IF m%=-1 THEN GOSUB
21000:m%=recno% ELSE text$="    MORE - press <SPACE>    ":GOSUB 22000:GOSUB 210(
0:m%=recno%
11130 recno%=recno%+1
11140 WEND
11150 IF m%=-1 THEN recno%=0:text$="     No match found - press <SPACE>    ":GO:
UB 22000:GOSUB 21000 ELSE recno%=m%
11160 RETURN
12000 REM To search file and store
12010 CLS#1:LOCATE#1,13,1:PRINT#1,"Searching file"
12020 OPENOUT "@"
12030 m%=0:FOR y%=0 TO lastrec%
12040 IF search$=LEFT$(array$(i%,y%),LEN(search$)) THEN FOR x%=0 TO 8:PRINT#9,a
ray$(x%,y%):NEXT:m%=m%+1
12050 NEXT
12060 CLOSEOUT
12070 text$="  Matches:"+STR$(m%)+" found - press <SPACE>    ":GOSUB 22000
12080 RETURN
13000 REM To print file as labels
13010 text$="Whole file or Selected records? (W/S)":GOSUB 22000
13020 IF in$="W" THEN text$="   Set up printer and press <SPACE>    ":GOSUB 220(
0:FOR y%=0 TO lastrec%:GOSUB 23000:NEXT:CLS#1:RETURN
13030 CLS#1:PRINT#1,"<SHIFT>+";CHR$(242);"&";CHR$(243);"Select <COPY>Print <ENT!
R>End";
13040 WHILE INKEY(6)<>0
13050 IF INKEY(8)=32 THEN IF recno%>0 THEN recno%=recno%-1:GOSUB 21000
13060 IF INKEY(1)=32 THEN IF recno%<lastrec% THEN recno%=recno%+1:GOSUB 21000
13070 IF INKEY(9)=0 THEN y%=recno%:GOSUB 23000
13080 WEND:CLS#1:LOCATE 1,1
13090 RETURN
```

```
14000 REM To load file from disc
14010 text$="LOAD":GOSUB 20000
14020 CLS#1:LOCATE#1,13,1:PRINT#1,"Loading file ";in$:file$=in$
14030 OPENIN file$
14040 y%=0:WHILE NOT EOF
14050 FOR x%=0 TO 8
14060 INPUT#9,array$(x%,y%)
14070 IF array$(x%,y%)="" THEN array$(x%,y%)=" "
14080 NEXT:y%=y%+1
14090 WEND:lastrec%=y%-1
14100 CLOSEIN:CLS#1
14110 recno%=0:GOSUB 21000
14120 RETURN
15000 REM To save file to disc
15010 text$="SAVE":GOSUB 20000
15020 CLS#1:LOCATE#1,13,1:PRINT#1,"Saving file  ";in$
15030 OPENOUT in$
15040 FOR y%=0 TO lastrec%
15050 FOR x%=0 TO 8
15060 PRINT#9,array$(x%,y%)
15070 NEXT:NEXT
15080 CLOSEOUT:CLS#1
15090 RETURN
20000 REM To read filename from keyboard
20010 CLS#1:LOCATE#1,7,1:PRINT#1,"File to ";text$;"? (A to Z or @)"
20020 in$="":WHILE in$<"@" OR in$>"Z":in$=UPPER$(INKEY$):WEND
20030 RETURN
21000 REM To display record on screen
21010 CLS:FOR x%=0 TO 8
21020 LOCATE 1,x%+1:PRINT array$(x%,recno%)
21030 NEXT:LOCATE 1,1
21040 RETURN
22000 REM To ask Yes/No question
22010 CLS#1:LOCATE#1,(40-LEN(text$))/2,1
22020 PRINT#1,text$:in$=""
22030 WHILE in$<>LEFT$(RIGHT$(text$,4),1) AND in$<>LEFT$(RIGHT$(text$,2),1):in$
UPPER$(INKEY$):WEND
22040 RETURN
23000 REM To print a label
23010 i$=array$(0,y%):GOSUB 24000
23020 PRINT#8,i$;" "array$(1,y%)
23030 i%=1:FOR x%=2 TO 7
23040 i$=array$(x%,y%):GOSUB 24000
23050 IF i$>"" THEN PRINT#8,i$:i%=i%+1
23060 NEXT:FOR x%=1 TO 9-i%:PRINT#8:NEXT
23070 RETURN
24000 REM To strip spaces from a string
24010 WHILE RIGHT$(i$,1)=" "
24020 i$=LEFT$(i$,LEN(i$)-1)
24030 WEND
24040 RETURN
```

Figure 7.2 Listing of RAM Database Program

Lines 100 to 220 define the arrays, draw the main display, and set up suitable text windows. If you want to use the database as something other than an address list, all you need to do is alter the strings in lines 140 and 170. You would probably also be safe in increasing the size of `array$` to hold 120 records, but you would also need to change the test constant in line 330 to '118'. The limiting factor is, once again, the size of the CPC464/664's memory.

Lines 230 to 350 are the main control loop. Several keys are repeatedly checked, and calls are made to the subroutines controlling the various functions of the program.

The main sub-routines are held between lines 1000 and 5090. These, in turn, call routines between lines 10000 and 15090. The general-purpose sub-routines are held between 20000 and 24040.

Lines 1000 to 1070 handle the section of program common to the sort and search functions, the selection of the key field. The sort or search routines are then called, as appropriate.

Lines 2000 to 2150 deal with the printing of a list from the records in the database. If labels are required instead of a list, then a further subroutine is called to handle this. Printing on the Amstrad micros is simply achieved by directing output to stream #8, using the **PRINT#8** command.

Lines 3000 to 3070 are the section of the program common to the load and save functions. If the user wants to load a new file, a check is made to see if the current file should be saved first.

Lines 4000 to 4120 read the record on display and store it in **array$**. **CALL &A53C** calls the machine code, which itself calls the operating system and returns the ASCII value of the character at location &A544. The display is wiped, character by character, as the record is read into the array. To speed this process, the routine is designed not to read beyond the end of the displayed data on each line of the record.

The last of these main subroutines is held from line 5000 to line 5090. This is the main edit loop. If none of the function keys has been pressed, the main control loop calls this routine to deal with cursor movement and the placing of characters on the screen.

The next level of subroutines, all of which are called from the ones just detailed, is the place where the majority of the 'work' of the program is done.

Lines 10000 to 10070 are the sort routine. As you can see, the routine is very compact, using nested FOR...NEXT loops. It uses the *interchange sort* algorithm, which is faster than some others because it makes fewer data swaps to get the records in order.

Lines 11000 to 11160 search the file and display any records which match the search string. A branch is made from this subroutine if the result is to be stored on disc rather than displayed. The search routine itself lies between lines 11100 and 11140. Note the use of **LEFT$$** in line 11110, which ensures that matching records longer than the search string are found.

Lines 12000 to 12080 search the file, but store the result on disc.

Lines 13000 to 13090 handle the printing of labels, either in one go in a FOR...NEXT loop, or individually. The printing of each individual label is dealt with by a further subroutine, called from within this one.

The routines beginning at 14000 and 15000 load a file from, and save it to, disc. This is done by opening an input or output file, and inputing or printing the data on stream #9 .

The general-purpose routines cover four particular tasks used in various places in the program.

Lines 20000 to 20030 read the filename letter for loading or saving a file.

Lines 21000 to 21040 transfer a record from `array$` to the screen.

Lines 22000 to 22040 handle the display of a question with a 'yes/no' type of answer, and the extraction of that answer from the keyboard. This type of question is used widely throughout the program.

Lines 23000 to 23070 print a single label from a record.

With a program of this length (under 8K), the structured, top-down approach isn't crucial. Database programs are usually a lot longer and more involved than this, however, and the advantages of this style of programming become more apparent the longer the program becomes.

Although the idea of holding all the data in memory is restrictive at present, with the cost of RAM dropping all the time, and memory capacities increasing, it may not be long before this kind of storage becomes the norm.

In the mean time, the only way of holding a lot more data on file is to store and retrieve it directly from disc when it's needed. As detailed earlier in this chapter, there are two ways of doing this. You can either have a *sequential* or random *access* files. A sequential file means that each piece of data has to be read in turn before you can get to the one you want. This is the way a file would have to be handled on tape.

With discs, however, it is possible to move the drive-head to any part of the disc, and read the data directly. All that's needed is to build some kind of 'map' of the disc into the program, so the micro knows where to send the head to read a specific piece of data.

Some languages on some computers provide all that's necessary to do this easily. Unfortunately, Locomotive BASIC doesn't have these facilities, because the Amstrad operating software is only designed to support sequential files. CP/M does support random access, which is one of its main claims to fame, but none of the languages covered in this book support random access directly.

An idea of what's needed, however, can be gained from looking at a memory-based simulation. This may be written conveniently in Locomotive BASIC, and shows the principles behind the technique. A 'cheat' version, which uses pseudo random access, will also be discussed. This pseudo random access technique is still a lot quicker than dealing with a sequential file on disc.

Simple Random Access Filing

To show how a simple random access filing system might be implemented, we'll set one up in a section of the computer's memory. This can be thought of as an area of the disc, although the equivalent commands to **PEEK** and **POKE** aren't available for working with discs.

First you'll need to reserve a section of memory to act as the 'disc'. The easiest way to do this is to move the end of memory used by BASIC. This address is held in the variable **HIMEM**, so by issuing the command:

```
MEMORY HIMEM-1000
```

you can reserve 1000 bytes for your file. This should only be done once, before you start using the program, otherwise you'll keep reserving extra memory, which the program doesn't use. The final, disc-based program will do this reservation for you.

You can either have variable-length records, where the record is as long as the data in it, or the record length can be fixed at a reasonable maximum. The latter method is simpler to do, but can waste space on your 'disc'. For ease of programming, though, this example will use 20 × 50 byte, fixed-length records.

Assume you want to store names and telephone numbers on the 'disc'. You would probably want to allow about 15 bytes for the corresponding name.

To make sure each new record overwrites anything already on the 'disc' at that point (when updating a telephone number, for instance), any name or number less than the full length of the record needs to be padded with spaces. This is easily done in a couple of string variables before saving the data.

To make sure the data is recorded in the right place on the 'disc', a 'pointer' is created, and this variable holds the address of the part of the 'disc' you're using. In this example, it would start at **HIMEM+1**, and increase by 50 for each new record added.

Loading a record from the 'disc' could be done by quoting its number, or by searching each record for a match to a given search 'key'.

A random access file has the further advantage of being easy to sort efficiently. You *can* go through the disc, swapping records that are out of order into their proper places, but it is quicker to keep a list of the record numbers in memory. If records need swapping to sort them into order, you can then swap *just the numbers*, and leave the 'disc' alone. Although you need to read the 'disc' to find out which records need swapping, you save two 'disc' write operations on each swap.

```
10 ptr=HIMEM+1:filend=ptr
20 name$=""
30 number$=""
40 disc$=""
50 CLS
60 PRINT "1 Add record
70 PRINT "2 Change record
80 PRINT "3 Delete record
90 PRINT "4 Find record
95 PRINT "5 List records
100 PRINT:PRINT:i%=0
110 PRINT "Select option (1-5)"
120 WHILE i%<1 OR i%>5:i%=VAL(INKEY$):WEND
130 ON i% GOTO 200,300,400,500,600
140 '
150 '
200 CLS:PRINT "ADD RECORD"
210 IF filend=HIMEM+1001 THEN PRINT:PRINT "FILE FULL - press any key":WHILE INKE
Y$="":WEND:GOTO 50
220 GOSUB 1000
230 GOSUB 2000
240 ptr=filend
250 GOSUB 5000
260 filend=filend+50
270 GOSUB 6000
280 IF i$="Y" GOTO 200 ELSE 50
300 CLS:PRINT "CHANGE RECORD"
310 GOSUB 1000
320 GOSUB 3000
330 IF ptr<HIMEM+951 THEN PRINT:PRINT "Enter new name/number":GOSUB 1000:GOSUB 2
000:GOSUB 5000
340 GOSUB 6000
350 IF i$="Y" GOTO 300 ELSE 50
400 CLS:PRINT "DELETE RECORD"
410 GOSUB 1000
420 GOSUB 3000
430 IF ptr=HIMEM+951 GOTO 470
440 i$="":PRINT:PRINT "Confirm deletion (Y/N)"
450 WHILE i$<>"Y" AND i$<>"N":i$=UPPER$(INKEY$):WEND
460 IF i$="Y" THEN FOR del=ptr TO filend-1: POKE del,PEEK(del+50):NEXT:filend=fi
lend-50
470 IF i$="N" THEN PRINT:PRINT "Record NOT deleted"
480 GOSUB 6000
490 IF i$="Y" GOTO 400 ELSE 50
500 CLS:PRINT "FIND RECORD"
510 GOSUB 1000
520 GOSUB 3000
530 GOSUB 6000
540 IF i$="Y" GOTO 500 ELSE 50
600 CLS:PRINT "LIST RECORDS":PRINT
610 ptr=HIMEM+1
620 WHILE ptr<filend
630 GOSUB 4000
640 PRINT LEFT$(disc$,35);RIGHT$(disc$,15);
650 PRINT
660 ptr=ptr+50
670 WEND
680 PRINT:PRINT "Press any key"
690 WHILE INKEY$="":WEND:GOTO 50
```

```
1000 ' ENTER NAME
1010 PRINT:INPUT;"Name: ",name$
1020 IF LEN(name$)<35 THEN name$=name$+SPACE$(35-LEN(name$))
1030 IF LEN(name$)>35 THEN name$=LEFT$(name$,35)
1040 RETURN
2000 ' ENTER NUMBER
2010 PRINT:INPUT;"Tel. No.: ",number$
2020 IF LEN(number$)<15 THEN number$=number$+SPACE$(15-LEN(number$))
2030 IF LEN(number$)>15 THEN number$=LEFT$(number$,15)
2040 RETURN
3000 ' FIND RECORD
3010 ptr=HIMEM+1
3020 WHILE name$<>LEFT$(disc$,35) AND ptr<HIMEM+1001
3030 GOSUB 4000
3040 ptr=ptr+50
3050 WEND
3060 IF name$<>LEFT$(disc$,35) AND ptr=HIMEM+1001 THEN disc$=SPACE$(35)+"NOT FOU
ND      "
3070 ptr=ptr-50
3080 PRINT:PRINT "Tel. No.: ";RIGHT$(disc$,15)
3090 RETURN
4000 ' READ RECORD
4010 disc$=""
4020 FOR i%=0 TO 49
4030 disc$=disc$+CHR$(PEEK(ptr+i%))
4040 NEXT
4050 RETURN
5000 ' WRITE RECORD
5010 disc$=name$+number$
5020 FOR i%=0 TO 49
5030 POKE ptr+i%,ASC(MID$(disc$,i%+1,1))
5040 NEXT
5050 RETURN
6000 ' LOOP?
6010 i$=""
6020 LOCATE 1,12:PRINT "Any More? (Y/N)"
6030 WHILE i$<>"Y" AND i$<>"N":i$=UPPER$(INKEY$):WEND
6040 RETURN
```

Figure 7.3 Listing of memory-based random access filing example.

Figure 7.3 gives a listing of the simple example that's just been discussed. Hopefully the ideas put forward here will give you an idea of how the system works. There's nothing mysterious about random access filing, although the intricacies of reading and writing data to and from real discs add some complications to the technique.

The listing of the program itself should be fairly clear. It is subdivided into sections for the various functions of the file, and the subroutines that are called from these sections.

The five main functions will add a new record, change, delete or find an existing one, or list the entire file. The subroutines controlling the 'disc' itself only form 12 lines (between 4000 and 5050). The majority of the program, as with most business programs, is involved with screen display and error checking.

It's very important to try and predict the kinds of mistakes a user will make, and to guard against them. One of the many corollaries of Murphy's Law states that the more unlikely an operator error is predicted to be, the more people will make it, and the more disasterous an effect it will have on the

program.

The idea of using plenty of subroutines to control the execution of the program is not just to make it clearer, although this is a good end in itself, but also to keep the written code to a minimum. A good general rule is *If you perform the same action more than twice, it should be in a subroutine.*

There are only two key variables in this program: `ptr` and `filend`. `ptr` is moved freely around by the program, to point at the record being dealt with; `filend` acts as a marker for the last record on the file at any time. When you add a record, `filend` is increased by 50 bytes, and when you delete one, it's decreased by 50 bytes.

A record is built up by taking the keyboard entries, which are stored in `name$` and `number$`, padding them up to the right lengths and adding them together in `disc$`. `disc$` is then `POKEd`, character by character, onto the 'disc', starting at the location held in `ptr`.

Reading the record back simply reverses this process. The bytes are `PEEKed` from the 'disc' and put into `disc$`, which is then split into its left-and right-hand components for display.

The next stage of development of this program is to make use of a 'real' disc drive, the DDI-1 or the drive on your CPC664. What happens is that the memory-based 'disc' used in the previous program becomes a *buffer*; a place where the data can be stored temporarily, before being transferred to disc.

Each block of 1000 bytes is moved to and from the disc as it's needed. The person using the program is unaware of these transfers of data, other than having to wait occasionally for the disc to be read or written to. The file appears to be one big database, although in actual fact the program is dealing automatically with several sequential disc files. The disc buffer in memory can be much bigger than 1000 bytes, and the program would then need to make fewer transfers. It depends how long your program is, and how much data you want to store.

Pseudo random access filing

In the revised version of the program, listed in Figure 7.4, you can have as many disc files of 1000 bytes as you need. They are dynamically allocated by the program, which means that a new file is only created when it's needed. This method of storing data is very economical, and allows you to have other programs on your disc at the same time as the database.

```
10 MODE 2: PRINT "OPENING FILE"
20 buffer=41360:MEMORY buffer
30 OPENOUT "dummy"
40 MEMORY HIMEM-1
50 CLOSEOUT
60 ptr=buffer
70 name$="":number$=""
80 disc$="":file%=1
90 OPENIN "filespec.dat"
100 INPUT#9,filend,lastfile%
110 CLOSEIN
120 IF file%<lastfile% THEN buffend=buffer+999 ELSE buffend=filend
130 GOSUB 7000
140 CLS
150 PRINT "1 Add record
160 PRINT "2 Change record
170 PRINT "3 Delete record
180 PRINT "4 Find record
190 PRINT "5 List records
200 PRINT "6 Finish session
210 PRINT:PRINT:i%=0
220 PRINT "Select option (1-6)"
230 WHILE i%<1 OR i%>6:i%=VAL(INKEY$):WEND
240 ON i% GOTO 300,400,500,600,700,800
250 '
260 '
300 CLS:PRINT "ADD RECORD"
310 IF file%<lastfile% THEN GOSUB 8000:file%=lastfile%:GOSUB 7000:buffend=filend
320 IF buffend=buffer+999 THEN PRINT:PRINT "BUFFER FULL - please wait":GOSUB 800
0:buffend=buffer:filend=buffend:file%=file%+1:lastfile%=lastfile%+1:GOTO 300
330 GOSUB 1000
340 GOSUB 2000
350 ptr=buffend+1
360 GOSUB 5000
370 buffend=buffend+50:filend=filend+50
380 GOSUB 6000
390 IF i$="Y" GOTO 300 ELSE 140
400 CLS:PRINT "CHANGE RECORD"
410 GOSUB 1000
420 GOSUB 3000
430 IF ptr<buffer+1000 THEN PRINT:PRINT "Enter new name/number":GOSUB 1000:GOSUB
 2000:GOSUB 5000
440 GOSUB 6000
450 IF i$="Y" GOTO 400 ELSE 140
500 CLS:PRINT "DELETE RECORD"
510 GOSUB 1000
520 GOSUB 3000
530 IF ptr=buffer+1000 GOTO 580
540 i$="":PRINT:PRINT "Confirm deletion (Y/N)"
550 WHILE i$<>"Y" AND i$<>"N":i$=UPPER$(INKEY$):WEND
560 IF i$="Y" THEN FOR del=ptr TO buffend-50: POKE del,PEEK(del+50):NEXT:FOR del
=buffend-49 TO buffend:POKE del,0:NEXT:IF file%=lastfile% THEN filend=filend-50
570 IF i$="N" THEN PRINT:PRINT "Record NOT deleted"
580 GOSUB 6000
590 IF i$="Y" GOTO 500 ELSE 140
600 CLS:PRINT "FIND RECORD"
610 GOSUB 1000
620 GOSUB 3000
630 GOSUB 6000
640 IF i$="Y" GOTO 600 ELSE 140
```

```
700 GOSUB 8000:file%=1
710 WHILE file%<=lastfile%
720 CLS:PRINT "LIST RECORDS":PRINT
730 GOSUB 7000:ptr=buffer
740 IF file%<lastfile% THEN buffend=buffer+999 ELSE buffend=filend
750 WHILE ptr<buffend:GOSUB 4000
760 PRINT LEFT$(disc$,35),RIGHT$(disc$,15);:PRINT
770 ptr=ptr+50:WEND
780 file%=file%+1:PRINT:PRINT "Press any key"
790 WHILE INKEY$="":WEND:WEND:file%=lastfile%:GOTO 140
800 CLS:PRINT "CLOSING FILE":PRINT
810 GOSUB 8000
820 OPENOUT "filespec.dat"
830 PRINT#9,filend,lastfile%
840 CLOSEOUT
850 CLS:PRINT "FILE CLOSED":PRINT
860 END
1000 ' ENTER NAME
1010 PRINT:INPUT;"Name: ",name$
1020 IF LEN(name$)<35 THEN name$=name$+SPACE$(35-LEN(name$))
1030 IF LEN(name$)>35 THEN name$=LEFT$(name$,35)
1040 RETURN
2000 ' ENTER NUMBER
2010 PRINT:INPUT;"Tel. No.: ",number$
2020 IF LEN(number$)<15 THEN number$=number$+SPACE$(15-LEN(number$))
2030 IF LEN(number$)>15 THEN number$=LEFT$(number$,15)
2040 RETURN
3000 ' FIND RECORD
3010 GOSUB 8000:file%=1
3020 WHILE file%<=lastfile% AND name$<>LEFT$(disc$,35)
3030 GOSUB 7000:ptr=buffer
3040 IF file%<lastfile% THEN buffend=buffer+999 ELSE buffend=filend
3050 WHILE ptr<buffend AND name$<>LEFT$(disc$,35)
3060 GOSUB 4000:ptr=ptr+50
3070 WEND
3080 ptr=ptr-50
3090 IF ptr>buffend THEN file%=file%+1
3100 WEND
3110 IF file%>lastfile% AND name$<>LEFT$(disc$,35) THEN disc$=SPACE$(35)+"NOT FO
UND      "
3120 PRINT:PRINT "Tel. No.: ";RIGHT$(disc$,15)
3130 RETURN
4000 ' READ RECORD
4010 disc$=""
4020 FOR i%=0 TO 49
4030 disc$=disc$+CHR$(PEEK(ptr+i%))
4040 NEXT
4050 RETURN
5000 ' WRITE RECORD
5010 disc$=name$+number$
5020 FOR i%=0 TO 49
5030 POKE ptr+i%,ASC(MID$(disc$,i%+1,1))
5040 NEXT
5050 RETURN
6000 ' LOOP?
6010 i$=""
6020 LOCATE 1,12:PRINT "Any More? (Y/N)"
6030 WHILE i$<>"Y" AND i$<>"N":i$=UPPER$(INKEY$):WEND
6040 RETURN
7000 ' LOAD BUFFER FROM DISC
7010 i$=RIGHT$(STR$(file%),1)+"FILE.BIN"
7020 LOAD i$,41359
7030 RETURN
8000 ' SAVE BUFFER TO DISC
8010 i$=STR$(file%)+"FILE"
8020 SAVE i$,b,41359,1000
8030 RETURN
```

Figure 7.4 - Listing of disc-based pseudo-random access example

This program is based on the original database of Figure 7.3, with the necessary extensions to save and load the 1000 byte buffer to and from disc. The buffer-fulls of data are saved on disc as binary files.

In addition to the straightforward saving and loading of the data buffer when you start and finish each session, the program's search and list routines have to be modified. These modifications ensure the routines start from the first data file on the disc and work through them to the end.

The `ADD RECORD` routine is also amended, so that new records are always added to the end of the last file on the disc. When a buffer is filled, an attempt to add an extra record will open a new file, rather than reporting `FILE FULL`, as the previous program did. Each new file is saved as `1FILE.BIN`, `2FILE.BIN`, `3FILE.BIN`, etc.

Now the program loads its records from disc, it will need to know the total number of records and files it is dealing with. These two pieces of information are held in the variables `filend` and `lastfile%`, and are stored on a separate file called `FILESPEC.DAT`, from where they are loaded each time the program is run.

The first five lines of the new program are included to overcome a minor bug in the CPC464 operating system, which can jumble the filenames of binary files when they are loaded or saved with their names held in string variables. Since this is an important part of the dynamic file mechanism used in the program, the 'fix' is essential.

The program is intended as a demonstration of the possible uses of disc-based files on the CPC464/664 micros. It is by no means a finished system. As it stands, the method of PEEKing and POKEing data is very slow. A combination of array storage, as used in the fully-programmed system of Figure 7.2, and the file storage outlined here would proably result in a system with a fair capacity and reasonable speed of data handling.

To get the most out of the system, however, a true random access database would have to be constructed. This could be done in a high-level language which directly supports random-access files, or in machine code, under CP/M. The advantage of constructing a program along these lines is that it will work on any CP/M micro.

This comes round full circle to the main advantage of a CP/M micro. CP/M has become as popular as it is today by virtue of its availability on a wide variety of 8080 and Z80-based micros. Programmers working with CP/M can be confident that, with a small amount of adaptation, their programs can be made to work on many different machines.

6128 Special

As has been stated several times in this book, the main advantage of the CPC 6128 is its extra 64K of RAM memory. This memory can be handled 'invisibly' by CP/M Plus, so that programs may use the extra workspace for data storage. It may also be used from Locomotive BASIC, however, by first loading an add-on module of utilities, supplied on disc.

This module is known as the 'Bank Manager', perhaps to inspire dread into the hearts of potential users. Actually, the extra commands, which are added to BASIC as **RSX's** (they all begin with a vertical bar, '|'), are easy to understand, and provide two simple ways of using the 64K of banked memory on the CPC 6128.

The first way of using the memory is to assign it to the screen display. This means, in effect, that you can design up to six different screen displays and switch between them, almost instantaneously, by changing banks. Very useful for certain visual effects, and likely to be used in commercial games programs in the coming years.

The second use of the bank manager, and the one more relevant to this chapter, is as a 'RAM disc'. What this means is that you can use the 64K of RAM as one big file in memory, and read, write and search for records within it. The four commands provided to do this are:

```
|BANKOPEN,record size
|BANKWRITE,return code,string to write,record number
|BANKREAD,return code,variable to take record,record number
|BANKFIND,return code,search string,start recnum,end recnum
```

|BANKOPEN divides up the memory bank into records of a certain size, which is specified as a parameter. It doesn't clear the banked memory, and is really more of a formatting command. It's even possible to change the size of the memory division within a program. This means you can handle individual fields at one level, and then, by altering the memory partitions, handle whole records at another. The maximum size of a record is 255 characters.

|BANKWRITE saves a given string, or the contents of a string variable, in the section of memory specified by the record number. The return code provides information on completion of the write operation.

|BANKREAD does the opposite of **|BANKWRITE** and reads the specified record from the banked memory into a waiting string variable, given as the second parameter to the command. Again, the return code provides additional information. It's important to note that the string variable must

be long enough to take the whole record. Locomotive BASIC normally initialises a string with zero length, and you should use a command such as:

```
input$=STRING$(20," ")
```

to give the empty string a length of 20 characters. If you don't do this, and the variable has zero length, the **|BANKREAD** command will appear not to have worked at all.

|BANKFIND searches through your memory file for a given string, starting and stopping at the record numbers you provide. If you don't give any, the search will start at the current value of the return code, and continue to the end of the file.

The return code takes the form @**r%**, where **r%** is an integer variable which you must set up with a statement such as:

```
r%=0
```

before using any of the **BANK** commands. The @ indicates the address of the r% variable to the machine-code bank managing routines. The return code normally returns the number of a record, or an indicator confirming a successful BANK operation. Note that the two return code values -2 and -3 have the following meanings:

-2 No match found
-3 Bank switching failure (hardware fault)

and not as given in the User Manual.

To show how you can use the bank manager, Figure 7.5 is a second version of the complete, memory-based database, using the facilities of the bank manager to extend the capacity of the data file. Using this technique, each of the 26 lettered files can hold up to 300 records of 216 characters (a total of 64800 bytes). This gives you a total capacity of 7800 records, enough for many small businesses. With this system. most home users will find it quite possible to keep their entire telephone list on one file.

The changes made to the program reflect the extra commands available in the 1.1 version of BASIC. The cursor is now switched on and off using the **CURSOR** command, and characters are read from the screen using the **COPYCHR$** function. This second feature saves part of the machine-code written into the start of the original address file program.

The whole program has been restructured to use the bank manager commands, and the changes are particularly noticable in the saving, loading, searching and printing routines. Both arrays have been dispensed with and the strings `in$`, `i$` and `i1$` are now used in their place. Study of these sections of the program should reveal how the bank manager can best be used for file handling.

When running the program, remember that you have to run the bank manager routines first, from a disc with **BANKMAN.BAS** and **BANKMAN.BIN** on it. Type:

`RUN"BANKMAN <RETURN>`

`RUN"ADDRESS <RETURN>` (or whatever you call the address book file)

```
10 MEMORY 41251
20 FOR x=HIMEM+1 TO HIMEM+50
30 READ in$:POKE x,VAL("&"+in$):NEXT
40 DATA cd,06,b9,f5,3e,00,cd,a5,bb,dd
50 DATA 21,32,90,06,08,7e,dd,77,00,dd
60 DATA 77,01,23,dd,23,dd,23,10,f2,f1
70 DATA cd,0c,b9,3e,fe,21,32,90,cd,a8
80 DATA bb,3e,ff,21,3a,90,cd,a8,bb,c9
90 r%=0:GOSUB 25000:MODE 1
100 recno%=0:lastrec%=0:s$=SPACE$(24)
110 |BANKOPEN,24
120 KEY DEF 5,1,0:FOR n%=13 TO 15:KEY DEF n%,1,0:NEXT
130 INK 0,2:INK 2,26:BORDER 0
140 PEN 1:PAPER 0:CLS
150 x$="ADDRESS FILE":LOCATE 14,1:FOR i=1 TO LEN(x$):POKE 41257,ASC(MID$(x$,i,1
):CALL 41252
160 PRINT CHR$(254);CHR$(10);CHR$(8);CHR$(255);CHR$(11);:NEXT
170 WINDOW 7,15,8,17
180 PRINT " Forename  Surname  DetailsAddress 1Address 2Address 3Address 4Post (
odeTelephone";
190 PEN 0:PAPER 2:WINDOW 1,40,20,25:CLS
200 PRINT "  To browse, use <SHIFT>+ cursor keys      To edit, use cursor keys a:
d <DEL>      To store finished record, use <COPY>"
210 PRINT "  To delete record, use <SHIFT>+<COPY>"+CHR$(13)+CHR$(10)+CHR$(10)+"
 f0-Sort  f1-Search  f2-Print  f3-File";
220 WINDOW 17,40,8,18:PEN 2:PAPER 0:WINDOW#1,1,40,4,4:PEN#1,3:PAPER#1,2:CLS#1:W:
NDOW#2,5,5,8,17:PEN#2,3:PAPER#2,0
230 WHILE 1: CURSOR 1,1
240 IF INKEY(0)=32 THEN IF recno%>0 THEN recno%=0:GOSUB 21000
250 IF INKEY(2)=32 THEN IF recno%<=lastrec% THEN recno%=lastrec%+1:GOSUB 21000
260 IF INKEY(8)=32 THEN IF recno%>0 THEN recno%=recno%-1:GOSUB 21000
270 IF INKEY(1)=32 THEN IF recno%<=lastrec% THEN recno%=recno%+1:GOSUB 21000
280 IF INKEY(9)=32 THEN IF recno%<=lastrec% THEN in$=s$:FOR y%=recno%*9 TO last:
ec%*9:|BANKREAD,@r%,in$,y%+9:|BANKWRITE,@r%,in$,y%:NEXT:in$=s$:FOR x%=y% TO y%+(
:|BANKWRITE,@r%,in$,x%:NEXT:lastrec%=lastrec%-1:GOSUB 21000
290 IF INKEY(15)=0 THEN flag%=1:GOSUB 1000
300 IF INKEY(13)=0 THEN flag%=0:GOSUB 1000
310 IF INKEY(14)=0 THEN GOSUB 2000
320 IF INKEY(5)=0 THEN GOSUB 3000
330 IF lastrec%>298 THEN CLS#1:LOCATE 15,1:PRINT#1,"FILE  FULL" ELSE IF INKEY(9
=0 THEN GOSUB 4000
340 in$=INKEY$:IF in$<>"" THEN GOSUB 5000
350 WEND
360 END
```

```
1000 REM To select sort/search of file
1010 PRINT#2,"123456789";
1020 CLS#1:PRINT#1,"    Select key field by number (1-9)"
1030 in$="":WHILE in$<"1" OR in$>"9":in$=UPPER$(INKEY$):WEND:i%=VAL(in$)-1
1040 CLS#2:LOCATE#2,1,i%+1:PRINT#2,CHR$(243)
1050 IF flag% THEN GOSUB 10000 ELSE GOSUB 11000
1060 CLS#1:CLS#2
1070 RETURN
2000 REM To print file as list
2010 text$="Print Document or Labels? (D/L)":GOSUB 22000
2020 IF in$="L" THEN GOSUB 13000:RETURN
2030 text$="   Set up printer and press <SPACE>    ":GOSUB 22000
2040 CLS#1:LOCATE#1,14,1:PRINT#1,"Printing file"
2050 PRINT#8,CHR$(13);"File ";file$;CHR$(10);CHR$(10)
2060 i$=s$:il$=s$:WIDTH 80
2070 FOR y%=0 TO lastrec%
2080 !BANKREAD,@r%,i$,y%*9
2090 !BANKREAD,@r%,il$,y%*9+1
2100 GOSUB 24000:PRINT#8,i$;" ";il$
2110 i$=s$:¶BANKREAD,@r%,i$,y%*9+2
2120 GOSUB 24000:IF i$>"" THEN PRINT#8,i$
2130 FOR x%=3 TO 8
2140 i$=s$:¶BANKREAD,@r%,i$,y%*9+x%
2150 GOSUB 24000:IF i$>"" THEN PRINT#8,;i$;:IF x%<8 THEN PRINT#8,;", "; ELSE PR
NT#8,CHR$(10)
2160 NEXT:NEXT
2170 CLS#1
2180 RETURN
3000 REM To load/save file from/to disc
3010 text$="Load or Save file? (L/S)":GOSUB 22000
3020 IF in$="S" THEN GOSUB 15000:RETURN
3030 text$="Save current file first? (Y/N)":GOSUB 22000
3040 IF in$="Y" THEN GOSUB 15000
3050 CLS
3060 GOSUB 14000
3070 RETURN
4000 REM To save record in memory
4010 LOCATE 1,1
4020 FOR x%=0 TO 8:in$="":n%=1:flag%=0
4030 WHILE flag%=0
4040 i$=COPYCHR$(#0):IF i$="," THEN i$=CHR$(128)
4050 IF i$=" " THEN PRINT CHR$(9);:il$=COPYCHR$(#0):PRINT CHR$(8);:IF il$=" " T
EN flag%=1:PRINT CHR$(13);CHR$(10);
4060 IF flag%=0 THEN in$=in$+i$:PRINT" ";
4070 n%=n%+1:IF n%=25 THEN flag%=1
4080 WEND
4090 in$=in$+SPACE$(24-LEN(in$))
4100 ¶BANKWRITE,@r%,in$,recno%*9+x%
4110 NEXT:IF recno%>lastrec% THEN lastrec%=recno%
4120 recno%=recno%+1:GOSUB 21000
4130 RETURN
5000 REM To edit record on screen
5010 in%=ASC(in$)
5020 IF in%=240 THEN IF VPOS(#0)>1 THEN PRINT CHR$(11);
5030 IF in%=241 THEN IF VPOS(#0)<9 THEN PRINT CHR$(10);
5040 IF in%=242 THEN IF VPOS(#0)>1 OR POS(#0)>1 THEN PRINT CHR$(8);
5050 IF in%=243 THEN IF VPOS(#0)<9 OR POS(#0)<24 THEN PRINT CHR$(9);
5060 IF in%=13 THEN IF VPOS(#0)<9 THEN PRINT CHR$(13)+CHR$(10);
5070 IF in%=127 THEN IF VPOS(#0)>1 OR POS(#0)>1 THEN PRINT CHR$(8)+CHR$(16);
5080 IF in%>31 THEN IF in%<127 THEN PRINT CHR$(in%);:IF VPOS(#0)>9 THEN LOCATE
,1
5090 RETURN
```

```
10000 REM To sort file in memory
10010 CLS#1:LOCATE#1,14,1:PRINT#1,"Sorting file"
10020 i$=s$:il$=s$
10030 FOR pntr1%=0 TO lastrec%-1
10040 FOR pntr2%=pntr1%+1 TO lastrec%
10050 ¦BANKREAD,@r%,i$,pntr2%*9+i%
10060 ¦BANKREAD,@r%,il$,pntr1%*9+i%
10070 IF i$<il$ THEN FOR n%=0 TO 8:¦BANKREAD,@r%,i$,pntr2%*9+n%:¦BANKREAD,@r%,i
$,pntr1%*9+n%:¦BANKWRITE,@r%,i$,pntr1%*9+n%:¦BANKWRITE,@r%,il$,pntr2%*9+n%:NEXT
10080 NEXT:NEXT
10090 recno%=0:GOSUB 21000
10100 RETURN
11000 REM To search file and display
11010 CLS#1:PRINT#1," Search for? >                              <"
11020 search$="":LOCATE#1,15,1
11030 WHILE LEN(search$)<25 AND INKEY(18)<>0
11040 in$=INKEY$
11050 IF in$=CHR$(127) THEN IF search$>"" THEN search$=LEFT$(search$,LEN(search.
)-1):PRINT#1,CHR$(8);CHR$(16);
11060 IF in$>=" " AND in$<"(" THEN search$=search$+in$:PRINT#1,in$;
11070 WEND
11080 text$="Display or Store in file '@'? (D/S)":GOSUB 22000
11090 IF in$="S" THEN GOSUB 12000:RETURN
11100 in$=s$:recno%=0:m%=-1
11110 WHILE recno%<=lastrec%
11120 ¦BANKREAD,@r%,in$,recno%*9+i%
11130 IF search$=LEFT$(in$,LEN(search$)) THEN IF m%=-1 THEN GOSUB 21000:m%=recn
% ELSE text$="     MORE - press <SPACE>     ":GOSUB 22000:GOSUB 21000:m%=recno%
11140 recno%=recno%+1
11150 WEND
11160 IF m%=-1 THEN recno%=0:text$="      No match found - press <SPACE>     ":GO
UB 22000:GOSUB 21000 ELSE recno%=m%
11170 RETURN
12000 REM To search file and store
12010 CLS#1:LOCATE#1,13,1:PRINT#1,"Searching file"
12020 OPENOUT "@"
12030 m%=0:in$=s$
12040 FOR y%=0 TO lastrec%
12050 ¦BANKREAD,@r%,in$,y%*9+i%
12060 IF search$=LEFT$(in$,LEN(search$)) THEN FOR x%=0 TO 8:¦BANKREAD,@r%,in$,y
*9+x%:PRINT#9,in$:NEXT:m%=m%+1
12070 NEXT
12080 CLOSEOUT
12090 text$="   Matches: "+STR$(m%)+" found - press <SPACE>    ":GOSUB 22000
12100 RETURN
13000 REM To print file as labels
13010 text$="Whole file or Selected records? (W/S)":GOSUB 22000
13020 IF in$="W" THEN text$="   Set up printer and press <SPACE>     ":GOSUB 220
0:FOR y%=0 TO lastrec%:GOSUB 23000:NEXT:CLS#1:RETURN
13030 CLS#1:PRINT#1,"<SHIFT>+";CHR$(242);"&";CHR$(243);"Select <COPY>Print <ENT
R>End";
13040 WHILE INKEY(6)<>0
13050 IF INKEY(8)=32 THEN IF recno%>0 THEN recno%=recno%-1:GOSUB 21000
13060 IF INKEY(1)=32 THEN IF recno%<lastrec% THEN recno%=recno%+1:GOSUB 21000
13070 IF INKEY(9)=0 THEN y%=recno%:GOSUB 23000
13080 WEND:CLS#1:LOCATE 1,1
13090 RETURN
```

```
14000 REM To load file from disc
14010 GOSUB 25000:text$="LOAD":GOSUB 20000
14020 CLS#1:LOCATE#1,13,1:PRINT#1,"Loading file ";in$:file$=in$
14030 OPENIN in$
14040 y%=0:WHILE NOT EOF
14050 FOR x%=0 TO 8
14060 INPUT#9,in$:in$=in$+SPACE$(24-LEN(in$))
14070 |BANKWRITE,@r%,in$,y%*9+x%
14080 NEXT:y%=y%+1
14090 WEND:lastrec%=y%-1
14100 CLOSEIN:CLS#1
14110 recno%=0:GOSUB 21000
14120 RETURN
15000 REM To save file to disc
15010 text$="SAVE":GOSUB 20000
15020 CLS#1:LOCATE#1,13,1:PRINT#1,"Saving file  ";in$
15030 OPENOUT in$
15040 FOR y%=0 TO lastrec%
15050 FOR x%=0 TO 8
15060 in$=s$:|BANKREAD,@r%,in$,y%*9+x%:PRINT#9,in$
15070 NEXT:NEXT
15080 CLOSEOUT:CLS#1
15090 RETURN
20000 REM To read filename from keyboard
20010 CLS#1:LOCATE#1,7,1:PRINT#1,"File to ";text$;"? (A to Z or @)"
20020 in$="":WHILE in$<"@" OR in$>"Z":in$=UPPER$(INKEY$):WEND
20030 RETURN
21000 REM To display record on screen
21010 CLS:CURSOR 0:in$=s$
21020 FOR x%=0 TO 8
21030 |BANKREAD,@r%,in$,recno%*9+x%
21040 LOCATE 1,x%+1:PRINT in$
21050 NEXT:LOCATE 1,1
21060 RETURN
22000 REM To ask Yes/No question
22010 CLS#1:LOCATE#1,(40-LEN(text$))/2,1
22020 PRINT#1,text$:in$=""
22030 WHILE in$<>LEFT$(RIGHT$(text$,4),1) AND in$<>LEFT$(RIGHT$(text$,2),1):in$=
UPPER$(INKEY$):WEND
22040 RETURN
23000 REM To print a label
23010 i$=s$:il$=s$
23020 |BANKREAD,@r%,i$,y%*9
23030 |BANKREAD,@r%,il$,y%*9+1
23040 GOSUB 24000:PRINT#8,i$;" ";il$
23050 i%=1:FOR x%=2 TO 7:i$=s$
23060 |BANKREAD,@r%,i$,y%*9+x%:GOSUB 24000
23070 IF i$>"" THEN PRINT#8,i$:i%=i%+1
23080 NEXT:FOR x%=1 TO 9-i%:PRINT#8:NEXT
23090 RETURN
24000 REM To strip spaces from a string
24010 WHILE RIGHT$(i$,1)=" "
24020 i$=LEFT$(i$,LEN(i$)-1)
24030 WEND
24040 RETURN
25000 REM To clear memory bank
25010 CLS#1:PRINT#1,"   Clearing memory bank - Please wait"
25020 |BANKOPEN,255:x$=SPACE$(216)
25030 FOR y%=0 TO 255
25040 |BANKWRITE,@r%,x$,y%
25050 NEXT
25060 RETURN
```

Figure 7.5 Listing of banked RAM database program

Round Up

For several years now the computer pundits have predicted the imminent death of CP/M, in favour of 16 bit operating systems such as MS-DOS, which is favoured by IBM. Although it's true that the majority of 'serious' business micro makers have moved on from the CP/M 8 bit standard, CP/M has been rejuvenated by the continued interest shown in it by popular and home computer manufacturers, such as Amstrad. Digital Research have been quick to realise the new market potential for dear old CP/M, and by releasing it relatively cheaply to the mass computer market, have assured its continued existence for a few years yet.

Despite its initially 'unfriendly' appearance, CP/M is still a very workable operating system, which contains many useful controls squeezed into a small amount of memory and disc space. There is also a certain 'elegance' in CP/M's economy of language. In the end, it's likely to be an operating system you learn to love or to hate. If you love it, there's plenty of scope for using its facilities. If you don't, there are many sources or ready-made programs written by those who do. With CP/M you have the choice, and can make of it what you will.

GLOSSARY

Absolute The exact reference of a cell in a spreadsheet.

Address The reference number of a byte in memory.

AMSDOS The AMSTRAD Disc Operating System. A set of commands used to control disc operation.

ASCII The American Standard Code for Information Interchange. A system of coding letters and numbers to make them more easily understood by a computer.

ASM The machine code assembler supplied by Digital Research to work with CP/M.

Assembler A programming aid which converts a series of mnemonic instructions into machine code.

Bdos The basic disc operating system. A part of CP/M.

BOOLEAN A general term describing a system of logic developed by Rev Boole. Includes terms such as AND, OR, NOT and XOR.

BOOTGEN A CP/M utility program which will copy the tracks on a CP/M disc which control what happens when the disc is started up.

Byte A unit of computer memory. It can be thought of as containing one character (letter or number).

C A computer language often used for writing business and 'system' programs.

Catalogue The section of a disc which holds details of all the files held on it.

CCP The Console Command Processor. The part of CP/M which handles commands typed in at the keyboard (or entered from a SUBMIT file).

Cell — One of many small areas making up a spreadsheet. Each can contain a word or number, entered by the user of the program.

CHKDISC — An AMSDOS program which compares the contents of two discs on a single-disc system.

CLOAD — An AMSDOS routine which loads programs from cassette on a disc-based machine.

Compiler — A program which converts commands in a high-level language into machine-code. The machine code can then be run without further translation.

Control code — A code produced by pressing a letter or number key, together with the <CTRL> key. Usually used to perform a special function in a program.

Co-ordinates — The row and column numbers referencing a cell in a spreadsheet.

COPYDISC — An AMSDOS program which copies the contents of one disc onto another, on a single-disc system.

| CPM — An AMSDOS command which starts CP/M.

CP/M — A popular operating system for microcomputers.

CSAVE — An AMSDOS routine which saves programs to cassette on a disc-based machine.

Database — An organised collection of information held on a computer.

DDT — A CP/M utility which aids the debugging of machine code programs.

DIR — A CP/M command which reads the directory of a disc and displays it on the monitor screen.

| DIR — An AMSDOS command which does the same as 'DIR' in CP/M.

directory — (see catalogue)

DISCCHK — An AMSDOS program which compares two discs on a twin-disc system.

DISCCOPY An AMSDOS program which copies the contents of one disc onto another, on a twin-disc system.

Disc interface The electronic connection between a computer and a disc drive (or drives).

DUMP A CP/M program which displays the contents of an area of memory to the monitor screen.

ED The text editor supplied by Digital Research to work with CP/M.

Editor A programming aid which speeds the entry of text into a computer.

ERA A CP/M command which erases a named file from a disc.

\| ERA The AMSDOS equivalent of ERA.

File control block An area of CP/M in memory, which holds details of a file on a disc.

Field A defined area of the display from an applications program, where data may be entered by the user of the program.

File An organised set of characters, stored on a disc. It might be the text from a word a processor, or the information in a database, for example.

FILECOPY An AMSDOS program which copies the contents of a file from one disc to another.

Filename The name given to a file stored on disc.

Filetype A three character extension to the filename of a CP/M file, which provides information about the type of data held in the file.

Footer The repeated section at the bottom of a page of a printed document. It often contains the page number.

FORMAT An AMSDOS program which records a set of 'markers' on a disc, so the disc operating system can *find its way about.*

Format The process of rearranging the text in a document.

Header The repeated section at the top of a page of a printed document.

Hexadecimal A number system based on the number 16, as opposed to decimal, which is based on 10.

Interpreter A program which converts commands in a high level language into machine code. It does this one command at a time, as the program is running.

Justification The process of lining text up against the margin of the paper.

'K' A unit of computer memory, representing approximately 1000 bytes (actually 1024 bytes).

Key field A field in a database record, designated by the user of the program, for sorting the file into a specific order.

List processing The process of handling data represented as a series of 'lists'.

Logo A computer language designed as a teaching aid for mathematics and computer programming. Is also a good 'list processing' language.

Macro A programming aid offered by some machine code assemblers, which allows sections of program to be represented by a meaningful name.

Merging files The process of combining text from two or more documents. Often used to prepare 'customised' business letters.

Monitor (TV) A specially designed television, suitable for the display of computer data.

Monitor (m/c) A programming aid which can run a machine code program under the control of the user of the program.

MOVCPM A CP/M utility which will move the part of the operating system held in memory, to allow it two work in computers with less than 64K of memory.

Object program The machine code program produced from a source program by an assembler.

Parent-child A relationship set up in a database between two or more records, allowing the data from one to be associated with the other.

Pascal A fast, compiled computer language designed to foster the idea of 'structured' programming.

PIP A CP/M utility which specialises in copying files from one peripheral to another. The Peripheral Interchange program.

Prompt A symbol or short message displayed by a computer program to indicate a required entry.

Random access The ability to read any record on a disc at random, without having to read all the records before it in a file.

Record A section of a database file which contains information on a particular subject.

Recursion The ability of a computer language to y of a database program to associate records within a file, perhaps in a 'parent-child' relationship.

Relative The reference of a cell in a spreadsheet in relation to the position of another, specified cell.

REN A CP/M command which will rename a file on disc.

:REN The AMSDOS equivalent of 'REN'.

Replicate To copy a cell, or range of cells, from one position in a speadsheet to another.

Resident program A CP/M program which is held in memory all the time CP/M is in use, eg. DIR, ERA and REN.

Search The ability of a database program to hunt through a file for a specified piece, or pieces, of information.

Search and replace A routine in some text editors and word processors, which will hunt through the text and replace all occurrences of one word or phrase with another.

Sequential A database file in which all records have to be read in turn.

SETUP An Amstrad utility program which allows the user

of the program to alter certain features of the computer, such as the colour of the display.

Sort — The ability of a database program to re-order the information in a file according to a rule supplied by the user of the program.

Source program — The text of an assembler or high-level language program, before it is converted into machine-code.

Spreadsheet — A computerised 'sheet of squared paper' onto which numbers can be entered and manipulated.

STAT — A CP/M utility which displays status information about a disc or the files on it.

SUBMIT — A CP/M utility which allows a set of CP/M commands to be stored in, and executed from, a disc file.

SYSGEN — A CP/M program which copies the section of a disc holding CP/M to another disc.

System disc — A disc containing the CP/M operating system.

Transient program — A CP/M program loaded into memory only when specifically called for. E.G. ASM, DDT and ED.

TYPE — A CP/M program which displays a text file on the monitor screen.

Wildcards — Characters which may be used in AMSDOS and CP/M disc commands to substitute one or more alphanumeric characters. CP/M uses '?' for a single character, and '*' for any number of characters.

Word processor — An application program which enables the user of the program to enter text into the computer, edit it, store it on disc and print it out.

XSUB — A CP/M utility which allows a set of application program commands to be stored in, and executed from, a disc file.

Z80 — The microprocessor used in the Amstrad CPC micros. CP/M is designed to work with this family of processors.

BIBLIOGRAPHY

Chapter 1

DDI-1 or CPC664 manual, Amsoft, 1984/5

Chapter 2

CP/M The Software Bus.
Clark, Eaton and Powys Lybbe,
Sigma Technical Press, 1983
CP/M User Guide, Digital Research

Chapter 3

The CP/M Software Library, CP/MUGUK, updated regularly

Chapter 4

Programming the Z80
Zaks R,
Sibex, 1982

Chapter 5

Logo for Micros.
Martin Lesser
Newnes, 1985
Simple Pascal.
McGregor and Watt,
Pitman, 1982
Introducing Pascal.
Allen B.
Granada, 1984.
Hisoft Guide to Pascal,
Link D.
Amsoft, 1985

Chapter 6

The Writer and the Word Processor,
Hammond R.
Coronet 1984.

Chapter 7

Powerful Programming for Amstrads,
Johnson W.
Sigma Press, 1986.

For details of the CP/M user group write to:
The Secretary,
CP/M User Group UK,
72 Mill Road,
Hawley, Dartford,
Kent DA2 7RZ.

INDEX

The Amstrad Disc Companion

About this book

Amstrad users are fortunate in being able to use the power of disc drives, without the usual high price tag. Owners of the CPC-464 can purchase a disc unit for a small extra cost, whilst on the CPC-6128 (and earlier CPC-664) the disc drive is included neatly within the main body of the micro.

Whether you use your Amstrad for business or pleasure, this book will provide both an introduction and an in-depth reference guide. You'll see how to use the essential software packages for disc users - word processors, spreadsheets and databases. After mastering these, you can progress to writing your own programs using BASIC, LOGO, Pascal or C. Separate sections describe these languages and their important disc-related features. You'll even find complete and useful programs ready to run on your Amstrad.

Finally, all aspects of CP/M and AMSDOS are covered in great depth. Every command that you'll need to use is carefully explained, giving you valuable insights and a reference manual, all in one superb book.

About Us

We produce quality books for computer users. Ask for our catalogue or tell us about the book that you'd like to write.

Sigma Press
98a Water Lane
Wilmslow
Cheshire SK9 5BB

GB £ NET +007.95

ISBN 1-85058-034-0

00795

9 781850 580348

SIGMA PRESS

The Amstrad Disc Companion

Simon Williams

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/