AMSTRAD

AN INTRODUCTION TO CP/M PLUS ON AMSTRAD COMPUTERS

P. K. McBride

GLENTOP

AN INTRODUCTION TO CP/M PLUS ON AMSTRAD COMPUTERS

INCLUDES CP/M 2.2

AN INTRODUCTION TO CP/M PLUS ON AMSTRAD COMPUTERS

INCLUDES CP/M 2.2

P. K. McBride

GLENTOP PUBLISHERS LTD

First published 1986

ISBN

1 85181 054 4

Printed in Great Britain by The Eastern Press Ltd., Reading

A Newtech production for
Glentop Publishers Ltd
Standfast House, Bath Place, Barnet, Herts EN5 5XE

CONTENTS

ILLUSTRATIONS

INTRODUCTION

Do you really need an introduction to CP/M? Haven't you already met it when formatting your disks? Perhaps you have also been running a spreadsheet or other business software package under CP/M. But have you done anything else with it? And have you thought about what it was actually doing, or what else you could do with it? This book will try to answer those questions.

What is CP/M?

CP/M either stands for Control Program for Microprocessors or Control Program and Monitor – it depends on which authority you believe. Digital Research, who market the system, don't seem to care. They always refer to it simply as CP/M and so shall we.

CP/M is an operating system, a link between the hardware itself and any applications software – languages, word processors and business programs, games or whatever. It handles the interactions between the Central Processing Unit and all peripheral devices, which include the screen, keyboard, printer and disk drive. This latter is a key component, because CP/M was developed to handle the disk drive when it first appeared.

Dr Gary Kildall was working for the Intel Corporation when he originally designed CP/M as an operating system to support a high-level language compiler to run on the new 8080 chip, back in 1974. One major aspect of CP/M has been present since the beginning. Its command set is divided into resident and transient parts. The resident commands are loaded into memory when the system is 'booted' – brought into operation; they include the most commonly used ones. Transient commands are loaded in from disk as and when needed. This makes for great flexibility as further commands can be added at any time, but more importantly, it reduces to a minimum the amount of memory needed for the operating system. This was even more important in 1974 than it is today, as at that time memory was expensive and 8K was a standard size.

CP/M was good even then, but might have remained just another machine-specific system if Kildall had stayed with Intel. As it was, Intel decided to concentrate on the hardware side, marketing their hugely successful chips, and the software division was disbanded. Gary Kildall went to teach at the Monterey Naval Postgraduate School, where he continued to develop CP/M. In 1976 he was asked by Imsai, the floppy disk drive manufacturers, to design an operating system to work with their equipment.

Rather than start from scratch with a new machine-specific system, he decided to adapt CP/M so that programs written under it would work with their hardware as well as with the original Intel equipment. He divided up the FDOS (Full Disk Operating System) into two parts: the BDOS (Basic Disk Operating System) and the BIOS (Basic Input/Output System). (Basic here means elemental, and has no reference to the language.)

The BIOS is a set of peripheral drivers – routines that handle the interactions with input and output devices. It is therefore very largely machine-specific, but each implementation can perform the same range of functions. The BDOS controls the way in which data is stored and accessed on the disks, and is much the same in all implementations. Variations are mainly in terms of disk capacity and organisation. As a result, any program written using CP/M will run on any machine that has CP/M installed. All input and output – disk reading and writing, printing on screen and paper, keyboard input – is directed through the standard BDOS calls, which then draw on the appropriate BDOS and BIOS routines. Implementing CP/M on a new machine is therefore largely a matter of rewriting the BIOS calls.

Kildall's vision had a major impact on the development of the software market, in particular business software. Because the same program could be run on a range of machines, companies could upgrade their hardware without the additional expense of buying new software and without the trouble of transferring all their data; software writers were freed of the need to learn the peculiarities of each new system; and computer manufacturers could launch a new machine knowing that, as long as they had arranged for CP/M to be implemented, there would be plenty of software available immediately.

In the ten years since Gary Kildall founded Digital Research to market CP/M and to exploit its potential, the system has been implemented on nearly one hundred different micros, and its users number countless hundreds of thousands.

CP/M has developed over that time – though each new version has been upwardly compatible with the previous. Initially it was available only on machines that used the Intel 8080 chip, but was then extended to cover others in that family, and the Z80 that lies at the heart of your Amstrad. More recently, CP/M has also been implemented on the new 16-bit chips, the Intel 8086, Motorola 68000 and the Zilog Z8000. The first publicly available version of the system (Version 1.4) had fairly restricted functions. These were enhanced in version 2.2 – the one supplied with the 664 and the 464's disk drive. Neither of these can support graphics. This facility was not felt necessary at first, nor could all of the earlier computers handle graphics. More recent machines have the memory capacity to map detailed screens and the latest version of the system – CP/M Plus – has greatly enhanced capabilities, not the least of which is the ability to handle graphics through the associated GSX system.

If software is to run under CP/M it must be written in assembly language or in a compilable language such as Pascal, C or Cbasic. Either way, it must finish up as a block of code. The second requirement is obvious, but worth restating; the programmer must direct all inputs and outputs through the FDOS. This can produce a time overhead. A machine-code program written under CP/M will not be as fast as pure, machine-specific code, though any compiled program will run faster than one in Basic. This time cost is a small disadvantage that is more than offset by the many advantages of the system.

What CP/M offers to its users

First and foremost, CP/M offers access to a massive bank of software that has been developed by commercial software houses, university and other research departments and by private enthusiasts over a period of ten years. Much of this is in the public domain, and freely available through the CP/M User Group. (There is a charge for copying, but not for copyright.) The public domain software mainly comes from the United States, where research students are required to make copies of their programs public. Some has been donated by software houses – either to encourage interest in other commercial products, or because the programs that they had developed had no further commercial use. Either way, the public benefits. The range of public domain software includes assemblers, implementations of most computer languages and other development tools for programmers, as well as utilities for scientific and other research, word processors and some business programs and non-graphic games. Most of these can be obtained in 3-inch format through the UK CP/M User Group. The quality of the programs vary, which is to be expected, but some of them are very good.

Commercial software in 3-inch format is rather more restricted in scope at the time of writing, but can be expected to increase steadily. There is plenty of choice of business software, including graphic applications for CP/M Plus versions, and a good variety of languages and programming tools. Many of the programs have been around, in one form or another, for several years – in itself a guarantee of quality. Given the competition within the market, and from public domain software, a program has to be good to survive, and must evolve constantly to retain its place.

A word of warning may be appropriate here. Both CP/M Version 2.2 and CP/M Plus can support a hard disk unit and multiple floppy disk drives. Some software may only work properly with a minimum of two drives. This shouldn't matter to Joyce owners, who can use part of the memory as a RAM disk, but could prove a problem to those with 464, 664 or 6128 machines. Do check that programs will run on your own configuration of equipment as well as being suitable for your version of CP/M.

What CP/M offers to programmers

Portability of skills and of programs are the main advantages offered by CP/M to programmers. If you will only ever want to write for your present machine, then there is no great point – apart from the interest and challenge of handling a new system – in working with CP/M. Locomotive and Mallard Basics are both good, efficient languages, and the machine-code programmer has more flexibility within the native systems – particularly so on the 464, 664 and 6128 machines where the firmware is fully documented and easily accessed.

The professional and the enthusiastic amateur programmer on the other hand should welcome the chance to work in a standard system, rejoicing in the fact that all the effort that has gone into creating effective programs will not go to waste in a few short years as the hardware becomes obsolete.

Working under CP/M does not mean that you have to learn new languages, or even that you have to learn the intricacies of the CP/M system. The Basic programmer can transfer to a compilable Basic such as Digital Research's Cbasic. There are some variations in the language, inevitably, and to get the best out of it, you will need to spend some time getting to grips with the way that it works, but you will have programs up and running the same day that you get the package. We will have a look later in the book at programming in Cbasic, and in the other popular language, Pascal.

The machine-code programmer will find that a little more effort is needed to transfer to CP/M, mainly because the firmware calls must be replaced by calls to the BDOS. There is also a problem in that the assemblers and debuggers supplied with the system are based on the 8080, not the Z80 chip, and use 8080 mnemonics. Although they are different from Z80 mnemonics, they are logical and follow an easily learnt pattern – the table in Appendix A should help with conversion. The main problem with 8080 assembly is that some of the Z80's finer features are simply not present. It's a problem you must live with if you rely on ASM and DDT (or RMAC, SID and the rest in the Plus suite) for your machine-code work. If you use a full Z80 assembler, such as the one in Hisoft's DEVPAC, then you can continue to write normal Z80 code and keep your new learning to a minimum.

We will return to programming later, but first, let us look closer at the CP/M operating system, and the resident command set.

CHAPTER 1

USING CP/M

In this chapter we will look at the resident commands and the ways in which they can be used for efficient file management. You don't need an in-depth understanding of how CP/M works before you start to use these commands, but it is useful to know how it organises your computer's memory and how it structures file storage on disks. This, then, is where we will start.

Memory organisation under CP/M

The way in which CP/M organises the memory varies between machines, and the simplest Amstrad system is probably that in the 464/664. As this shows all the essential principles, it is a good place to start.

Under the normal Basic Operating System, the memory map looks like Figure 1.2. The top 16K holds the screen, and immediately below this is a 5K area used by the system for the stack, system variables and the firmware jumpblock. The stack is used for storing return addresses, loop control counters and addresses and other temporary working data. The system variables hold such things as current colour values, mode numbers, cursor position and other status indicators. The jumpblock has the addresses of the firmware routines, of which more later.

Just below this lies the area occupied by AMSDOS – the Amstrad Disk Operating System – while at the bottom of memory – the first 64 bytes only – is a small area also reserved by AMSDOS. In between these is something over 42K of free space.

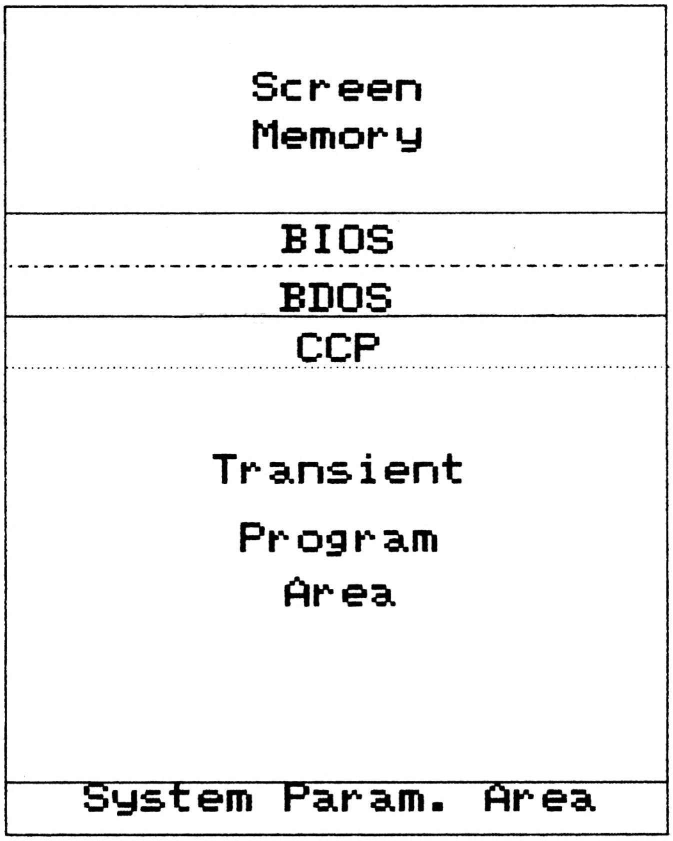


Figure 1.1 *CP/M 2.2 memory map*

The great bulk of the system resides in ROM and is accessed at need by bank-switching. The Basic language is in the 16K upper ROM that lies parallel to the screen memory; and the firmware routines occupy the 16K lower ROM at the bottom of memory. Bank-switching allows these ROMs to be read, in preference to the RAM memory, when they are needed. When you are writing in Basic, you are simply not aware of this, which is a sign of an efficient operating system. The machine-code programmer needs to know more about the firmware routines, as these offer the simplest means of handling inputs and outputs and are the only safe way to change systems variables, but accessing them poses no problems. The address of each routine in the lower ROM is stored in the jumpblock in RAM memory. Calls to this jumpblock are redirected to the firmware ROM, with the system handling the bank-switching automatically.

When CP/M 2.2 is in operation, the memory map is changed, but follows a similar pattern. The top 16K block is still used for screen memory. Below that is the permanent part of CP/M, the FDOS (Full Disk Operating System) divided into BIOS (Basic

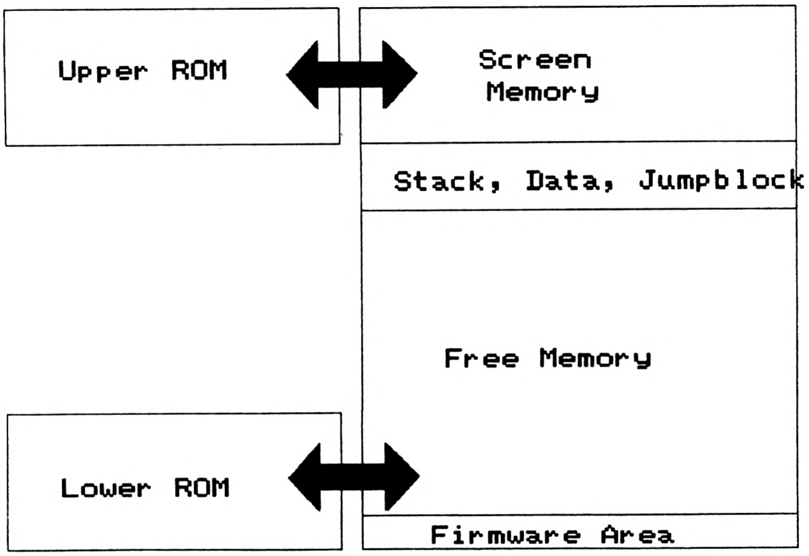


Figure 1.2 *AMSDOS memory map*

Input/Output System) and BDOS (Basic Disk Operating System). The bootstrap loader, used for loading in the resident commands and for restarting the system, is also slotted in up here. The resident commands themselves are grouped with some control routines into a block known as the Console Command Processor (CCP) which lies at the top of the free memory space or Transient Program Area (TPA). Transient commands and applications programs which use the TPA are allowed to overwrite the CCP, which is reloaded when control returns to the system.

CP/M also reserves a 256-byte block at the bottom of memory, the System Parameter Area. This is mainly used as buffer space for the temporary storage of command lines and file control data, but it also has two other vital attributes. A call to address 0000 will reboot the system, loading in the CCP, resetting the disk drives and restoring other initial values. Secondly, all access to the FDOS functions is directed through a call to address 0005, of which more later.

CP/M Plus

Although CP/M Plus cannot address more than 64K at any time, no more than the 2.2 version can, it is able to make full use of the additional memory of the PCWs and 6128 models by extensive bank-switching.

Memory is arranged in 16K blocks. The top block contains the permanent parts of the BDOS and BIOS, any RSX routines and some of the TPA. This block is active at all times. The 48K below this is switched between TPA in Bank 1 and the operating system, mainly in Bank 0. The highest of the blocks in Bank 0 contains additional BDOS

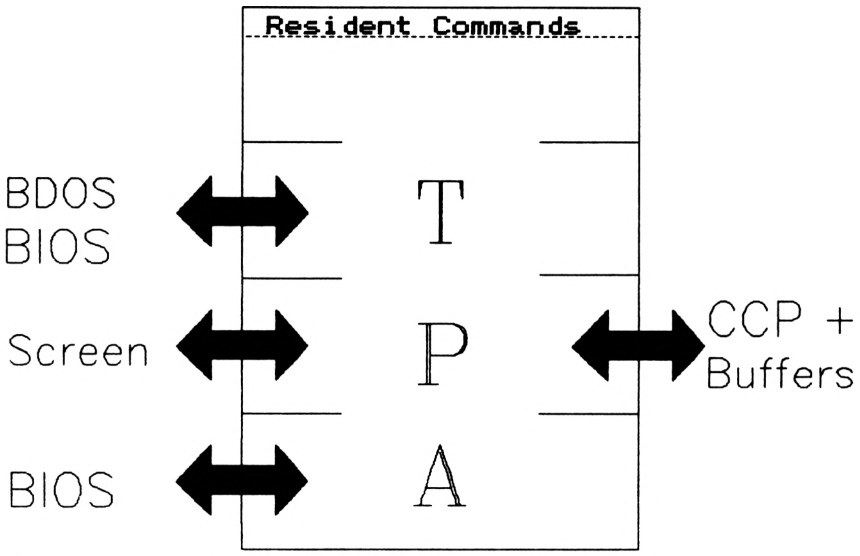


Figure 1.3 *CP/M Plus memory map*

and BIOS routines. The second block maps the screen, but is also switchable with a block in which the CCP and data buffers are held, and the lower block holds further BIOS routines. There are some variations between the machines, and the most significant of these is probably that a large part of the 8256's and 8512's additional memory is treated as a RAM disk.

In CP/M Plus, the size of the Transient Program Area depends upon the amount of memory taken up by RSX routines. These Resident System eXtensions will normally be routines from GSX, the Graphics System eXtensions available under CP/M, but can also include those written by the user or built into applications software.

Disk storage

If you have made any use of your disk drive, you will be aware that before you can save anything on a disk it must be formatted. This process divides up the raw magnetic surface into sectors each capable of holding bytes of data. Part of the formatted disk is also marked off for use as a directory of the files on the disk. For each file, the directory holds details of its name, attributes, date stamps and password (in CP/M Plus only) and of whereabouts on the disk it is stored. The file itself is not stored as a single block, but rather as a collection of records, one in each sector. If the file has extra data added to it, then more sectors are allocated; likewise the sectors are freed for reuse if the file is edited down in size. This dynamic allocation of disk space means that the directory must keep track of the sectors used by a file, but more importantly, it means that very little space is wasted on the disk.

FILESPECS (specifications) must conform to certain conventions. The filespec consists essentially of two parts, a name and a type. The name can be up to eight letters long, and can be any combination of letters and digits. The case of letters is irrelevant, as filenames are automatically forced into upper case by the system.

Some symbols may also be used, but not these:

```
* , . : ; < = > ? [ ]
```

If the name you give is less than eight letters long, it will be padded out with spaces in the directory.

The second part of the filespec is the file type. This is a three-letter extension to the name which can be used to identify the nature of the file. So a file ending in .COM is treated as a COMmand file by the system; .DAT would indicate that the file contained data. We will return to these later when we examine a directory with the DIR command.

Every filespec on a disk must be unique. You can have several files of the same name, but only as long as they are of different types. The whole name should also be meaningful to you, because otherwise you are bound to forget the difference between 'CODE471.ASM' and 'CODE482.ASM'.

Master disks

DO NOT USE THE MASTER DISKS either of your CP/M system or of any other utility. There is always a danger of corruption or loss of files through user error or other accidents. Always make a working copy then store the master disk out of harm's way. Blank disks are far cheaper than replacement copies of commercial software.

If you have not already done so, make a working copy now. This should be the only time you use the master. Set the write-protect notch on the master before you start. If you have a single drive, copying a disk will mean that you have to switch disks several times. Always wait for the drive motor to stop before removing a disk.

CP/M 2.2 users

Switch on, insert the master disk and type |cpm. Prepare a blank disk for use by formatting it:

```
A> FORMAT
```

Follow the instructions, and at the end, call up the program to copy the disk. You should use DISCCOPY if you have only one drive, or COPYDISC if you have two:

```
A> DISCCOPY
```

Follow the instructions again, then label the working copy and store the master.

CP/M Plus users

Switch on, insert the first system disk with Side 1 uppermost (or facing the screen in a PCW drive), and type |cpm. The DISCKIT utility will handle all the copying. If the blank disk has not been formatted, this will be taken care of automatically before the program starts to copy onto it. Start it up with

```
A> DISCKIT (or DISCKIT3)
```

Follow the instructions, making sure that the right disk is in the drive at the right time if you have only one drive. Repeat the process with the other three sides of the system disks.

CCP – the human interface

The Console Command Processor (CCP) is the user's link with the routines of the operating system. It is active whenever you see the system prompt. This is normally

```
A>
```

but can also be either

```
B>
```

or

```
M>
```

when you have logged in to either a second disk drive or the RAM disk.

In theory, CP/M can access up to 16 disk drives, labelled A to P, but in practice few Amstrad owners will use more than A, B and M. Most 464 and 664 owners will in fact only use drive A. Those with the 6128 and using CP/M Plus are able to treat their single drive as if it were two. It may be addressed as either A: or B: and for most purposes is as good as having a second drive.

Those with 8512s do, of course, have their second high-capacity drive, and both they and the 8256 owners are also able to use part of memory as a RAM disk (addressed as M:). This book will treat the second drive as a convenience, rather than as an essential part of the configuration.

The CCP responds to six commands, and to single characters to switch drives:

```
A> B:[ENTER]

B>
```

A single letter, followed by a colon and [ENTER], will log you into another drive. The colon shows that you mean the drive called B, not a file called B. Any programs or data files that are then called up will be taken from this drive unless you specify otherwise.

The resident commands

These are present in both the 2.2 and CP/M Plus version, though in the latter they have extra facilities, some of which are called up through associated transient commands. You must have the files DIR, ERASE, RENAME and TYPE on a disk if you want to use them to the full, and SAVE, which is a resident command in 2.2, is a transient in CP/M Plus. The Plus enhancements will be covered at the end of the common material for each command.

Note that all CP/M commands, and filenames, can be typed in either upper- or lower-case letters. The CCP automatically converts them to upper case. ENTER must always be pressed at the end of each command line. Typing errors can be corrected with DELETE or CTRL and H – hold down CTRL and press H. If the whole line is wrong, or the error is near the beginning, then either erase the lot with CTRL and X, or just press ENTER. The CCP will echo back anything it doesn't understand:

```
A> DOR [ENTER]
DOR?
A>
```

DIR – disk directory

This reads the disk's internal directory and displays the information on the screen. It is the CP/M equivalent of Basic's 'CAT' command, though far more flexible. The output will normally be to the screen, but it can be redirected to the printer by pressing CTRL and P when the prompt is waiting for a command to be input.

The simple DIR will give you a directory of the disk in the current drive:

```
A> DIR [ENTER]
A:ED      COM: ASM        COM: ZAPPO  ASM: LOAD   COM
A:ZAPPO   HEX : ZAPPO     PRN : ZOOM  ASM: DDT    COM
A:ZAPPO   COM: COMMENTS   TXT : READ  ME  : NOTES
```

This is a fairly typical example of a disk that is mainly being used for assembler work. The programmer would appear to be working on two pieces of code at the moment, ZAPPO and ZOOM. He clearly appreciates the speed advantage of machine-code programs.

You should notice two things about the directory. Firstly, it is not given in alphabetical order; instead filenames are displayed in the order in which they are stored in the disk's internal directory. Secondly, for each filename the system displays two parts – the name itself and the type extension. In this example, a COM file identifies a program that can be run under CP/M; ASM identifies the source file of a program that is to be assembled by the ASM.COM utility. TXT has no particular meaning to CP/M, but reminds the programmer that it is a text file; and in READ.ME the type specification ME is simply used to make the whole name more visible. NOTES has no type extension, because the programmer decided he didn't need one.

Where files are being created for processing by another program, that program may require a particular type name to be used. In other circumstances, the extension is optional. A table of standard typenames is given in Figure 1.4.

A third element in the file specification is the drive identifier. In a disk directory it is shown at the start of each line only:

```
A:ED     COM: ASM     COM:....
```

In some uses this disk letter is very relevant.

Where there is a RAM disk, or a second drive attached, the drive name can be added after the DIR command. (Don't forget that with the 6128 you can treat the disk drive as if it were two separate ones called A: and B:.) A colon after the letter identifies it as a drive name rather than a single-letter filename:

```
A> DIR B:
```

is equivalent to, and more convenient than:

```
A> B:
B> DIR
```

DIR can also be used to search a disk for a particular file, or set of files. The name and type must be given after DIR. If the system can find no match for the filespec in the disk's directory, then 'NO FILE' will be displayed:

```
A> DIR ZAPPO.ASM
ZAPPO ASM
A> DIR BINGO.TXT
NO FILE
```

Where there are two or more drives, the file search can be directed to a particular drive by including the drive identifier in the specification:

```
A> DIR B:MEMO.TXT
```

This will search drive B for the text file MEMO.TXT.

Wildcards are an important part of all file-handling commands. They are characters which can stand for any others, in the same way that jokers, or wildcards, can be substituted for other playing cards in some games.

A question mark ? replaces a single character.
An asterisk * replaces a group of characters.

Use wildcards when you are looking for sets of files. You might, for example, want to check what command files are present on a disk:

There is nothing to stop you using any of these typenames for your own purposes, but it will save confusion if you keep to the standard usages. (*) denotes those used in files created by the CP/M system or by utilities.

ASM	Assembler source code – text that is to be compiled
BAK	(*) Back-up created by CP/M
BAS	Basic source code for compilation
CBL	Cobol source code
COM	(*) Command program
ERL	(*) Relocatable code produced by Pascal compiler.
FOR	Fortran source code
HEX	(*) Hexadecimal code produced by the ASM assembler
HLP	Help text. When all else fails, read the HLP file.
LIB	(*) Library file for use with macro assembler (CP/M Plus)
LST	(*) Listing for printer output – similar to PRN
OVR	Overlay file
PAS	Pascal source code
PRL	(*) Page Relocatable code (CP/M Plus)
PRN	(*) Fully assembled listing produced by ASM assembler
REL	(*) Relocatable routine or program. (CP/M Plus)
SAV	(*) System file
SPR	(*) System page relocatable (CP/M Plus)
SRC	(*) Assembler code produced in compiling Pascal
SUB	File to be executed by SUBMIT
SYM	(*) Symbol file produced by MAC assembler (CP/M Plus)
TXT	(*) Text file produced by WordStar
$$$	(*) Temporary file created by system

Other common typenames

ASC	ASCII file
DAT	Data file
DOC	Document

Figure 1.4 *Standard typenames*

```
A> DIR *.COM
A:ED       COM: ASM   COM: LOAD   COM: DDT   COM
A:ZAPPO  COM
A>
```

The wildcard * here replaced the whole filename, but it can be used for part of the name. In the following example 'Z*.ASM' is equivalent to 'Z???????.ASM' and means any filename starting with Z, of type ASM:

```
A> Z*.ASM
A:ZAPPO   ASM:ZOOM   ASM
```

The question mark can be used for finding files with names of a particular length: A 'DIR ???.*' would display the names of any files with names of three or fewer letters, of any type. It can be mixed with characters: 'DIR ?O*.*' would search for any files where the second letter was 'O':

```
A> DIR ?O*,*
A: LOAD   COM: ZOOM   COM: COMMENTS   COM
A>
```

As with specified filenames, drive specifications can be included. The following command line would search both drives for .TXT files. Notice the exclamation mark that is used to separate the commands:

```
A> DIR *.TXT ! DIR B: *.TXT
A:COMMENTS TXT
B:NOTEPAD     TXT: MEMO    TXT
```

File attributes Though the DIR command does not show it, the disk directory stores more than simply the name and type of its files. It also holds, as you might expect, the whereabouts of the file's data on the disk, and the file's attributes.

A file can be Read Only (R/O), so that it can be accessed and used but not changed, or Read/Write (R/W), giving the user full control.

Its second attribute is its status as a System file or simple Directory file. In CP/M 2.2 the essential difference between these is that SYS files are hidden from the DIR command. In CP/M Plus the difference is more significant and will be returned to later.

The attributes of a file can be seen, and altered, in 2.2 using the transient command STAT which is covered in Chapter 2. In CP/M Plus, an enhanced DIR command reveals the attributes as well as giving extra facilities.

CP/M Plus extensions In CP/M Plus, DIR displays only standard directory files, unless the SYS option is specified. A second version of the command – DIRSYS, or

DIRS – displays only files with the system attribute. DIRSYS is equivalent to DIR with the [SYS] option, and accepts no other options.

Attribute display options In CP/M Plus disk directories, files can be password-protected and given time and date stamps to show when they were created or updated, or when they were last accessed. The techniques for setting these are given later, but they can be seen, if present, by some of the following DIR options:

ATT	Shows the attributes of DIR files, and of SYS files if SYS option has also been selected.
DATE	Shows the time and date stamps, if active.
RO	Picks out the Read Only files.
RW	Shows only the Read/Write files.
SYS	Displays the system files, which would otherwise be hidden.
SIZE	Gives the file size in kilobytes.
FULL	Gives all available data on the files, sorted into alphabetical order.

The options must be written in square brackets [ ], but can be included in the command line at any point. It is only necessary to give the first one or two letters of the option, but it may be written in full. If more than one option is wanted, a comma is used as a separator:

```
A> DIR B:*.TXT [SIZE,DATE]
```

This gives the size and time and date stamps of all files of type .TXT on the disk in drive B.

```
A> DIR [FULL,SYS] *.COM
```

will produce a complete display of the command files on the current disk.

Search options
DRIVE = ... determines the drive(s) which are to be searched. If the option is not used, the current drive is selected by default.

```
A> DIR [DRIVE = ALL]
```

gives a directory of all logged-in drives.

```
A> DIR *.ASM [DRIVE = (A,B)]
```

In the above, the chosen drives are enclosed in brackets. This command is the same as:

```
A> DIR A: *.ASM! DIR B:*.ASM
```

USER = .. likewise selects the user areas to be checked:

```
A> DIR [USER = 5] ED.COM
```

This searches the files of User 5 for the ED utility.

```
A> DIR [USER = (0,1,4)] A*.TXT
```

looks for text files beginning with A in three user's areas. (The concept of users is covered below under the CCP command USER.)

EXCLUDE is used to specify file names and types which are not to be included in the directory display. For example,

```
A> DIR [EX] *.TXT
```

will list all files other than .TXT.

MESSAGE makes the system show which drive and user area it is currently searching, and after each directory has been shown it waits for RETURN or ENTER to be pressed before displaying the second.

NOSORT displays files unsorted, as they are found on the disk. This is only needed to override the normal sort of the FULL option.

Output options

NOPAGE is used with long directory displays which normally wait for a keypress at the end of each screen. This option gives continual scrolling. If the output has been directed to the printer by CTRL and P, then these two options can also be used:

FF	Forces a form feed at the end of each page.
LENGTH = ...	Sets the number of lines to be printed on each page.

ERASE

This command deletes a file or group of files from a disk. In CP/M 2.2 the short form ERA must be used; in CP/M Plus, use either ERA or ERASE.

ERASE does not actually wipe clear those parts of the disk on which the file was stored. Instead, it marks the file as being deleted from the disk directory, and frees the disk space for reuse. You can find fancy disk toolkits that will let you access the disk directory and recover deleted files, but there is nothing in the standard CP/M package which will do this, so ERASE with care.

The command line has the same format as that used for DIR:

```
A> ERA filespec
```

The file specification can include the drive letter where necessary, and wildcards may be used. If there are no files that match the specification, you will get the NO FILE message. Typical examples might be:

```
A> ERA FRED.BAS
A> ERA *.BAK
```

Wildcard use is a quick way to clear a disk of a set of files, but should be used only when you really mean it. In the Plus version only, the use of a wildcard will cause the system to ask for confirmation:

```
A> ERASE *.COM
ERASE *.COM (Y/N)?N        (It was a mistake!)
A>
```

A CONFIRM option, available only in CP/M Plus, allows you to erase selectively from a set of files:

```
A> ERASE *.BAK [CONFIRM]
```

The system will display each .BAK filename in turn and check that it should be deleted by asking for a Y/N response.

As with the DIR options, you only have to type enough of the word for the system to be able to identify it. Here, [C] will do.

TYPE
This resident command will display the contents of a file on the screen or printer. It should only be used with files in ASCII format, i.e. pure text, or assembler or other program listings. Attempts to type out COM files can have highly unpredictable results.

TYPE is probably most useful for displaying the HELP, READ.ME or DOCument files that so often accompany packages, but it also offers a simple way to check through text files or to generate printer copy of program listings.

You must give the full filename. Wildcards are not acceptable here.

In CP/M 2.2 the display will scroll continuously, but can be halted at any time by pressing CTRL and S. Restart the output by pressing any key, or end it completely by pressing CTRL and C. CP/M Plus will normally output one page (24 lines) at a time. This can be changed by the use of the NO PAGE option. Restore paging with PAGE:

```
A> TYPE READ.ME [NO PAGE]
```

As always, output can be redirected to the printer with CTRL and P. This is a toggle (on/off) switch, so pressing CTRL and P again will end printer output.

RENAME
This command allows you to change the name of any file. It is useful for general disk

tidying and comes in handy when you find that you have given a file the wrong typename – for instance, assembler source code must be of type .ASM if it to be processed by the ASM utility. It can also be used before a blanket ERASE to rename selected files out of harm's way. It is an uncomplicated command, but as there are significant differences between the 2.2 and Plus versions the two are treated separately.

In CP/M 2.2, the command must be given as REN, and can only rename one file at a time – wildcards may not be used. Drive letters may be included in the file specification as necessary, allowing files to be renamed and transferred to a new disk at the same time. The format is REN newname = oldname:

```
A> REN NEWFILE.TXT = OLDFILE.TXT
```

It may be thought of in the same way as the Basic assignment statement:

```
LET new$ = old$
```

If you already have a file with the new name on the same disk, the command will abort with a 'FILE EXISTS' message.

In CP/M Plus, the command can be given as either RENAME or REN, and wildcards may be used, allowing the renaming (and disk transfer if wanted) of whole sets of files. Note that the wildcards must be in the same place in both names.

The format is essentially the same as in 2.2:

```
A> RENAME *.TEX = *.TXT
```

If an existing file has the new name, you will be prompted with the following message:

```
NOT RENAMED:filespec ALREADY EXISTS, DELETE (Y/N)?
```

A 'Y' response will delete the old file and replace it with the renamed one.

You can, if you wish, call up the command by the name alone. The system will then ask for the new and old file specification:

```
A> RENAME

Enter new name:_.....
Enter old name:......
```

There are no options with this command.

SAVE

This command will transfer a block of code from memory to disk, and is therefore a useful means of storing the final version of a .COM file after debugging. It also allows you to copy files between user areas, as you will see later.

The two CP/M versions both do the same job, but in distinctly different ways.

CP/M 2.2 To save a block of memory you need to know its size in term of pages – a page being 256 bytes long, or twice as long as a record. If the file already exists, you can find its size (in records) using STAT (see next chapter). The start of the block is always the start of the Transient Program Area – 100hex.

The format is SAVE number-of-pages filename.

```
A> SAVE 16 FRED.COM
```

This saves the 4K block starting at 100hex.

CP/M Plus Here, the SAVE command is given by itself before the file is loaded into memory – usually via SID. Note that SAVE is a transient command in this version, so it is only available if it is on a current disk.

```
A> SAVE
A> SID TEST.COM
```

On exit from that program, you are taken through the SAVE routine rather than returning directly to the system.

CP/M 3 SAVE (Version 3.0)

```
Enter file (type RETURN to exit)
```

If the filename you give is the same as that of an existing file, you will be asked if the previous file should be deleted. Save then asks for the start and end addresses. These should be given in hex. (Digits only – in some utilities you will need to put H after a hex number, but not here.)

USER

Both CP/M 2.2 and Plus can act as multi-user systems if required. Each file can be marked with its user's identification number, and is accessible only to that user. (In the Plus version, files belonging to user 0 and given the system attribute can be accessed by any other user.)

In terms of security, the concept of separate users has only limited value, as there is nothing to stop anyone changing the current user number and thereby accessing another user's files. In CP/M Plus, files can be made secure, if need be, through the addition of passwords:

```
A> USER 7
A> DIR B:
```

```
B:PIP   COM: ED    COM: SEVENUP    TXT: PROG7    ASM
A> USER 0
```

Following the switch to user 7, the DIR command only shows up those files belonging to that user.

The user numbers are from 0 to 15, with 0 as the default value. In CP/M Plus, user numbers (other than 0) are given before the current drive letter in the prompt, and in file specifications:

```
A> USER 7
7A> DIR B:
7B:ED          COM: NOTE7    TEX
7A> USER 0
A>
```

Files can be copied from one user's area to another's by using the PIP command (see Chapter 2). In CP/M Plus, User 0 can copy files out to other users to get them started, but in CP/M 2.2 you can only copy into an area – which means that you have to get PIP there first before you can copy other files. This is where the SAVE command comes in. Call up PIP and as soon as it is ready, exit again by pressing RETURN. The program is now in memory starting at 100hex. PIP is 58 records (29 pages) long in CP/M 2.2.

The procedure for setting up a new user's area on CP/M 2.2 is then as follows:

```
A>USER 0
A>PIP
```

...program sign on and prompt

```
* [RETURN]
A> USER 1    (or whatever)
A> SAVE 29 PIP.COM
A> DIR
A:PIP          COM
```

CHAPTER 2

FILE MANAGEMENT

Introduction

The CP/M operating system revolves around the concept of files and of peripheral devices. The term 'file' covers any organised set of data, whether program listing, program code, word-processing text, numerical or textual data for spreadsheets or databases, A 'peripheral' is any part of the system outside the Central Processing Unit and other key chips. In CP/M terms, the keyboard is as much a peripheral as the printer. Files can be copied from more or less any peripheral to any other – disk-to-disk, disk-to-memory, keyboard-to-disk, memory-to-printer, disk-to-screen or whatever.

It is no surprise then that there are a good number of commands concerned with the management of files. You have already met the few resident commands, and there is also an important set of transient commands. In CP/M Plus, these make up the bulk of those on Side 1 of the system disks supplied with the machines.

Transient commands are those which have to be loaded in from disk whenever they are needed. The ones covered in this chapter are those that you will come back to frequently in your CP/M work. They are the ones used for copying files and controlling the flows of data and for finding out more about the current state of your system and of disks. In a single-drive system you may find it useful to copy many of these onto your normal file disks as well as onto the working masters. Methods of copying individual files are covered at the end of the chapter.

FILE MANAGEMENT

The facilities offered by CP/M 2.2 are also offered by CP/M Plus, though the names and structure of some of the utility programs are different. This chapter covers the common core, taking both versions together.

PIP

This is the Peripheral Interchange Program. Its implementations in CP/M 2.2 and CP/M Plus are virtually the same.

PIP's main function is to copy files on a disk, or from one drive to another. With a single-drive 2.2 system, you cannot use it to copy from one disk to another in the same drive. There is a special program called FILECOPY which will do that job, and we will return to it later.

Amstrad users with single-drive systems shouldn't ignore PIP, for it has several very valuable functions. It can be used for joining files together; it can transfer files between users' areas or between different peripherals; and it can process files while it copies them.

Standard names are given when PIP is used to transfer files between peripheral devices. These are covered in detail in Chapter 3, and for the moment it is enough to know that CON: refers to either the keyboard or monitor part of the console and LST: means the printer.

The basic format of the PIP command is similar to that of RENAME:

```
PIP destination = source
```

For disk copying, both destination and source specifications will normally include drive letter and filename, but one or other part may be omitted in some situations. Wildcards may be used, allowing the copying of sets of files.

When a number of different PIP transactions are to be performed, the program can be called up with its name only. The asterisk prompt will indicate that it is running. Give the 'destination/source' data for each transaction, and press ENTER at the end to leave the program.

```
A>PIP M: = A:MENU.COM
```

copies file MENU.COM from drive A to the RAM disk.

```
A>PIP MENU.COM = B:
```

looks for MENU.COM on drive B and copies it to the current drive.

```
A> PIP
* A:RUNMENU.COM = B:MENU.COM
* B: =K*.*
* [ENTER]
A>
```

Here the same file is copied from B to A, though this time it is renamed; then all files starting with K are copied from the current drive onto the disk in B.

Wildcards must be used with care. They are acceptable where a set of files is being copied without renaming, but if you wanted to retype a set – from .TXT to .TEX for example – and used the line 'PIP *.TEX=*.TXT', the system would not accept it. Likewise, you cannot copy a set of files to the screen or printer with 'PIP CON:=*.TEX' or 'PIP LST:=*.TEX'. Each file must be named individually in this usage.

PIP options

The flexibility of PIP comes through its wide range of options. These are given in square brackets after the source specifications. Several options may be set at once, in which case they can be written in a continuous line or separated by spaces. Do not separate them with commas. (Note that in CP/M 2.2 you must not leave a space before the brackets.)

We will take the options one at a time when they are appropriate.

Combining files Two or more files can be merged into one by writing them into a source list – the filenames should be separated by commas. The destination can be on the same disk, or elsewhere:

```
A> PIP CHAPTER.TXT=PART1.TXT,PART2.TXT,PART2.TXT
A> PIP B:PROGRAM.COM=A:START.COM[O],A:MAIN.COM[O],
   B:ENDIT.COM[O]
```

You should notice from the second example that the source files do not come from the same drive, and that all have the O option selected. The O is needed because the files are Object code, not ASCII. It causes PIP to ignore any CTRL Z characters that may be embedded in the files. If it were not selected, PIP would treat any CTRL Z characters that it met as End of File markers and stop copying that file. The O option is not needed when copying single machine-code files, but only when combining two or more.

Files can be transferred between users with the Gn option – where n is a number between 0 and 15. There is a slight variation here between CP/M 2.2 and CP/M Plus. In 2.2, Gn can only mean 'Get source from user n'; while in Plus it can also mean 'Go to user n'.

When starting up a new user area it is often necessary to copy a number of commands across to the new user. In CP/M 2.2 it would be managed by passing control to the new user and getting the files in from user 0. (Remember that PIP will have to be transferred via SAVE as shown earlier.)

```
A> USER 3
A> PIP
* ED.COM = ED.COM[GO]
* ASM.COM = ASM.COM[GO]
```

In a CP/M Plus system, it is normal to copy out from user 0:

```
A> PIP
* ED.COM[G3] = ED.COM
```

This is the only situation in which options are given in the destination.

Partial copies Sometimes, for example when copying program listings out to a printer or when combining several files, you may only want to copy part of a file. Two options allow you to set the points at which to Start and to Quit copying. This is done by using text strings as markers:

```
A> PIP LST: = TESTPROG.ASM[SLOOPIT^Z QJMP LOOPIT^Z]
```

will print out the TESTPROG.ASM listing,starting from 'LOOPIT' and stopping after 'JMP LOOPIT'. The ^Z at the end of each string is produced by typing CTRL Z, and must be given.

As all commands given through the CCP are automatically converted to upper case before processing, the only way to give lower-case strings to PIP is by going into the PIP program and writing the command line there:

```
A>PIP LST: = TEST.TXT[SPart 2^Z]
```

would produce an error message if it really was 'Part 2' in the text, and not 'PART 2', but you could manage it through this:

```
A>PIP
* LST: = TEST.TXT[SPart 2^Z]
```

Check and translate All the following options assess each individual character of a file as it is copied, altering as necessary. H and I are for use with machine-code files only, and L, U and Z can only work on ASCII files:

H	Checks that a file is in the correct hexadecimal format.
I	Ignores all :00 records in a hex file. (This also sets the H option.)
L	Converts any capital letters to lower case.
U	Converts any lower-case letters to upper case. This is useful for tidying assembler listings – normally, but not necessarily, written in upper case.
X	Turns off the normal check for valid ASCII characters, and copies the file exactly as it is.

Z Sets the parity bit to zero. This is only really needed where an unusual input device is the source for PIP.

Formatting output These options are mainly intended for controlling the layout of printer copies, though some may be useful when PIPping a file to the screen.

Dn Delete all characters after the nth column. Use this to prevent PIP from trying to print beyond the right-hand edge of the paper, or screen area.Owners of the PCW machines who have written across the full width of their 90-column screens may need this to trim output for an 80-column printer.

F Remove Form Feeds currently in the file. If form feeds (ASCII character 12 or ^L) have been set for an earlier printing, then use F and P to set up for paper of a different length, or F alone for continuous printing.

Pn Will set PIP to print n lines before form-feeding – ejecting a page. If n is not given or is set to 1, then the form feed will be after 60 lines, as the standard 66 line paper is assumed.

N Number the output file. The line numbers that are given to text by the ED program are not stored in the file. N will put them in again. The numbering will always be 1,2,3,4.

N2 In this variation, the line numbers will be given with leading zeros, to make them six-digit numbers, and followed by a TAB before the text. As this will force all text to the right, it may be necessary to use the Delete option to prevent overrun from the printer.

Tn TABulated printing. Any TAB markers (CTRL I) will cause the next piece of text to be printed at the next tab position, where n is the tab width. If n is not given, the default value is 8, so that the tab positions are usually at columns 8, 16, 24, 32, 40:.

```
A>PIP LST: = TESTPROG.ASM[D80 F P N2 T12]
```

The listing will be printed in 60-line sections, with a form feed at the end of each page. The lines will be numbered; the text will start at column 12 and be cut off at the 80th column.

Miscellaneous options These are as follows:

A Archiving. Copy only those files that have been changed since the last backup copies were made. (CP/M Plus only.)

C Confirm. Used when a set of files is being PIPped, this will cause the system to check that each individual file is to be copied. (CP/M Plus only.)

E Echo. The file will be echoed to the screen as it is being copied. This should only be done with ASCII files, as attempting to print non-printable characters has unpredictable and sometimes disastrous results.

R Read system files. Normally these are ignored by PIP. R will allow them to be copied – note that the new file will also be a system file.

V Verify that the file has been copied correctly. This can, obviously, only be done where the file was copied to a disk.

W Write over a file protected by the R/O attribute. Normally the system would check with you before attempting to do this. W authorises the overwriting.

Single-drive copying

When you are using the disk drive through the AMSDOS operating system, you can load a program into memory from a disk, then replace the disk with another and save the program on that. You can do much the same in CP/M Plus where the drive is handled under two separate names, so a file can be copied onto a new disk by:

```
PIP B: = A:filename.typ
```

After the program has been read in from A:, the system displays 'Drive is B:' at the bottom right, then runs a banner message across the screen, telling you to put the destination disk in the drive.

This pseudo-second drive is absent from CP/M 2.2. There only one disk can be active at any time. Once a disk has been logged in – at start-up or after a 'Warm Reboot' produced by CTRL C – other disks can be read, but not written to. The disk can only be accessed for writing by a warm reboot, and this will abort any current program. Hence, PIP cannot be used for file copying.

Amsoft have overcome this problem by producing three copying utilities: FILECOPY, SYSGEN and BOOTGEN. Each has its own special function.

FILECOPY is the general file copier. It will handle any file that can be seen by a DIR command, which means that it will copy any 2.2 file, most ordinary Basic and machine-code programs, text files from most word processors, and so on. It can also be used for copying CP/M Plus files, as long as these have not been given date stamps and/or passwords. These require an extended directory structure that cannot be handled by the 2.2 system.

The utility is very simple to use. Make sure that the destination disk is formatted and not write-protected, and follow the instructions, switching the disks as necessary.

Preparing system disks

If a disk is only to be used for the storage of files, then formatting is all the preparation that it needs. On the other hand, if you want to be able to boot up (start) from the disk and have access to the operating system and the basic CCP commands (DIR, ERA, REN, TYPE, SAVE and USER), then it must have the system copied onto it. In CP/M Plus, this is a convenience, but not essential as the system can always be called from a second disk in the alternative drive. In the 2.2 system, with a single drive, it is worth making all disks into system disks unless they are only to be used to hold data files for other programs. It will cost you nothing in disk space, as the operating system is stored in two reserved tracks that are not available for normal files.

In CP/M 2.2 the operating system can be copied to the system tracks of a new disk via the utility SYSGEN. The source disk required by the program can be any system disk, and the tracks can be copied onto a blank formatted disk, or one with files currently stored on it. As only the reserved tracks are used, existing files are not affected even on the fullest disk. SYSGEN copies the start-up specifications created by the SETUP utility (see Chapter 3), and if these include the use of programs not on the new disk you will find it worth copying SETUP as well and reconfigure the start-up routine.

The parts of the system tracks that hold the start-up data can be copied onto existing system disks using the BOOTGEN utility. This offers the quickest way of passing a new configuration around your system disks.

In CP/M Plus, only part of the operating system – the boot sector - is stored on the system tracks. This part is loaded in as part of the formatting process. The bulk of it is held in the .EMS file on Side 1 of the CP/M disks supplied with the machine, and this can be copied across by PIP, just like any other file.

Getting information about disks and files

The key 2.2 command here is STAT. It does not exit in the Plus system, where these information-getting functions are taken over partly by an extended DIR, and partly by the new command SHOW.

STAT enables you to find out how much space is left on your disks and how long individual files are. It also lets you see and alter the Read/Write status of disks and files, and the assignments of peripheral devices to the system.

Used alone, STAT (SHOW in CP/M Plus) will give you a brief run down on the active disk:

```
A> STAT (or SHOW in Plus)
A: R/W, Space 47K
```

The disk in the single drive is Read/Write-enabled and has 47K bytes free.

```
A> SHOW [SPACE]
A: R/W, Space 80K
B: R/O, Space 24K
```

The SPACE option requests a run down on all active drives. Here, the disk in A is Read/Write, and has 80K free; B is Read Only with 24K free.

STAT drive: (SHOW drive:) will report on a specified drive. You must put the colon at the end to indicate a drive, and not a file. An oddity of STAT, when used this way, is that is doesn't report the Read/Write status.

```
A> STAT A:
A: Space 80K
```

STAT will display detailed information about files, or sets of files; much the same as 'DIR filespec [FULL]' in CP/M Plus. Wildcards may be used as needed.

```
A> STAT DUMP.* (DIR DUMP.* [FULL] in Plus)
Recs   Bytes   Ext   Acc
33     5K      1     R/W A:DUMP.ASM
3      1K      1     R/W A:DUMP.COM
Bytes Remaining On A: 65K
```

This shows, for each matching file, the number of 128-byte records it occupies; its size in kilobytes; the number of 16K 'Extents' used by it; its access mode – Read/Write or Read/Only; and finally its file specification.

When a file is established as a random-access data file, space is allocated on the disk for all the records that it is designed to hold, though it will be empty until actual records are keyed in. It's rather like an array, where memory space is allocated at the start, and then gradually used as data is added. An option allows you to see the designated record size:

```
STAT filespec $S
```

An extra column, headed 'Size', tells you the number of 'virtual' records. The 'Recs' column shows the number actually used.

STAT is similar to SHOW in two other respects. The commands will tell you about the users who have files on the disk, and the characteristics of the current disk drive:

```
A> STAT USR:    (notice the colon at the end)
Active User 0
Active Files 0 7
```

This tells you that User 0 is currently active, but that both 0 and 7 have files on the disk. The CP/M Plus equivalent SHOW [USERS] also tells you the number of files, and the number of free directory entries:

```
A> STAT DSK: (SHOW [DRIVE] in CP/M Plus)
    A: Drive Characteristics
1368: 128 Byte Record Capacity
 171: Kilobyte Drive Capacity
  64: 32 Byte Directory Entries
  64: Checked Directory Entries
 128: Records/Extent
   8: Records/Block
  36: Sectors/Track
   2: Reserved Tracks
```

Apart from an additional line '512: Bytes/Physical Record', the output is identical in

CP/M Plus for the standard 3-inch disks. On the 8512, SHOW B: [DRIVE] will display the extra capacity of the extra high-density disks. One final SHOW option is not available through STAT:

```
SHOW drive__letter:[DIR]
```

reveals the number of directory entries still free. CP/M 2.2 users will have to count up the number of files and take that from 64 to find the number!

File attributes

Another aspect of STAT allows you to change the attributes of files. In CP/M Plus, the SET command does this, and is also used for adding passwords and date stamps. These additional functions of SET will be covered in Chapter 3.

In the earlier CP/M system, each file had two variable attributes. It could be protected against rewriting or deletion by making it Read Only, or left fully accessible as a Read/Write file. Secondly, the file could be designated as a system file, or a normal directory file. The 'system' files are not displayed by the DIR command, and so are hidden from casual view. They can still be used, if you know their filespecs, and a STAT *.* will show all files, including system ones.

In CP/M Plus, system files, which can be seen with the DIRSYS command, or DIR with the SYS option, have rather more to them. A system file belonging to user 0 can be accessed by any other user. This makes good sense on a multi-user system, where a single, high-capacity hard disk is used for all regular file storage. User 0 will be the system manager – the teacher or section leader – and it will be his responsibility to maintain the system utilities. The other users – students or junior programmers – will need to be able to use the core programs, but should not need to access them for any other purpose.

As the majority of Amstrads will have only single users, the whole matter of user areas and system files has limited practical application. Carving up storage into user areas may be useful, especially for 8512 owners, with the large RAM disk and high-capacity second drive. It can be easier to keep files in a set of user sub-directories, rather than in one large one.

Write-protection has a more obvious value. Setting a file to Read Only will prevent accidental change or erasure.

To alter the attributes of a file, use the form:

```
STAT filespec $attribute
```

where '$attribute' can be $R/O, $R/W, $DIR or $SYS. Don't forget the $ sign.

```
A> STAT DUMP.*
Recs    Bytes   Ext    Acc
33      5K      1      R/W A:DUMP.ASM
3       1K      1      R/W A:DUMP.COM
A> STAT DUMP.COM $R/O
DUMP.COM set to R/O
A> STAT DUMP.ASM $SYS
DUMP.ASM set to SYS
A> STAT DUMP.*
Recs    Bytes   Ext    Acc
33      5K      1      R/W (A:DUMP.ASM)     (Brackets indicate a system file.)
3       1K      1      R/O A:DUMP.COM
```

A similar sequence in CP/M Plus might read:

```
A> DIR P*.COM [FULL]
   Name            Bytes   Recs   Attributes
PALETTE   COM       1K        8   Dir RW
PUT       COM       7K       55   Dir RW
PIP       COM       9K       68   Dir RW
Total Bytes = 17K   Total Records = 131   Files Found = 3
Total 1K Blocks = 17   Used / Max Dir Entries for Drive A: 30 / 64
A> SET PALETTE.COM [SYS RO]
A: PALETTE.COM set to system (SYS), Read Only (RO)
A> DIR P*.COM
A: PIP COM: PUT COM
A> DIR P*.COM [SYS]
A: PALETTE COM
```

You should notice that with SET more than one attribute can be set at a time, and that they are given in square brackets. The separating space is optional, but does make it easier to read. The attributes are written in the forms:

```
RO   RW   SYS   DIR
```

In CP/M Plus there are five attributes in addition to these – ARCHIVE and four user-defined attributes.

ARCHIVE offers a means of keeping track of back-up copies. The attribute can be SET to ON or OFF, but it is also a key aspect of the PIP [A] option. PIP with [A] selected will make back-up copies of files that have the ARCHIVE attribute set to OFF, and afterwards, turn them all ON. Any that are already ON will be ignored. PIP cannot be used in this way with individual files – wildcards must be given – PIP *.TXT .

```
A> SET filename [ARCHIVE = ON]
A> SET filename [ARCHIVE = OFF]
```

The user-defined attributes can mean more or less whatever you want them to. They are identified as F1, F2, F3 and F4, and are SET to ON or OFF. These, and the ARCHIVE attribute will show up in a DIR command when they are ON.

```
A> SET filename [F1 = ON, F2 = OFF]
```

SET gives you access to other aspects of files that are only present in CP/M Plus – time and date stamps, passwords and user-defined attributes. It also allows you to label and set attributes for whole disks.

Whole disk options

In all of the following the normal syntax rules apply. The disk drive only needs to be named if it is not the current one; and the option identifiers do not have to be typed out in full. You need only give enough – no more than two letters – to distinguish the option from others.

A disk can be labelled by the command:

```
A> SET d: [NAME = label.typ]
```

The label has the same structure as a filename – eight characters for the main name plus three for the extension. You don't have to give disks names, but it can make cataloguing easier. The name can be seen by:

```
A> SHOW [LABEL]
```

A disk can also be given a password:

```
A> SET d: [PASSWORD = password]
```

By itself, this has no effect. You must also turn password protection on with:

```
A> SET [PROTECT = ON]
```

Now, anyone attempting to reset the attributes of the disk must give the password. This in itself provides limited security. The disk directory can still be read, and data files or programs on the disk can be accessed unless they have been given individual protection. The files on the disk can be made secure from rewriting and deletion by setting the normal Read Only attribute as well – SET [PROT=ON,RO] – but this will mean having to change back to Read/Write whenever you want to change a file. The effective aspect of it is that when the protection is on, individual files can be given passwords to restrict access to them. What's more, file protection can be at different levels, as you will see below, while at disk level protection is simply ON or OFF.

From the practical point of view, it is best to use passwords only where they are essential – which for most single users should mean nowhere. It slows things down having to remember and key in a password every time you want to use a file, and there is always a danger of forgetting what the password is! Use an obvious word and other people may guess it; write it down and others may see it; choose an obscure password and write it in an unlikely place and you may forget it and be unable to find your note! Remember, if you lock yourself out of a disk, there is no simple way of getting back in.

You can remove a password by setting it to the single [ENTER] keystroke, or turn it off by setting PROTECT to OFF.

Date stamping

Before you can add time and date stamps to files, you must restructure the disk directory using the INITDIR utility from Side 2 of the system disks. Like most of the single-purpose utilities, it is simple to use.

Place a disk with it on in a drive, and have the disk that is to be initialised ready at hand or in the second drive. Call up INITDIR and follow the prompts:

```
A> INITDIR B:
```

If the directory has already been reformatted to take time stamps, you can use INITDIR to restore the simpler directory structure. The second thing you must do if you are going to use time stamps is to set the system's calendar and clock going. This can be done with the utility DATE:

```
A> DATE SET
Enter today's date (MM/DD/YY):
```

The date must be given as three pairs of digits separated by slashes, and the month is given before the day – something which betrays its American origins. Thus 12 April 1986 would thus be given as 04/12/86.

```
Enter the time (HH:MM:SS):
```

Again, pairs of digits are used, but here colons are used to separate the numbers. This is a 24-hour clock, so 7.30 in the evening would be written as '19:30:00'. Set the time a few seconds ahead of what it really is, as the clock will not actually start at that moment.

```
Press any key to set the time.
```

Once the clock is started, it will keep perfect time until you switch off the computer. It can be seen by calling up DATE:

```
A>DATE
Mon 04/14/86 15:33:56
```

A variation of this will give a continuous time display, which may be useful for stopwatch purposes:

```
A> DATE CONTINUOUS (or DATE C)
```

Pressing any key should stop the display and exit the program. Unfortunately, some versions of the system disks have a bug which locks off the keyboard!

Unless you never turn your machine off, you will have to reset the time at the start of every session. DATE SET can be written into a PROFILE.SUB program so that it is built in to your boot-up routine (see Chapter 3).

The clock can also be set by the single command line:

```
A> DATE mm/dd/yy hh:mm:ss
```

Date and time are given exactly as above, and the system will ask you to start the clock with a key press.

SETting stamps

Time stamps are disk- not file-based. This means that they are active, or not, for all the files on a disk, even though you may only be interested in the stamps on individual files.

There are three types of time stamp. They record when the file was created, when it was last accessed, and when it was last updated. You can have any single one of these active on a disk, or UPDATE and one of the other two active. The structure of the directory does not let you have both CREATE and ACCESS stamps on at the same time.

If you wish to record the date on which a file was created, the CREATE stamp must be turned on before the file exists. If an existing file is edited while the CREATE stamp is active, it will have a new create date given to it, as editing actually produces a new file. To maintain full and accurate dating records of your files therefore, you need to keep the UPDATE stamp on constantly, with ACCESS also on except for those times when new files are created. Stamps are set with the commands:

```
SET [CREATE=ON]
SET [ACCESS=ON]
SET [UPDATE=ON]
```

The stamps can also be given in pairs separated by a comma:

```
SET [ACCESS=OFF,UPDATE=OFF]
```

When setting either CREATE or ACCESS to ON it is not necessary to turn the other stamp OFF if set, as the system will do this automatically. The time stamps on a disk can be viewed by the DIR options [DATE] or [FULL].

Passwords and protection

Passwords are set for files in the same way that they are for disks:

```
A> SET filespec [PASSWORD = password]
```

This gives an individual password to a file, but you can also use wildcards to apply the same word to a set of files:

```
A> SET *.COM [PASSWORD = KEEPOUT]
```

At file level there are four varieties of protection, unlike the simple ON/OFF options for the disks.

DELETE offers the least security. This only requires the password for deleting or renaming the file. It can be read, run – if it is a program – and edited without restriction.

WRITE gives the protection of DELETE and the password is required for writing.

READ prevents all unauthorised use of the file.

NONE cancels any previous protection.

```
A> SET DIARY.ME [PROTECT = READ]
A> SET GENERAL.USE [PROTECT = DELETE]
A> SET WHOCARES.TXT [PROTECT = NONE]
```

Perhaps the simplest way to use passwords is to assign the same one to all the files on a disk, and then to set the DEFAULT option to the same word. When a protected file is called up, the system compares its password with the DEFAULT one. If it is the same the file is accessed immediately, without going through the 'Enter Password' routine. Regular use of a single password tends to reduce security, but it can be changed as often as need be.

```
   A> SET *.* [PASSWORD = MEONLY]
   A> SET [DEFAULT = MEONLY]
```

CHAPTER 3

CUSTOMISED CP/M

In this chapter we will look at how CP/M links peripheral devices into the system, and how those links can be customised to suit your own configuration. We will also tackle another aspect of customising – that of tailoring the start-up routines so that CP/M boots up the way you want it. (You can even choose your own colour scheme!) The main utilities that are used in these respects are STAT and SETUP in CP/M 2.2; and SETDEF, SETLST, SETSIO, SETKEYS, PALETTE and DEVICE in CP/M Plus. While the names are very different, their functions are very similar.

Peripheral devices

As well as being used to access the attributes of files, STAT may also be used to find out about, and alter the assignment of devices attached to the system. In CP/M Plus these functions are handled by DEVICE. If you intend attaching any extra peripherals to your system, perhaps a modem, then you must become familiar with these utilities. Even if you will be using only the standard configuration of monitor, keyboard, disk drive and printer, a knowledge of how devices are handled by CP/M will allow you to use the system in a much more flexible way.

CP/M makes a distinction between physical devices, i.e. the pieces of hardware, and logical devices, i.e. the channels into and out of the system. This parallels the split between BDOS and BIOS in the operating system. The logical devices are fixed while the physical ones, which need to be handled in very specific ways, may be changed. The assignments to physical or logical devices is under the user's control. All devices have identifying names and these differ slightly in the two versions of CP/M.

CP/M 2.2 recognises four logical devices – CON:, RDR:, PUN: and LST:. The names hark back to the earlier systems. CON is the console – normally monitor and keyboard; RDR gets its name from paper tape reader and PUN from paper tape punch though they can be used for any serial input and output devices; LST is the output listing device, which may have been a teletype or a printer.

CP/M Plus distinguishes between console input and output in its names, and it allows a wider choice of acceptable names for logical devices.

CONIN: can also be CON:, CONSOLE: or KEYBOARD.
CONOUT:may be replaced by CON: or CONSOLE:
AUXIN: and AUXOUT: handle additional input and output devices. Either of these may be written as AUX: or AUXILLARY:
LST: can be given as PRINTER:

For either system, the list of recognised physical device names is the same:

CRT: Cathode Ray Tube, which covers keyboard as well as monitor.
CRT2: Auxillary CRT.
LPT: Line Printer – a Centronics type is assumed.
TTY: Teletype or other slow serial output device.
BAT: For Batch Processing – this sets CONIN: to RDR: and CONOUT: to LST:
UC1: User-defined console.
UL1: User-defined listing device.
UR1: and UR2: User-defined readers or other serial input.
UP1: and UP2: User-defined punches or other serial output.
NULL: No device

In CP/M 2.2, STAT DEV: shows the current assignments of your system:

```
A> STAT DEV:
CON: is CRT:
RDR: is TTY:
PUN: is TTY:
LST: is LPT:
```

To change an assignment, use STAT like this:

```
A> STAT RDR: = PTR: (colons must be given)
A> STAT PUN: = TTY:,UP1:
```

The first would assign a paper tape reader to the logical auxiliary input device, while the second directs output from PUN: to two physical devices: teletype and a user-defined device.

The valid STAT commands can be checked by 'STAT VAL'. It produces this display:

```
Temp R/O disk: d: = R/O
Set Indicator: d:filename.typ $R/O $R/W $SYS $DIR
Disk Status :DSK: d:DSK:
User Status :USR:
Iobyte Assign:
CON: = TTY: CRT: BAT: UC1:
RDR: = TTY: PTR: UR1: UR2:
PUN: = TTY: PTP: UP1: UP2:
LST: = TTY: CRT: LPT: UL1:
```

STAT VAL provides a compact summary of the valid options available through STAT; e.g. to set a file attribute use STAT followed by the filename and one of the indicators from the end of that second line – 'STAT B:TEST.ASM $R/W'.

The 'Iobyte Assign' list gives the valid physical to logical device assignments, so that CON: can be any of TTY: CRT: BAT: and UC1:.

CP/M Plus's DEVICE is like STAT DEV but slightly more informative. DEVICE followed by the name of a physical or logical device will give information about that one. Used by itself, the command displays a summary of the assignments:

```
A> DEVICE
Physical Devices:
I = Input,O = Output,S = Serial,X = Xon-Xoff
CRT NONE IO LPT NONE O

Current Assignments:
CONIN: = CRT
CONOUT: = CRT
AUXIN: = Null Device
AUXOUT: = Null Device
LST: = LPT

Enter new assignment or hit RETURN:
```

The first part of the display tells you which physical devices are used, and what their characteristics are. This section can be called up by itself by 'DEVICE NAMES'. The second part shows the assignment of physical to logical devices. To get this alone, use 'DEVICE VALUES'. New assignments can be made at the end of the display. The form is 'logical device = physical device', thus:

```
LST: = TTY:
```

would reassign the line printer output to a teletype.

The change could be made, without going through the display, by typing:

```
A> DEVICE LST: =TTY:
A> DEVICE LST: =LPT:,TTY:
```

You may need to alter the way in which inputs and outputs are handled if you connect new peripherals to your system. In CP/M 2.2 this is done through STAT and the SETUP utility, and in CP/M Plus through DEVICE and SETSIO. The theory is much the same in either case.

The assignment can be configured to suit the device. There are two aspects to this, 'protocol' and 'Baud rate'. Protocol refers to whether or not the system should check that the device is ready to receive data before transmission starts. If XON is set, the system will wait for a 'ready' acknowledgement; but if NOXON is selected, transmission will start immediately, whatever the peripheral's state.

Baud rate is the speed at which data is sent or received, and it is measured in bits per second. In a CP/M system, it can be set to a range of values between 50 and 19,200. The rate needed by a peripheral will be given somewhere in its documentation, but will conform to certain standards. The typical teletype runs at 110 baud; a slow-speed cassette recorder and most bulletin boards on modem networks transmit and receive at 300 baud; Prestel input is taken at 1200 baud, but output travels at the much slower 75 baud; the monitor and keyboard console links are 9600 baud both ways.

Configure the device directly, or as part of the assignment:

```
A> DEVICE UR2 [XON,1200]
A> DEVICE AUXIN: =UR2 [XON,1200]
```

Assignable Baud Rates

50	75	110	134.5
150	300	600	1200
1800	2400	3600	4800
7200	9600	19200	

If you wished to connect a serial device to your system, and an obvious example would be a modem, then the assignments for this are handled separately by the SETSIO utility in CP/M Plus, and within SETUP in 2.2 (see below).

One final option relates to the size of the screen:

```
A>DEVICE CONSOLE [PAGE]
```

will tell you the current values for screen columns and rows. They will normally be 79 by 24, but can be changed by:

```
A>DEVICE CONSOLE [COLUMNS = number,LINES = number]
```

PCW owners can also make use of another command to control the screen size:

```
A> SET24X80 OFF
```

will restore the full 90 column width and 30 line height of the screen, while

```
A> SET24X80 ON
```

(ON can be omitted) pulls it back to the size of the other Amstrads.

SUBMIT utility

The SUBMIT utility is a good place to start customising the way your CP/M system works. It will carry out a set of commands that have been previously stored on a file on the disk. That file must have the filetype .SUB, and can be produced very simply by using ED. Note though, that when SUBMIT processes a file, it creates a small working file on the disk. If the disk is write-protected, SUBMIT will not be able to work.

SUB files are a neat way of executing a regularly used sequence of commands. For example, this disk tidying SUB file will erase all back-up files, check the directory and display the free space.

```
TIDY.SUB

ERA *.BAK
DIR
STAT (SHOW in Plus)

A>SUBMIT TIDY
```

Only the file name is given to activate the command sequence.

The SUB files can be made more flexible through a variety of wildcarding. Up to 9 parameters – identified by the dollar sign and number $1, $2, etc. – can be set in a file. Arguments – specific data – are then included in the command line to replace each parameter. The first argument replaces $1 in the filed commands, the second replaces $2 and so on. You can see it in this version of the TIDY.SUB which will erase a specified .BAK file, then give a directory of a particular type of file:

```
TIDYIT.SUB

ERA $1.BAK
DIR *.$2
A> SUBMIT TIDYIT LETTER TXT
```

This will erase the file LETTER.BAK, then give a directory of all .TXT files.

Start-up routines: CP/M Plus

If there are a set of commands which you will want to carry out at the start of every working session, then it may be simpler to include these in the SETUP utility (CP/M 2.2) or in the special PROFILE.SUB file in CP/M Plus. Both allow you to create command sequences that will be executed automatically on entry to the system.

CP/M Plus is supplied with a PROFILE.ENG file which consists of the two commands:

```
SETKEYS KEYS.CCP
LANGUAGE 3
```

These will reconfigure the keyboard to give it the standard CP/M values, and select the English character set, where the pound sign replaces the hash (#). The file should be renamed PROFILE.SUB to activate it, but this should only be done on the write-enabled working copy of the disk, not the protected master. When the system is booted, it runs the EMS program (Early Morning Start-up) which, amongst other things, looks for a file named PROFILE.SUB. If it finds one, it will process the commands in the file before entering the CCP and waiting for keyboard input.

PROFILE.SUB can be rewritten or extended using ED to give the particular sequence that you want. You may, for instance, like to select a different colour scheme using PALETTE. (PCW owners can use this to select inverse or normal video, and other Green Screen users can use PALETTE to set the intensity of inks.)

PALETTE cannot be described as a user-friendly utility. It must be given in the form 'PALETTE code code code ...' where each code is the colour code for an ink. The ink number itself is not given – the system works that out from its place in the sequence. It can save a little typing, but it does mean that if you want to change INK 3, you have to retype the codes for INKS 0,1 and 2 as well. Just to complicate the issue further, the colour codes are not the ones that you may have become accustomed to in Amstrad Basic.

PALETTE colour codes are based on the intensity of green, red and blue light in an ink. Each light is controlled by two bits within a single byte: green in bits 4 and 5, red in bits 2 and 3 and blue in bits 1 and 0. Code 00 turns the light off, 01 or 10 (they are both the same) give the first level of intensity, and 11 is the brightest available. So, the code 00 00 11 would produce no green or red, but bright blue; 01 01 00 would mix a low level of green and red to produce yellow. (Remember that mixing light is not like mixing paint.)

The colour chart given at the end of the next section shows the codes needed for the various colours available on the Amstrad palette. The binary codes are included just to show how the colours are mixed. Use decimal when setting the colours.

```
A> PALETTE 1 60 12 63
```

This will give a blue background, with bright yellow, red and white inks.

8256 and 8512 machines
With these machines you only have two inks, which must be either black or bright green. The only choice you have is between green on black (normal) or black on green (inverse video).

```
A> PALETTE 0 1    (normal video)
A> PALETTE 1 0    (inverse video)
```

Actually, any numbers between 0 and 63 can be used. The highest one will select the bright green. 'PALETTE 42 57' would work just as well as 'PALETTE 0 1'

Start-up routines: CP/M 2.2

In CP/M 2.2, the start-up routine can be altered by the SETUP utility. This allows you to write in an initial command sequence and a sign-on string, as well as setting the keyboard translations and expansions and the input/output assignments that would be handled by SETKEYS and DEVICE in CP/M Plus.

```
A>SETUP
```

will take you through a series of screens in which current settings are displayed and new ones made. The first two are of immediate interest, as they control the initial commands and screen displays.

Initial command buffer:

```
DIR^MSTAT^M
Is this correct ?(Y/N)_
```

Here the system has been primed to give an instant check on the disk with DIR and STAT. Notice that each command has ^M after it; '^' makes the system convert the character as a non-printing control character by masking its ASCII code into the range 0 to 31. It is equivalent in this case to CTRL M – character 13 or ENTER. The '^' is produced by typing it in from the keyboard and not by pressing CTRL.

Only two commands are given in this example, but the command buffer has space for 128 characters, which allows you room for a very complex sequence.

Sign-on string

```
^\@ww^\ a@@^]wwCP/M 2.2 – Amstrad Consumer Electronics plc ^J^M
Is this correct (Y/N)
```

CP/M 2.2 does not have Plus's PALETTE utility to set ink colours, but this can be done instead by using the non-printing characters 28 and 29 which are control codes for ink and border settings. By including them in the sign-on string, they are 'printed' – executed – and colour codes are passed to the system. As a way of organising your colour scheme it is no more user-friendly than PALETTE.

No.	Sign-on		PALETTE G R B
0	@	Black	00 00 00 = 0
1	a	Blue	00 00 01 = 1
2	b	Bright Blue	00 00 11 = 3
3	c	Red	00 01 00 = 8
4	d	Magenta	00 01 01 = 9
5	e	Mauve	00 01 11 = 11
6	f	Bright Red	00 11 00 = 12
7	g	Purple	00 11 01 = 13
8	h	Bright Magenta	00 11 11 = 15
9	i	Green	01 00 00 = 16
10	j	Cyan	01 00 01 = 17
11	k	Sky Blue	01 00 11 = 19
12	l	Yellow	01 01 00 = 20
13	m	White	01 01 01 = 21
14	n	Pastel Blue	01 01 11 = 23
15	o	Orange	01 11 00 = 28
16	p	Pink	01 11 01 = 29
17	q	Pastel Magenta	01 11 11 = 35
18	r	Bright Green	11 00 00 = 48
19	s	Sea Green	11 00 01 = 49
20	t	Bright Cyan	11 00 11 = 51
21	u	Lime Green	11 01 00 = 56
22	v	Pastel Green	11 01 01 = 57
23	w	Pastel Cyan	11 01 11 = 59
24	x	Bright Yellow	11 11 00 = 60
25	y	Pastel Yellow	11 11 01 = 61
26	z	Bright White	11 11 11 = 63

Figure 3.1 *Colour chart for sign-on strings and the PALETTE utility*

The set of characters up to 'CP/M..' selects colours for the inks and the border. ' ^ ' is equivalent to 'INK' and '^]' to 'BORDER' in Basic. The normal rules for setting colours apply and you must give two colour values — both the same if you want steady colours - for each ink. Number values are given through the ASCII characters @ to Z, where @ means 0, a is 1, b is 2, etc. The standard sign-on string ' ^ @ww^a@@^]ww' thus translates out to:

'INK 0 23 23 INK 1 0 0 BORDER 23 23'.

The '^J' near the end of the string moves the cursor down a line. Other non-printing characters may be included as wanted. A full table is given in Appendix B at the end of the book.

A colour chart for sign-on strings and the PALETTE utility is given in Figure 3.1.

Initialising the printer

In CP/M 2.2 this is done by sending any necessary control codes – perhaps to select a particular typeface – via a string in the Initial Printer Buffer. The codes are in the same form as in the sign-on string.

In CP/M Plus, the printer is initialised by using SETLST which will process a string of control codes stored in a named file. (See SETKEYS for methods of specifying these codes.) The command is:

```
A> SETLST filename
```

Keyboard definition

The two versions of CP/M are significantly different in the way in which they handle keyboard translations and expansion strings. CP/M Plus has the separate utility SETKEYS, while CP/M 2.2 has routines within the SETUP utility. The first that you will meet handles keyboard translations and is very similar to KEY DEF.

Keyboard translations

```
Key code   Normal   Shift   Control

  9        16        16      16
66         19        27      27
79          8       127     127
Is this correct (Y/N)
```

In this example, the COPY key (9) has been set to produce character 16 which is the same as CTRL and P, so that COPY becomes the Printer output toggle. ESCape (66) is equivalent to CTRL and S (19), or the normal CP/M escape code (27) when SHIFT or CTRL are pressed. DELete (79) is made to behave like CTRL and H – the standard CP/M backspace.

If you reply that the current translations are not 'correct', then you will be offered the chance to change them. There are four options: Add a new one (or Alter an existing translation), Delete a current one, Clear the whole set and Finish alterations.

If you wanted to change the DELete key's setting so that SHIFT and DELete would erase the whole line, then you would need to give it character 24. This is the same as CTRL and X. It would be done like this:

```
Enter command:A79,8,24,127
```

The key number follows directly on from the A command, and the translation codes are separated by commas. It is not necessary to give all three if you only want to reset the Normal or Normal and Shift states.

The Delete command works in much the same way – 'D66' would restore standard codes to the ESCape key. Clear and Finish need only the initial letter.

When you Finish, by entering 'F', the routine will display the new translations and check with you again. You can add as many translations as you like, and continue to adjust them until they are to your liking.

Keyboard expansions

The layout and commands of this routine are almost identical to those of the previous routine. To Add or Alter an expansion, give the key number and the string that it is to call up. Control codes can be included, as they are in the sign-on string, but must be signalled by a '^'.

```
Enter command:A 0,ERA *.BAK^M DIR^M
```

F0 – 0 on the number keypad of the 464/664 – will now call up the commands to erase the .BAK files and give a directory.

The SETKEYS utility

This CP/M Plus utility needs a file in which keyboard translations and expansions are detailed. 'KEYS.CCP' is supplied on the system disks and contains the set of instructions that will reconfigure your keyboard to the CP/M standard. If you wish to add to these, or change some of the settings, it may be simpler to copy the file with PIP and edit it rather than write a whole new file.

Each line of the file deals with a different key setting, and has the general form:

```
Key number   shift states   character or expansion string   comments
```

The 'comments' are entirely optional. SETKEYS will ignore anything written after the 'character or expansion string' field.

Shift states are indicated by N (Normal), S (Shift) and C (CTRL) on the 6128; and also A (Alt), E (Extra) and SA (Shift and Alt) on the PCWs. If no state is given then N is assumed; and several may be given at once if a key is to return the same value for different shift states.

Normal printing characters are given enclosed in double quotes, though the only time you would need to set these would be when you are restoring normal settings:

```
44 N "h"
44 S "H"
44 C "^H"
```

Control codes – like H above – are indicated by before a character. This character's ASCII code will then be masked into the range 0 to 31, so that H – character 72 – become 08, which is the backspace code. CTRL and H is now restored to its normal setting as the standard CP/M delete key. For many purposes it is not necessary to learn the codes. The main keyboard already produces the CTRL characters that CP/M needs, and the functions can be given to other keys by stating the letters. There is a good example in the KEYS. CCP file. CTRL and X will delete the current line. The DELete key is configured so that it too will delete the whole line when CTRL is also pressed:

```
79 C "^X"
```

Control codes, or escape sequences, can also be given to keys in two other ways: ^ can be followed either by a number or by the control character's name. In both cases the code specifier is enclosed in single quotes, and the whole definition is in double quotes. These three settings all do the same job:

```
79 C "^X"
79 C "^'#18'"
79 C "^'CAN'"
```

Where you want to call up a string from a single keystroke, the key – any key – can be designated for expansion by giving it a hexadecimal number between #80 and #9F (or #9E on the PCW models). The expansion string is then linked to this number. The examples below come from the 6128 KEYS.CCP file, and they show how the left and right cursor keys are made to jump to the ends of the line when used with CTRL.

First the character code is given to the key in the usual way:

```
1 C "^'9F'"
8 C "^'9E'"
```

Then the expansion string is given, with an E at the front to identify it:

```
E #9E "^F^B"
E #9F "^F^B^B"
```

^F moves the cursor one place to the right; ^B moves it to the beginning of the line, or to the end if it is already at the beginning. The results are that CTRL and left arrow resites the cursor at the start of a line – even if it was already there – and CTRL and right arrow take it to the end.

The expansion string can contain more than control codes. You can set up single key commands in the same way:

```
15 S "^' #80'"
E  #80 "DIR A:*.COM [FULL]^M"
```

A summary of the control codes is given in your machine's User Guide, where you will also find the ASCII standard names that are recognised by CP/M for use in escape sequences.

Setting up for serial devices

If you have a serial printer or a modem which you wish to connect into your system, then you may have to alter the settings on the serial input/output channel – identified as SIO in Plus. The utility SETSIO and the DEVICE option 'DEVICE SIO' handle this in CP/M Plus; and in 2.2 it is one of the SETUP routines. There is much in common in the two versions. You may set:

Transmitter Baud rate
Receiver Baud rate
Number of data bits
Parity
Number of stop bits

In CP/M Plus the XON protocol and control signal handshake are also set via SETSIO.

SETUP

The 2.2 user will meet this display within SETUP:

```
Z80 SIO Channel A: 9600 tx baudrate, 9600 rx baudrate,
8 data bits, NO parity, 1 stop bit
```

The new parameters for the channel must be typed in the order that they are displayed. They are given as data only, separated by commas. To reset parity you would therefore have to give all of the parameters up to and including that one:

```
Enter Channel A parameters: 9600,9600,8,ODD
```

SETSIO

This is rather more flexible in that you can alter one parameter without touching the rest. The new value is given after a word that identifies its purpose. The following command would reset the standard SIO parameters:

```
SETSIO TX 9600, RX 9600, BITS 8, PARITY NONE, STOP 1
```

If TX and RX are the same, the two settings can be replaced by a single number, and all identifying words can be called by their initials. This command could therefore be abbreviated to:

```
SETSIO 9600, B 8, P N,S 1
```

DEVICE SIO option works in an identical way, so that each of these commands resets the baud rate:

```
A> SETSIO TX 300, RX 1200
A> DEVICE SIO [TX 300,RX 1200]
```

SIO parameters
The identifier and acceptable values are as follows:

TX and RX		50,75,110,134.5,150,300,600,1200,1800,2400,3600,4800, 7200,9600,19200
Bits	B	5,6,7,8
Parity	P	Odd, Even, None
Stop	S	1, 1.5, 2
XON	X	=ON, =OFF
Handshake	H	=ON, =OFF

CHAPTER 4

TEXT EDITORS AND WORD PROCESSORS

It sometimes looks as if the only essential difference between these is that word processors are used by writers and secretaries, while text editors are used by programmers. They both create text files which can be manipulated in a variety of ways, saved on disk and copied out onto paper. In theory, if it is called a word processor, it should be easier to use. There is no guarantee that it will be.

Introducing ED

CP/M has its own text editor called ED. ED is flexible, convenient and, most importantly, free – but compared to the typical modern word processor it does not score highly in the user-friendly stakes. For a start, it does not allow on-screen editing, i.e. you cannot move the cursor to different lines to add, alter or delete text. With line-based editors such as ED, each line of text is numbered, and many of the commands work by reference to these numbers.

It is useful for jotting down notes and perfectly adequate for creating program listings, but too awkward to be recommended for any serious writing work. However, lack of memory for file storage is not a problem with this editor. It has a 25K memory buffer in which to handle files but this is no limit to their size. A file can be taken a little at a time from disk, edited and stored back on disk. In theory, a single file could take up the whole of the space on the disk, with the edited version being stored on a second disk – at least that is possible if you are working in CP/M Plus. In 2.2 the limit would be half of the free space. In practice, that size of file would prove unwieldy, and you should also leave yourself some uncommitted space on the disk for the temporary files that are created during cut and paste operations.

Where ED is being used to create a new file, the system opens a temporary, blank file on the disk – identified by a .$$$ type extension – and the text that is typed in through the keyboard is initially held in a memory buffer. Part, or all, of it can be written onto disk at any point, or left there until the end of the session when the whole of the buffer will be written out and the file closed properly.

If you are editing an existing file, this is opened for reading, and a new file is created to take the edited text. At the end of the session, the old file will be converted to a back-up (marked with the BAK type name) and the file name will be transferred to the new file.

To start up the editor, type ED and the name of the file to be created or edited. (It is good practice to always use the same type name for text files.)

```
A> ED COMMENT.TXT
    :*
```

The asterisk is ED's prompt. When you see this, you are in command mode, and as there are rather a lot of ED commands, it might be best to take them a few at a time. Get ED running, and test them out as you work through the rest of this section.

The Insert command

The first and most important command is I – Insert. This switches you to insert mode, where you can type text into the file. Use 'i' if you want mixed capitals and small letters, or 'I' if you want all text to be converted to capitals. (Note, it won't be capitalised when it first appears on the screen, but it will have been changed for storage in the buffer and will be displayed in capitals when you next recall it to the screen.)

This is the only command which responds differently to capitals and small letter versions. All the rest can be given in either case. If you make an error while typing, CTRL and H is the neatest way to delete. This will backspace and erase the character from both screen and memory. The DEL key will also delete the character from memory, but the screen display will show a hatched block followed by the offending character. A mistyped line can be removed by CTRL and U, or CTRL and X.

Press ENTER at the end of each line – a 'line' may in theory overrun the physical line on screen, but it is simpler not to do so. When you want to stop typing new text, press CTRL and Z at the start of the next new line. This returns you to command mode.

Enter Insert mode and type a few lines, then return to command mode. You might like to work through this sample piece of text – the mistakes in the first version are deliberate:

```
:*i
1: What does CPM stand for?
2: Control Program for Microporcessors,
3: Computer Printer nad monitor
4: Complete Peripheral Management,
5:                              (ENTER) for blank line.
6: Who cares – it works!
7: (CTRL and Z) to exit
7:*
```

The numbers that are printed at the start of each line are not stored in the buffer. They are there simply for your convenience, and can be controlled by the V (Verify line numbers) command.

-V turns off the line numbering.
V turns it back on. This is the default condition.
0V has a special meaning. It will tell you how much free space you have and how big the buffer is. The figures appear at the bottom left of the screen.

The Character Pointer

Though you can not see it, there is a Character Pointer (CP) which stores your current position within the text. It can be moved around by several commands.

B	moves the CP to the beginning of the file
-B	moves it to the end.
nC	moves the CP forward n characters
-n	moves it backwards
nL	moves the CP on (or back if n is negative) n lines, and sets it at the start of that line
0L	will move it to the start of the current line
n	number by itself acts the same as a number followed by L
n:	the colon identifies the number as a line number. The CP will move to the start of that line.

The only immediate sign of CP movement is that the line number will change if the CP is moved to a new line:

```
7:*B
1:*3:
3:*40C
4:*-3L
1:*
```

Text display

The T command is used to display text on the screen. It will also be echoed to the printer if the CTRL and P toggle has been used. The position of the CP is not affected by any T command, but T will show you where the CP has got to.

T	used alone, this will display the text from the CP to the end of the line.
nT	displays n lines from the CP: negative n displays lines before the CP.
:nT	displays from the CP up to line number n; a colon before a number always means 'until that line', and the format may be used with most commands.
#T	here, as elsewhere in ED, # means all lines.
0TT	this displays the whole of the current line.

The CP moving commands can be combined with T to display blocks of text. You can include a whole series of commands in a single line. Note that spaces are not needed between the different commands, or within commands.

B#T	displays the whole file from the beginning.
2:4T	displays the four lines beginning at line 2.
3::12T	displays from line 3 to line 12 inclusive.
6:T11CT	moves to line 6 and displays it, then move on 11 characters and displays the rest of the line. In the sample text, this will produce:

```
      6: Who cares – it works!
it works!
      6:*
```

Text editing

These commands will can be used to remove, insert or alter text.

nK Kill n lines from, or before if negative, the CP. Omit n if you only want to delete one line. In the sample text, 'Complete Peripheral Management' sounds impressive, but it's not right. Edit it out with 4:K.

nD Delete n characters from or before the CP. Use it for chopping out chunks or for correcting single letters. In line 3 of the sample, 'monitor' should have had a capital M. Delete the 'm'. You will need to move the CP into place first. The command sequence '3:21CT' should take you there, and show this:

```
        :*3:21CT

monitor

       3:*1D
```

Itext Insert the following text after the CP. If the CP was in the middle of a line, then simply pressing ENTER at the end of the text will make the previous end of line into a new line. Type CTRL and Z (to get ^Z on screen) after the text string to prevent the line split. Finish of the previous correction with:

```
3:*IM^Z  [ENTER]
3: *0TT
3: Computer Printer nad Monitor,
```

I Switches you back to Insert mode, with text going in from the CP. The simplest way to correct a bad line may be to delete it with K, then go to Insert and retype it, e.g. 3:KI. Exit with CTRL and Z as usual. You may like to add a few more lines giving alternative possibilities for CP/M:

```
:*4:I                    (Insert from line 4)
4: Cursedly Perplexing Method,
5: Costs Plenty of Money
6: [CTRL and Z]
```

nFtext Find the given text. If no number is included in the command, then it will find the first occurrence of the text, otherwise it will find the n'th occurrence. The CP will be moved to the start of the line in which the text exists. The message 'BREAK '#' AT ... indicates no match.

nNtext This is a version of Find. If the text is not in the buffer, then the routine will read more lines into memory from the disk file until it finds a match.

S Substitute. This will search through the text for a given string and replace it with a second string. You must type CTRL and Z after the first string, and it should be given after the second string if you need to prevent a line split. The format is therefore:

```
Sold text^Znew text(^Z)
```

Substitute is sometime the simplest way to correct an error. The 'nad' in line 3 could be changed to 'and' by this:

```
:*Snad^Zand^Z [ENTER]
```

Try some other corrections using S. You may like to omit the second ^Z to see the effect.

J Juxtapose. This is similar to Substitute, but more convenient where a long string has to be deleted. Three strings must be given in the command line; the first is the point after which text is to be added; the second is the new text; the third sets the limit of text to be deleted. If you want to delete to the end of the line, then use L – CTRL and L to represent the Carriage Return and Line feed. For example:

```
15: The rain in Spain falls mainly on the plain.
*15:jrain in^ZMajorca falls on my bit of the beach^Z^L^Z
*15T
15: The rain in Majorca falls on my bit of the beach
```

For Find, Nfind, Substitute and Juxtapose, the following all apply:

1 The search will start from the current position of the CP. Make sure it is above the lines to be edited before you begin.

2 The find string must be completely unambiguous. If 'in' rather than 'rain in' had been used in the example, then the new string would have been inserted after 'rain' which ends with 'in'.

3 If capital F, S or J are used, then the routines will search for or insert text as capital letters only. fSpain would find Spain, but FSpain would only find SPAIN and would ignore Spain.

4 If the commands are followed by others in the command line, then the final string must be terminated with ^Z.

5 The commands work once only. If you need to substitute several times within a file, then you should set up a Macro command line. This takes the form:

```
nMcommand_string
```

where n is the number of times to perform the commands.

That sample text might look better if we put question marks rather than commas at the end of each line. We want to start at the beginning of the text and there are something like half a dozen lines that need processing, so the Macro should look like this:

```
B 6MS,^Z?^Z
```

If ED only finds four or five commas to substitute, then you will get a 'BREAK..' message to let you know that the command was not completed fully.

Disk transfers

Several commands handle the movement of text between memory buffer and disk:

E End, saving the file and leaving the ED program. Try it now, and when you have returned to the CCP you can review your file using 'TYPE'. To re-edit the file, call it back again with 'ED filename'. This links the file to ED, but to work on it you must bring it into memory with Append.

nA Append – copy n lines of text from the disk into memory. Either a set number of lines can be stated, or two special options can be used. When you start to edit an existing file, you must first bring all or part of it into memory with Append.

#A will copy the whole file.

0A will copy lines until the memory buffer is half full.

H	This also saves the file, but then returns to ED for re-editing. It is equivalent to exiting with E then restarting with the new file name. You will need to Append text to the buffer again.
O	Restores the file to its Original state before editing started.
Q	Quits the program without saving the new file. You can also quit by pressing CTRL and C.

Block movements

Several block movements commands are available as follows:

nW	Write n lines of text onto disk deleting them from memory. This is only necessary with very large files, when buffer space runs short.
nX	Copy n lines of text, starting at the CP, to a temporary (Library) file. The text in the buffer is not deleted, and the temporary file – normally identified as X$$$$$$.LIB, is removed at the end of the session.
0X	Kills the temporary file. You must do this before using it a second time, as otherwise the new lines will be added to the old file. In CP/M Plus,files can be named, allowing the use of several Library files: e.g. 5XPARA1.LIB creates a five-line file called PARA1.LIB. If you needed to delete it, you would use 0XPARA1.LIB.
R	Read the temporary file into the buffer, inserting it after the CP. In CP/M Plus, a filespec may be given if need be – RPARA1.LIB.

To move a block from one place in the text to another, move the CP to the start of the block, and copy it to disk with X. The lines can then be deleted before moving the CP to the new position and copying the text back in. This can all be managed in a single command line:

```
16:5X5K27:R
```

This would move the five lines starting at line 16 to a new place further down the program. Note though, that line numbers are continually updated, and that after the deletion of those five lines, the current line 27 is the old line 32. It is safer to take the moves one stage at a time, checking the current text and CP using the T command.

Odds and ends

There are two last commands which have occasional value:

U -U	will switch upper-case mode on and off. Switching on before text is appended or inserted will cause the text to be converted to upper case.

nZ	Snooze for n seconds. This is only really useful in a Macro line, where you need time to view the result of a command before it is repeated.

ED80 – a program editor

ED80, in Hisoft's DEVPAC suite, is a good example of program editors. It is far more sophisticated and easier to use than ED, and designed with the specific tasks of creating and editing program listings. Unlike ED, it has full on-screen editing facilities, allowing the cursor to be moved freely around the screen, altering, inserting and deleting at will. More comprehensive controls and the ability to effect block transfers via an internal buffer, rather than by creating temporary disk files, all add to the ease of use. Editing is therefore much quicker and simpler than with the line-based ED.

Its default command set is based on WordStar, which has become almost the standard for word processors. This is fine for established WordStar users, but those accustomed to other processors are not forgotten. An installation program supplied with ED80 allows you to redefine all of the command keystrokes to suit yourself.

A quick tour around the command set will give you an idea of this text editor's capabilities:

Cursor control (21 commands)
Move left/right – by character, word, TAB, or to ends of lines.
Move up/down – by line, block, page or to ends of file.
You can also store a cursor position and leap straight back to it after doing some work elsewhere in the file.

Delete (8 commands)
Backspace or delete under cursor. Delete word, line to left or right of cursor, or whole line.

Block (8 commands)
Mark ends of block and store in buffer. Move, copy, delete, write to or read from disk.

Miscellaneous (15 commands)
Find and substitute, singly or throughout text.
Exit with or without creating new files or back-up files.
Toggles for Insert or Change modes, and Auto-indent.
Whoops – restore deleted line.
Access to disk directory.

It is also possible to embed printer control characters in the text, as long as you are familiar with your printer's own control set.

As there are over 50 commands, some require three keystrokes, e.g. CTRL and Q then K, and it can take a while to learn them all. The Locoscript user would be well advised

Syntax: ED filename edited-filename
The use of the filename is optional

CTRL characters

Active in either mode

CTRL + H	Backspace and delete
CTRL + U	Delete line
CTRL + X	Delete line

Command mode only

CTRL + C	Abandon program
CTRL + E	Start new line without deleting current line

Insert mode

CTRL + Z	Return to command mode

Editing commands

nA	Append n lines from disk to memory buffer
B-B	Move CP to beginning (B) or end (-B) of buffer
nC	Move CP n characters backwards or forwards
nD	Delete n characters from or before CP
E	End and save new file
Fstring	Find string and move CP to the line containing it
H	Save new file and return to ED to re-edit it
I	Enter insert mode
Istring	Insert string at CP
Jsl^Zs2^Zs3	Delete existing text between s1 and s3, and insert s2
nK	Delete n lines after or before CP
nL	Move CP n lines forwards or backwards
nMcoms	Repeat the command line n times
Nstring	Find string in buffer or in disk file
O	Restore Original file
nP	Print n lines from or before CP and move CP n lines, (CP/M Plus only)
Q	Quit, without saving file
R	Read library file into text at CP, Filespec may be given in Plus only
Ss1^Zs2	Delete s1 and replace it with s2
nT	Display n lines after or before CP
U, -U	Upper-case conversion switch
V, -V	Line numbering switch
OV	Display free space
nW	Write n lines to new file and delete from buffer
nX	Copy n lines to library file. OX to delete file
nZ	Do nothing for n seconds

Figure 4.1 *ED ready reference table*

to spend a little time installing the program so that the familiar command keys can be used.

Full word processors

PCW owners will already be aware of the capabilities of a good word processor, for Locoscript is a good word processor. The program will have, as well as the facilities listed above, the capacity to format text in a variety of ways. It will be possible to justify (line-up) the text to left and/or right, or to centre it, and to set margins and tabs. Selecting typefaces and other frills on your printer output should also be simple to do. Some of the better word-processing programs work on the WYSIWYG (say 'wizzy-wig') principle – What You See Is What You Get – so that you can tell how the hard copy will look.

Also, you would expect to have wordwrap. With wordwrap, if the word that you are typing is about to run off the edge of the screen, it is pulled round to the start of the next line. This allows you to type in a continual stream without having to keep glancing at the screen to see where you are. The lack of this is an irritating niggle with ED80. There, you know you are going off the right-hand side because the whole of the rest of the text is shunted over to the left. But then, if you are using the editor to create program listings, this is rarely a problem.

WordStar-style processors, e.g. NewWord from Newstar Software and Pocket WordStar from Quest, have some significant advantages over Locoscript. One is in the level of sophistication – both NewWord and Pocket WordStar have built-in mail-merge facilities, so that a letter produced by the processor can be combined with names and addresses from a database to send personalised standard letters. Secondly, WordStar files can be used with spelling checkers – there's even one built into Newword. The most important advantage though is probably that WordStar-style files are purely ASCII, while those produced by earlier versions of Locoscript contained embedded control characters that needed special processing. Because of this, many other programs, not just word processors, are able to read and use files produced by WordStar or NewWord. Many spreadsheets can read the WordStar text files, and, very important for programmers, WordStar-style processors can be used to generate listings for assembler, Basic and other high-level language programs. The ones that appear in the following chapters were written either on ED80, ED or Amsword – an old favourite among 464 and 6128 owners.

ED ready reference table

A ready reference table of ED commands is shown opposite in Figure 4.1.

CHAPTER 5

PROGRAMMING IN CP/M

In Chapter 1 you will have read how programs written to work in the CP/M operating system achieve machine independence by channelling all inputs and outputs through the BDOS. This has a standard set of functions which call up machine-specific routines in the BIOS whenever data has to be taken from or sent to the screen, keyboard, disk drives or other peripherals.

Machine-code programmers need to get to grips with the BDOS functions before they can write anything to work in the CP/M system. On the other hand, non-programmers and those who intend to write only in high-level languages such as Pascal or Cbasic, do not really need to know how the functions work or how they are accessed. It is sufficient for them to know that they do work, since any good utility or language compiler will take care of all interaction with the BDOS. However, a broad understanding of the lower levels of the system will give you a better grasp of the limitations and the possibilities of work done under CP/M. This chapter is intended to serve as an introduction to programming in CP/M for those with some experience of Z80 assembler, but I would hope that non-programmers will be able to follow enough of what is happening to be able to gain a better grasp of the way in which the operating system is organised.

CP/M was designed around the Intel 8080 chip, not the Z80 that is used in the Amstrad machines. The essential differences between them are that the 8080 has a simpler structure and, following on from this, it uses a simpler instruction set. The standard 8080 and Z80 assembly language mnemonics are also different, but they are compatible in that all 8080 mnemonics have Z80 equivalents, and the Z80 can handle any code generated on an 8080 machine.

Registers

Within each chip is a set of registers (see Figure 5.1) – named memory locations used by the processor for temporary storage and for calculations. They are identified in the same way in both the 8080 and Z80. A, the Accumulator, is a single store within which one-byte arithmetic and logical operations are carried out. BC, DE and HL are register pairs that can be used to store 16-bit data – usually addresses – but B, C, D, E, H and L can also be accessed individually for single-byte storage. The data in H might therefore be an 8-bit number in the range 00-FFhex (0-255 decimal), or the High byte of a larger number stored in HL. All register pairs handle numbers over 255 in the same way. The number is divided by 256 and the result (the high byte) is stored in the first of the pair – B, D or H – with the remainder (the low byte) held in the second register – C, E or L. If you work in hex, the system is much clearer.

If H held F6hex and L held 86hex, then the pair would address location F686. Translated into decimal, this gives 246 in H and 134 in L, and the address is 246*256+134 which is 63,110. (The significance of this particular address in CP/M Plus should become clear later.)

The HL pair has other special functions: 16-bit addition and subtraction is carried out

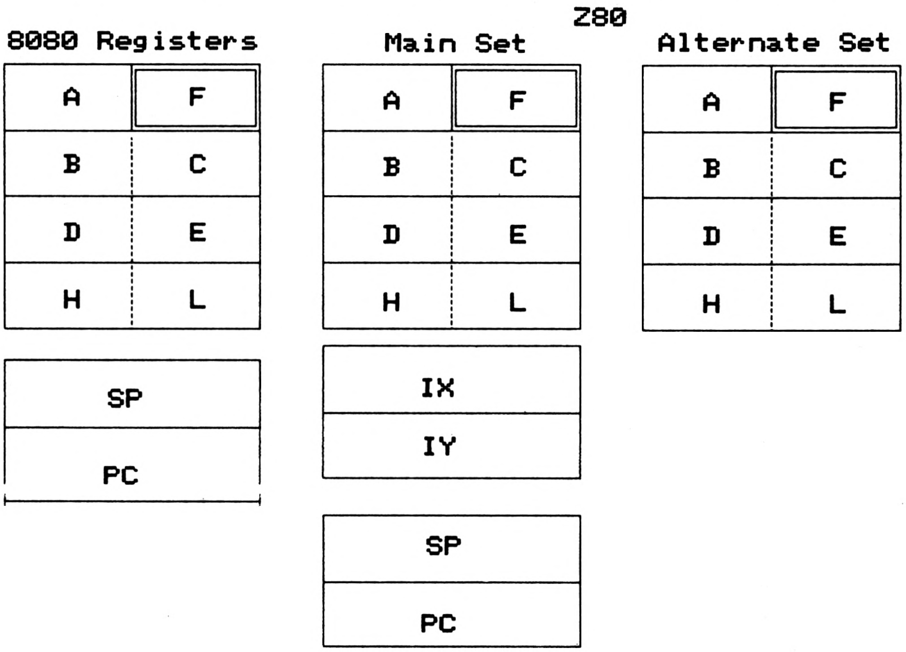


Figure 5.1 *Register formats*

here, and data can be readily transferred between the location addressed by HL and any internal register.

There are two other register pairs within the CPU which are used for storing two-byte addresses. The Stack Pointer (SP) holds the address of that part of memory which is reserved for the temporary storage of data, and the Program Counter (PC) points to the location of the next instruction to be processed.

A final register in the CPU is F – the Flag register. This is really a set of single bits which indicate the results of logical or arithmetical operations. Thus C, the Carry flag, is set to 1 if sum produces a number over 255, or if a 'borrow' is needed in a subtraction. Z, the Zero flag, is set if an operation results in the Accumulator holding zero. There are other flags which can be tackled as needed later.

All of these registers are present in both the 8080 and the Z80 chips, and while the Z80 also has additional internal memory in the form of the alternate register set, this can be left for the time being.

CALL 5

All BDOS functions are accessed in virtually the same way. The C register is loaded with the number of the function. If a single byte of data is to be passed to the function it is loaded into E, or if the function needs two bytes – perhaps the address at which a set of data is stored – then they are loaded into the DE pair. Where data is returned by the function, it will be in A and/or HL.

The BDOS call is always to the same address – 0005 at the bottom of memory. From here, the function call is redirected to a jumpblock at the top end of memory. Precisely where this is will vary from machine to machine, and in CP/M Plus it can change within a machine as the use of RSXs adds extra function addresses to the jumpblock and alters its position. By storing the jumpblock's address in a standard place, the variations in memory size and use are ironed out.

You can trace the progress of a BDOS call through the system by using either DDT in CP/M 2.2 or SID in CP/M Plus. Both are really debuggers, but as they can display the contents of any part of memory, and convert code back into assembler mnemonics, they offer a convenient way of peeking into the firmware. DDT (the Dynamic Debugging Tool) and SID (the Symbolic Instruction Debugger) work in almost identical ways, though SID has a larger command set. The only commands we are interested in at the moment are those that Display memory and List mnemonics.

Call up DDT, or SID as appropriate, by typing its name only. The utility will introduce itself and await your instructions. Normally at this stage you would read in a .COM file for debugging, but not this time. We can trace a path through the firmware by using the L (List mnemonics) and D (Display memory) commands. Either can be used to examine any area of memory.

Type 'D', either by itself or followed immediately by an address in hex – 'D9F06'. (If you type 'D' by itself, the debugger will display a block of memory starting at the current address – initially 100hex, the bottom of the Transient Program Area.)

The display has a special format. The main part of it consists of rows of sixteen hex numbers which show the contents of the memory locations. The addresses of the first location in the rows are listed down the left-hand side, and on the right-hand side are displayed the ASCII equivalents (if any) of the hex numbers. Those character codes that are not in the range 32 to 127 will show up as dots. The sample here is from within SID itself:

```
0180: 43 50 2F 4D 20 33 20 53 49 44 20 2D 20 56 65 72 CP/M SID Ver
0190: 73 69 6F 6E 20 33 2E 30 24 31 00 02 C5 C5 11 80 sion 3.0$1......
```

Type 'L', again either followed by an address or by itself if you want to carry on from where you left off, and the debugger will will convert a small block of code into mnemonics and List them and their addresses on the screen. The mnemonics will be in Intel 8080 format, which can take a little getting used to if you are a Z80 person. What's more, the disassembler cannot handle any instructions that are only in the Z80 set, and this means all bit setting and testing, block instructions like LDIR and any variety of Relative jumps. Unrecognised codes will be shown as ??, and the second or third byte of an instruction may sometimes be mistaken for an 8080 code, leading to patches of confusion. This can be thoroughly irritating if you are trying to disassemble a complex piece of Z80 code, but on this particular exploration it will hardly prove a problem. Most of the code is pure 8080, and all we are really looking for are JMP instructions.

In the following sequence of commands and jumps, the addresses given are those found in the 6128 using CP/M Plus. Other machines and CP/M 2.2 will produce a different sequence of addresses, but the structure will be exactly the same.

We will follow what happens when you CALL 0005. Start then by typing 'L05'. You will see that the instruction there is a jump to another address – JMP DB00. List the mnemonics at this address to see what goes on there. 'LDB00' reveals another jump to E1A4, and 'LE1A4' shows this set:

```
XTHL
SHLD address
XTHL
JMP F606
```

XTHL [EX (SP),HL] switches into HL the two bytes at the top of the stack – and these will be the address within the calling program to which the routine must return. This return address is then stashed in a known, safe location, so that no matter what happens to the stack within the function routine, control can be passed back to the right place in the calling program. This short routine ends with a jump to F606 which leads directly to another jump (to F624) and this is our immediate goal. We have now left the BDOS – the fixed part of CP/M – and entered the machine-specific BIOS.

The actual details of how the next routine works are not really relevant at this stage, and they can be readily worked out by anyone with a reasonable grasp of 8080 code. Suffice it to say that the contents of the DE register pair, and of the single E register are stored for later use as they probably contain data for the function call, and the value in C – the number of the desired function – is used to calculate the address of that function. This address is loaded into HL, then switched with the Program Counter so that the program flow is redirected this way, rather than via another jump.

On my version of CP/M Plus in the 6128, the function addresses are stored in a block at F686. If you try and examine them with 'LF686', you will get junk. This is where D comes into its own. 'DF686' will show this on the first line:

```
F686: 03 FC D5 F8 D5 F8 9E F6 A4 F6 A8 F6 AC F6 E3 F6....................
```

From this we can work out that function 0 is at FC03, function 1 (and 2 – they share a start point) at F8D5, and so on.

If you care to, you can continue with List and Display to follow any of the functions further. What I hope will have come out of this demonstration is that while all function calls start at the same place – 0005 – they are directed to the appropriate addresses, albeit after a rather tortuous trip. This complex sequence of jumps is an inevitable side effect of CP/M's flexibility and the need to cope, not just with different machines, but also with different memory sizes on the same type of machine.

We will return to the debuggers, DDT and SID, shortly when we work through some practical examples of machine-code programming under CP/M, but first let's have a brief look at the scope of the BDOS functions.

BDOS function calls

The BDOS function calls can be divided into two sets. Those numbered 0 to 11 are almost entirely concerned with input and output through the console, printer and auxiliary devices, while the functions from 12 upwards are mainly used in disk file handling. Functions 0 to 36 are common to CP/M 2.2 and Plus – apart from minor variations. All other functions are only present in the Plus version of the Basic Disk Operating System.

All functions are accessed by loading the C register with their number. Some will also require data to be passed to them via the E or DE registers, and some will return data in A or HL.

BDOS input/output functions

Function 0 – System Reset. Equivalent to a warm start.

Function 1 – Console Input. Used to get a single character via the keyboard.

Function 2 – Console Output. Prints a single character on the screen.

Function 3 – Auxiliary Input. Get a character from the device assigned to AUXIN.

Function 4 – Auxiliary Output. Send a character to the device assigned to AUXOUT.

Function 5 – List Output. Send a character to the printer.

Function 6 – Direct Console Input / Output. Flexible alternative to functions 1, 2 and 11.

Function 7 – Auxiliary Input Status. Is AUXIN: ready to send?

Function 8 – Auxiliary Output Status. Is AUXOUT: ready to receive?

Function 9 – Print String. Send a string of characters to the screen.

Function 10 – Read Console Buffer. Input a character string from the keyboard.

Function 11 – Get Console Status. Does keyboard have character ready to send?

Function 12 – Return Version Number. Which version of CP/M?

Function 13 – Reset Disk System. Log in a new disk during a program.

Function 14 – Select Disk. Log in a new disk drive.

Function 15 – Open File. Find an existing file and prepare it for use.

Function 16 – Close File. Updates the disk directory if any changes have been made to the file.

Function 17 – Search for First. Scan directory for first file that matches given specifications. This function uses wildcards.

Function 18 – Search for Next. Follows on from Search for First.

Function 19 – Delete File. Scan directory and mark any matching file as deleted. The function does not actually erase the file from the disk.

Function 20 – Read Sequential. Read the next 128-byte record in a file at the current pointer position.

Function 21 – Write Sequential. Write the next record, overwriting any existing record at the current position.

Function 22 – Make File. Create a new file in the disk directory.

Function 23 - Rename File. Rename any files where the name matches the specifications.

Function 24 – Return Login Vector. Test to see which drives are logged in.

Function 25 – Return Current Disk. Get current drive number.

Function 26 – Set DMA Address. Resite disk read/write buffer (Direct Memory Address) to new location. It is normally at 0080hex.

Function 27 – Get Address of Allocation Vector. This shows the position of records on a disk, and can be used to determine free space.

Function 28 – Write-Protect Disk. Marks a drive as Read-only.

Function 29 – Get Read-only Vector. Test to see which drives are Read-only.

Function 30 – Set File Attributes.

Function 31 – Get Address of Disk Parameter Block. The DPB for the current drive is in the BIOS and stores disk parameter values.

Function 32 – Set/Get User Code. Find or alter the number of the current user.

Function 33 – Read Random. Read a record at a given position in a file.

Function 34 – Write Random. Write a record to a given position.

Function 35 – Compute File Size. Returns address of record after the last. It can be used to calculate size, and also to add new records to end.

Function 36 – Set Random Record. Used after sequential record functions to calculate current (random) record position.

Functions 37 onwards exist only in CP/M Plus. They are covered, along with more detailed explanations of the common set, in Appendix D.

Working examples

This first program given below should illustrate the way in which the BDOS functions are used to read and write to the console, and will also serve to introduce the CP/M assembler utilities. It will work equally well in either 2.2 or Plus.

The CP/M assemblers: ASM, MAC and RMAC On the CP/M 2.2 system disk that is supplied with the 664 and the disk drive DD1 there is only one assembler – ASM, but in the Amstrad Plus utilities you have a choice of two – MAC and RMAC. All three are designed for use with Intel 8080 code, and cannot cope with Z80 mnemonics. If you prefer to use these, you will need a CP/M-compatible Z80 assembler such as GEN80 in Hisoft's DEVPAC 80 suite.

ASM and MAC are very similar, though MAC has extra facilities, and either can be used to assemble the sample READER program. They will convert the source code in the .ASM file into a .HEX file, and also produce an annotated listing of the source code in a .PRN file. This shows the addresses and the assembled hexadecimal code next to the original mnemonics. The .HEX file can be debugged with DDT or SID, or can be processed into working code using either LOAD (2.2) or HEXCOM (Plus). We will return to these shortly. MAC also produces a third file, identified by .SYM, in which the symbols used in the program and their values are listed.

RMAC is for more advanced programmers. It produces relocatable machine code, so that routines can be developed in independent blocks and then brought together in different combinations using the LINK utility. Start by creating the assembly language listing using ED or any other text editor or word processor that will produce pure ASCII files. The file must have the .ASM type – this is called READER.ASM.

Converting Z80 to 8080 mnemonics READER.ASM is written in Intel 8080 mnemonics, partly because all readers will have either ASM or MAC though only some will have a CP/M Z80 assembler, and partly to show that the differences between them are not that significant.

Intel 8080 follows simple rules. An 'I' at the end of the mnemonic indicates that data is given Immediately as a number, rather than through a register or address; an 'X' within a mnemonic shows that a register pair is being accessed. They are referenced by the first letter only, so that B means either the single B register, or the BC pair, depending upon the context in which it is used.

There are no relative jumps, and no compound instructions. The first point doesn't matter much when working in assembly language – it simply means that you must label every point to which a jump is made. The lack of compound instructions – particularly the block load and block compare ones – can be irritating. Using the B register as a counter and DJNZ to control a loop is very handy, and very much missed. Lack of BIT, SET and RES makes bit-level operations more cumbersome, as does the fact that rotations and shifts can only be performed in the Accumulator. If you are used to programming in Z80, have got a suitable CP/M Z80 Assembler and Debugger, and only expect your programs to be run on Z80 machines, then you may well be tempted to ignore ASM, MAC, DDT, SID and the rest. But first, do have a look at them in use.

These 8080 mnemonics are used in the READER program: 'r' stands for register; 'n' for a number; 'add' for address:

LXI rr	Load a register pair	LXIH = LD HL
MVI r,n	Load a single register with data	MVIC,6 = LD C,6
CPI n	Compare contents of A with the number	CPI0 = CP 0
JNZ add	Jump to the address if non-zero	JNZ WAIT = JP NZ,WAIT
INX rr	Increment the register pair	INX = INC HL
DAD rr	Add the contents of rr to the contents of HL	DADD = ADD HL,DE
MOV r1,r2	Load one register from another	MOVA = LD E,A

Assembler commands 8080 and Z80 assembler commands are very similar. In the example, these are used:

ORG	Address at which the code should start – normally 100hex
EQU	For example, BDOS EQU 5 - assign a value to a symbol
DB ...	(DEFB in Z80) The data following this will evaluate to a single byte, or a string of single bytes
DS n	Create data storage space of the given length

READER.ASM

The comments are for reference only, though they may be included in your file if wanted. Anything on a line after a semi-colon is ignored by the assembler.

```
;Read and write to console
;
;BDOS function numbers assigned to symbols
;
         READ     EQU 10
         STRING   EQU 9
         CHARIN   EQU 11
         PRINT    EQU 2
         BDOS     EQU 5
;
         CR       EQU 13       ;carriage return
         LF       EQU 10       ;line feed
;
         ORG 100H              ;bottom of TPA
;
; Print prompt message string
;
         LXI D,MSG
         MVI C,STRING
         CALL BDOS
;
;Clear character buffer
;
WAIT:    MVI C,CHARIN
         CALL BDOS
         CPI 0                 ;No keys pressed?
         JNZ WAIT
;
;Read string into program's buffer space
;
         LXI D,BUFF            ;Address of buffer
         MVI C,READ
         CALL BDOS;
;
```

```
;Move print cursor to next line
;
          MVI E,CR
          MVI C,PRINT
          CALL BDOS
          MVI E,LF
          MVI C,PRINT
          CALL BDOS
;
;Prepare the input string for printing
;
          LXI H,BUFF
          INX H
          MOV E,M               ;Get number of characters
          MVI D,0
          DAD D                 ;Calculate end of buffer
          INX H
          MVI A,'$'             ;String delimiter
          MOV M,A
;
;Print the input string

          LXI D,BUFF            ;String starts at buff + 2
          INX D
          INX D
          MVI C,STRING
          CALL BDOS
          RET
;
MSG:      DB 'PLEASE ENTER MESSAGE.$'
BUFF:     DB 128                ;Buffer size
          DS 129                ;Create buffer space
          END
```

When the listing has been created via ED, and saved, it can be processed through either ASM or MAC. If the files that the assembler will produce are to be written back onto the same disk as the source code, then the command is simply 'ASM READER' or 'MAC READER' – notice that the filetype is not given.

With ASM, input and output disks can be specified by three single-letter codes after the filename. These can be standard drive identifiers (A,B,M,etc.), Z if no output file is wanted, or X if the file is to go direct to the printer. The first letter always refers to the .ASM file, the second to the .HEX and the third to the .PRN file.

```
A>ASM READER.ABX
```

would direct ASM to take the source file from drive A, write the .HEX file on drive B and send the .PRN file to the printer.

As MAC has more options, the simple three-letter grouping is not sufficient. Options are given in pairs of letters – the first referring to the file, and the second of each pair identifies the drive. The whole option list is marked by a '$' at the start.

The file identifiers are A for .ASM, H for .HEX, P for .PRN and S for .SYM. Where macro program segments are used, L identifies a .LIB file.

MAC uses the same single-letter drive identifiers as ASM and Z again indicates that no output file is wanted, but here X will send a file to the screen, and P to the printer.

```
A>MAC READER $AB HZ PX SP
```

directs MAC to find READER.ASM on drive B; to suppress READER.HEX, send READER.PRN to the screen and the symbol table READER.SYM to the printer.

If you have keyed in the listing correctly then at the end of assembly, you should see:

```
CP/M ASSEMBLER – Ver 2.0 (or 3.0)
01D7
000H USE FACTOR
END OF ASSEMBLY
```

The first figure is the address of the last byte of the compiled code. The use factor is a measure of the symbol table space used by the program. Divide by FFhex (255) to find the fraction used – though here no space is taken.

Any errors in the listing will be shown on screen at this point, and will also be written into the .PRN file. A code letter at the start of the line will indicate the type of error.

D	Data of wrong size or type for given data area
E	Expression cannot be evaluated
L	Label duplicated or misused
N	Not compilable by the assembler
O	Overflow – expression needs to be simplified
P	Phase error – a label changes value between passes
R	Register value not compatible with operation
V	Value of operand in expression is wrong

Errors will also occur if the source file is incorrectly named, or not present on the given disk; and if the output files cannot be saved either because of lack of space in either the data area or the directory of the disk, or because the disk is write protected.

The READER.PRN file produced by the assembler should look like this:

```
                    ;Read and write to console
000A =                       READ     EQU 10
0009 =                       STRING   EQU 9
000B =                       CHARIN   EQU 11
0002 =                       PRINT    EQU 2
0005 =                       BDOS     EQU 5

000D =                       CR       EQU 13 ;Carriage return
000A =                       LF       EQU 10 ;Line feed
                    ;
0100                         ORG 100H
0100 113F01                  LXI D,MSG
0103 0E09                    MVI C,STRING
0105 CD0500                  CALL BDOS
                    ;
0108 0E0B           WAIT:    MVI C,CHARIN
010A CD0500                  CALL BDOS
010D FE00                    CPI 0          ;No keys pressed?
010F C20801                  JNZ WAIT
                    ;
0112 115501                  LXI D,BUFF     ;Address of buffer
0115 0E0A                    MVI C,READ
0117 CD0500                  CALL BDOS;
                    ;
011A 1E0D                    MVI E,CR
011C 0E02                    MVI C,PRINT
011E CD0500                  CALL BDOS
0121 1E0A                    MVI E,LF
0123 0E02                    MVI C,PRINT
0125 CD0500                  CALL BDOS
                    ;
0128 215501                  LXI H,BUFF
012B 23                      INX H
012C 5E                      MOV E,M        ;Get number of characters
012D 1600                    MVI D,0
012F 19                      DAD D          ;Calculate end of buffer
0130 23                      INX H
0131 3E24                    MVI A,'$'      ;String delimiter
0133 77                      MOV M,A
0134 115501                  LXI D,BUFF
0137 13                      INX D
0138 13                      INX D
0139 0E09                    MVI C,STRING
013B CD0500                  CALL BDOS
013E C9                      RET
                    ;
013F 504C454153MSG: DB 'PLEASE ENTER MESSAGE.$'
```

```
0155 80        BUFF:  DB 128          ;BUFFER SIZE
0156DS 129
01D7           END
```

After a successful assembly, the .HEX file can be converted into a working .COM file by either LOAD or HEXCOM:.

```
A> HEXCOM READER
HEXCOM VERS: 3.00
FIRST ADDRESS 0100
LAST ADDRESS 01D7
BYTES READ 0055
RECORDS WRITTEN 01
```

File control blocks

This second example of coding in CP/M focuses on the nature of the FCB – the File Control Block. Whenever a disk-file related function is called, the DE register must be loaded with the address of the FCB. Its structure corresponds to the 32-byte block in the disk directory in which all the essential data about a file is stored – its name, size and the locations at which its records are stored on the disk. The FCB is set up with the specifications of the file which is to be accessed, and this data is compared with the blocks in the directory, so that the relevant file can be found. When a file is opened, the control data from the disk is copied to the FCB, and may be modified by subsequent operations. Any changes are recorded there and copied back to the disk directory by a Close File call at the end. The four bytes at the end of the FCB are used for accessing records during random read/write operations.

FCB structure

00	Drive code – 0 = default, 1 = A, 2 = B, etc. In the disk directory, this byte stores the user number, or E5hex if the file has been deleted.
01-08	Filename, padded with spaces if necessary. Wildcard characters may be used here with some functions.
09-11	Filetype, with spaces and wildcards as above.
12	Current extent number. Normally set to 0 by the user but between 0 and 31 in the disk directory.
13,14	Reserved for system use
15	Record count in the disk directory.
16-31	New name and type would be given here between 17 and 27. This block in the disk directory holds the locations of the file's records on the disk.
32	Current record number – not held in disk directory.
33-35	Random record number – not in disk directory.

File attributes are stored in the high bits of the filename and type. As these are all ASCII characters under 127, seven bits are sufficient to store the character code, leaving the eighth bit free. If the high-order bit is 0, then the attribute related to that byte is OFF; if 1, the attribute is ON. Those on the three Type bytes are the most crucial:

T1 (Byte 09) If Bit 7 = 1, then file is Read/Only
T2 (Byte 10) If Bit 7 = 1, then file is System file
T3 (Byte 11) If Bit 7 = 1, then file has been archived – backed-up

In the filename section, the high-order bits in bytes 01-04 hold the four user-defined attributes; bytes 05 and 06 are used to request particular options in BDOS functions 16, 19, 22 and 30; and bytes 07 and 08 are not used. It is not generally necessary to set the attribute bits in an FCB – they are ignored by the system when comparing filenames. The FILEVIEW program acts as a kind of crude directory. An FCB is initialised with wildcards so that all the files on a disk can be accessed. The two related functions 'Search for First' and 'Search for Next' are then used to bring the disk directory entries into memory. They will be copied to the Direct Memory Address area. This is left at the normal location of 80hex, though it can be resited by function 26 if desired. The disk directory data is copied from the disk,one record – 128 bytes – at a time, and the address of the target file within this block is given as an offset multiplier by the Search functions. The value in the A register (the offset multiplier) must be multiplied by 32 and added to the DMA to get the actual address of the file's entry.

The 32 bytes of file data are then printed out as hex numbers and as characters. If you examine the disk map area – bytes 16 onward – you should see a pattern in the allocation of storage space. The first file to be displayed will be the first one that was stored on the disk (unless that particular file has since been deleted) and it will be in blocks 02, 03, 04 and on. The directory order is essentially the order of use, though later files will be slipped into the gaps left by deletions.

The whole display scrolls steadily up the screen. It can be halted in CP/M 2.2 by pressing CTRL and S, but if you are running it under CP/M Plus, you will find that this doesn't work. You may wish to add a routine to test for keypresses to control the scroll.

The program has been given its own Stack area. Normally the Stack is located at 100hex and grows down from there as it is used. As the area between 80hex and 100hex – the normal DMA area – is used by the program, Stack and program would compete for space. You will find new Stack at the end of the program, with the Stack Pointer addressing its highest location.

Create the listing with ED FILEVIEW.ASM, and assemble it with ASM FILEVIEW, or MAC FILEVIEW:

```
                ;FILEVIEW – EXAMINE FCB'S
                ;
0005 =          BDOS     EQU 5
0002 =          PRINT    EQU 2           ;Write to Console
0011 =          FIRST    EQU 17          ;Search for First
0012 =          NEXT     EQU 18          ;Search for Next
                ;
```

```
000D =          CR          EQU 13          ;carriage return
000A =          LF          EQU 10          ;line feed
                ;
0100                        ORG 100H        ;bottom of TPA as usual
0100 210000                 LXI H,00
0103 39                     DAD SP          ;ADD HL,SP (Z80) - so HL = SP
0104 22C401                 SHLD OLDSTK     ;LD (OLDSTK),HL - store old stack address
0107 31C601                 LXI SP,NEWSTK   ;set up new stack for program
                ;search for first file
010A 11A001                 LXI D,FCB       ;FCB at end of program
010D 0E11                   MVI C,FIRST
010F CD0500                 CALL BDOS
0112 CD2D01                 CALL FCBOUT     ;display FCB
0115 11A001     SEARCH:     LXI D,FCB
0118 0E12                   MVI C,NEXT
011A CD0500                 CALL BDOS
011D FEFF                   CPI 0FFH        ;FF if no file found
011F CA2801                 JZ EXIT
0122 CD2D01                 CALL FCBOUT
0125 C31501                 JMP SEARCH
                ;end of main loop
0128 2AC401     EXIT:       LHLD OLDSTK     ;LD HL,(OLDSTK)
012B F9                     SPHL            ;LD SP,HL - restore Stack Pointer
012C C9                     RET
                ;
                ;subroutines
                ;
012D 218000     FCBOUT:     LXI H,80H       ;normal DMA address
0130 87                     ADD A           ;A = offset multiplier
0131 87                     ADD A           ;80+A*32 = File data position
0132 87                     ADD A
0133 87                     ADD A
0134 87                     ADD A
0135 85                     ADD L
0136 6F                     MOV L,A
0137 E5                     PUSH H
0138 0620                   MVI B,20H       ;all 32 bytes of File data
013A 7E         HEXOUT:     MOV A,M         ; LD A,(HL)
013B C5                     PUSH B
013C E5                     PUSH H          ;save valuable registers
013D CD5E01                 CALL HEXPR      ;print hex numbers
0140 E1                     POP H
0141 C1                     POP B
0142 23                     INX H           ;next byte of data
0143 05                     DCR B
0144 C23A01                 JNZ HEXOUT
0147 CD9101                 CALL LINE       ;do carriage return and linefeed
014A 0620                   MVI B,20H
014C E1                     POP H           ;do it again with characters
014D 7E         CHARS:      MOV A,M
```

```
014E C5                  PUSH B
014F E5                  PUSH H
0150 CD8501              CALL CHROUT     ;print ASCII characters
0153 E1                  POP H
0154 C1                  POP B
0155 23                  INX H
0156 05                  DCR B
0157 C24D01              JNZ CHARS
015A CD9101              CALL LINE
015D C9                  RET
              ;
              ;print subroutines
              ;
015E F5       HEXPR:     PUSH PSW        ;save A - the File data byte
015F 1F                  RAR             ;each byte needs two hex digits
0160 1F                  RAR             ;start with hi nibble and rotate 4 times
0161 1F                  RAR
0162 1F                  RAR             ;move hi nibble across to lo
0163 CD7001              CALL HEXIT
0166 F1                  POP PSW         ;get the byte back
0167 CD7001              CALL HEXIT
016A 3E20                MVI A,20H       ;space after each pair
016C CD7E01              CALL OUTCHR
016F C9                  RET
              ;
0170 E60F     HEXIT:     ANI 0FH         ;AND A,0FH - mask off top four bits
0172 FE0A                CPI 0AH         ;0 - 9 digit or A - F?
0174 D27C01              JNC ATOF
0177 C630                ADI '0'         ;ADD A,'0' - get digit for number
0179 C37E01              JMP OUTCHR
017C C637     ATOF:      ADI 37H         ;so 0Ahex produces 'A' - ASCII 41hex
017E 0E02     OUTCHR:    MVI C,PRINT
0180 5F                  MOV E,A
0181 CD0500              CALL BDOS
0184 C9                  RET
              ;
0185 E67F     CHROUT:    ANI 7FH         ;char code under 127
0187 FE20                CPI 20H         ;non-printing char?
0189 D27E01              JNC OUTCHR      ;printable
018C 3E2E                MVI A,'.'       ;replace non-printing with dots
018E C37E01              JMP OUTCHR
              ;
0191 1E0D     LINE:      MVI E,CR        ;carriage return
0193 0E02                MVI C,PRINT
0195 CD0500              CALL BDOS
0198 1E0A                MVI E,LF        ;line feed
019A 0E02                MVI C,PRINT
019C CD0500              CALL BDOS
019F C9                  RET
              ;
```

```
01A0 003F3F3F3FFCB:  DB 0,'????????','???',0
01AD                 DS 17H        ;only first 13 bytes need defining
01C4 0000   OLDSTK:  DW 00
01C6                 DS 30H        ;Stack space
01F6                 NEWSTK:
            ;
01F6        END
```

The program can be assembled equally well by ASM or MAC, and be turned into a .COM file by either LOAD or HEXCOM. If you have altered it and your adjustments aren't doing quite what you want, or if you merely want to look more closely at what it is doing, then its time to turn to one of the debuggers – DDT or SID.

DDT and SID

We have already had a look at the way in which DDT and SID can be used to display the contents of memory and to list assembler mnemonics. These are of obvious value in debugging, but the utilities have other facilities that are even more useful in this respect. The actions of a program's execution can be examined in the smallest detail, and minor – or major – alterations can be made.

Files can be given to the debuggers in several ways. The simplest is to include the filename in the command tail:

```
A> DDT filename.HEX (or SID ...)
A> DDT filename.COM (or SID ...)
```

If a .HEX file is used, it will be converted into the same binary format as a .COM file before use. The filetype must be given with DDT, but SID will assume a .COM type if it is omitted.

You can also call up the debugger alone and load in the required file afterwards using I and R. I – Insert – loads the file specification into memory at 80hex. The file itself is then read in with 'R', in SID you use Rfilename only:

```
A>SID
CP/M 3 SID Version 3.0
#Rreader.com      (Note: no space after the R)
NEXT MSZE PC END
0200 0200 0100 D2FF
```

NEXT is the next space in memory after the file, and is where the debugger is located. PC is the current address of the Program Counter. These two are also displayed by

DDT. MSZE (memory size) and END (end of occupied memory, i.e. the end of the debugger) are only given in SID.

DDT and SID share most commands, though SID has a few extra. We will take the common ones first, and in the order that you are likely to use them. If numbers are ever given with any command in either version, they should be given in hex – without an H at the end. Where the number starts with a letter it is not necessary to give a leading 0 as it is with the assemblers. The debuggers work only in hex.

TRACE will execute one instruction or a block of instructions and display the contents of each register and the status of the flags. The command is 'T', and it should be followed by the number of instructions to be executed if this is more than one. A trace through the first few steps of FILEVIEW would look like this in DDT:

```
-T
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI H,0000*0103
-T5
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=0103 DAD SP
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0100 S=0100 P=0104 SHLD 01C4
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0100 S=0100 P=0107 LXI SP,0206
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0100 S=0206 P=010ALXI D,01A0
C0Z0M0E0I0 A=00 B=0000 D=01A0H=0100 S=0206 P=010DMVI C,11*010F
```

The left-hand column shows the flags:

C = Carry
Z = Zero
M = Minus (sign bit)
E = Even Parity
I = Interdigit carry (Half carry in BCD arithmetic)

The main section displays the registers, with the register pairs BD,DE HL, SP and PC indicated by the first letter only. If you look carefully at these, you can see how the contents change in response to the instructions being executed. At the end of every Trace instruction, the debugger displays the address of the next instruction to be executed.

The SID Trace display is almost identical. The only significant change is in the way in which flags are displayed. There, the flag information is in the same order but where one is set it is indicated by the presence of its identifier, and otherwise its space is held by a dash, e.g. C-M– – is equivalent to DDT's C1Z0M1E0I0 and shows that only the Carry and Minus flags are set.

UNTRACE is a variation on Trace. It allows you to step through the program by executing a given number of instructions, just as Trace does, but the flags and registers are only displayed after the last instruction. This means that you can skip quickly through those sections that you know work perfectly well. 'U' by itself will execute a single instruction and give the display line, and is therefore identical to a single Trace command.

Both Trace and Untrace are one-way processes. There are no options within the commands that let you choose your starting place but it can be arranged through the next command – X.

EXAMINE (X) allows you to look at, and to change, the contents of any register or the status of any flag. Type X by itself and you will get the full Trace-style display. Type X followed by a flag or register code letter and you will have direct access to that item. A new value can be entered at this point, or [ENTER] will retain the current value:

```
-XB
-B=0999 1234 [ENTER]
-XB
-B=1234 [ENTER]
-
```

The code letters are the same as in Trace: flags C, Z, M, E, I and Registers A, B, D, H, S, P. If you are changing values, the flags must be either 0 or 1;the A register takes a two-digit hex number; B, D and H refer to the register pairs and take four-digit hex numbers. A single register within a pair can only be reset by giving the digits for the whole pair.

The fact that P – the Program Counter – can be changed means that you can set the address at which you wish to start a trace. You can test a subroutine in isolation from the main program by setting the PC to its first byte, and initialising the other registers and flags with appropriate values.

GO (G) offers another way of looking at a program's execution. G is followed by the address at which you want to start, so that 'G100' would run the program, assuming it was at the normal bottom-of-TPA position. When you only want to run part of the program, one or two breakpoints can be set. If and when the Program Counter is the same as a breakpoint address, then control is returned to the debugger.

'G100,1A0,2FE' would therefore start the program from the beginning, but stop it when it reached either 1A0 or 2FE. With a program that has many branches, two breakpoints may not be enough to cover all possibilities. In this case additional breakpoints can be written into the code (normally at the ED stage) by using RST 07.

You can Go to any address in the program, though you should always eXamine the flags and registers before doing this to ensure that all hold suitable values.

GO – Go 0000hex – is one way of exiting from DDT or SID. It does a warm start, but leaves the program in memory, where it can be saved onto disk using SAVE.

There is an odd little command that may be useful for checking calculations. This is H, standing for Hex arithmetic. Follow it with two numbers that you want to add or subtract, and it will give you the answer:

```
-H150,60
-1B0
```

The next set of commands allow you to alter the program itself, and to correct those bugs that you may have tracked down with Trace, Go and eXamine.

Substitute (S) lets you replace the values at locations. Its most obvious purpose is in changing addresses, or the values used in comparisons or given to registers. The S command is followed with the address at which you wish to start. The address and the byte at that location is displayed and a new one can be entered in its stead. The next byte is then offered up to you, so that Substitute is another way of working closely through a whole block of memory. Valid data can be left intact by simply pressing ENTER. The process continues until you quit by entering a full stop '.'.

After Substituting values, you would probably want to check the assembled listing with L, then run through the revised version of the program with T or G.

Assemble (A) is an alternative way of rewriting your code. 'A' followed by an address instructs the debugger to accept assembler mnemonics and convert them into code to insert at that place in memory. You can overwrite existing code, or site the new code after the current end. Take care when doing the latter, as the debugger starts at the next page (address ending ..00) after your program.

It may well be that you want to insert a few extra instructions into a block, without overwriting the existing code. You can make space for more by moving a block to a new location. Use Move (M) for this, giving the start and end addresses of the block, and the new address where the block is to go. As in the next command – Fill – the start and end addresses are both included in the block:

```
-M180,1F0,2E0
```

This will move all the bytes between 180hex and 1F0hex to a new location at 2E0hex. Again, mind the debugger's memory area! If you are likely to want to hack your code around, create room for moves and additions by defining a large block of space within or at the end of your program. The assembler line

```
DS 0FFH
```

will give you a full 255 bytes free space for use in debugging.

Fill(F) This is the last of the commands that let you adapt your code. This will fill a block with any given value. Start and end addresses are inclusive, as with Move. You might, for example, want to zero table space while testing a program. If the table occupied 16 bytes starting at 200hex, you would need this:

```
F200,20F,0
```

Saving debugged code

It must be stressed that any alterations you make are only to the code in memory. The disk version of the .COM file is not updated and, of course, the original source file – the .ASM one – is not touched at all.

To keep the debugged .COM file, first make a note of its start and end points, then exit from the debugger with G0, and use SAVE to store the revised version on disk. If you are working in CP/M Plus, there is a Write option that will let you save code from within SID, or otherwise you can call up SAVE before you enter the debugger. Then, on exit, you will be taken through the saving routine.

The only way to write your alterations into the .ASM file, is to take note of what you have done, and to correct the file with ED or another text editor.

SID's extra commands

The Symbolic Instruction Debugger has all of the Dynamic Debugging Tool's facilities and a few more, in the same way that MAC will do all that ASM can plus a little more.

Call (C) will execute a subroutine at a given address, and this command also allows you to set the contents of the BC and DE registers before running the code. Exactly the same results can be achieved in DDT by Going to a subroutine with a breakpoint set at the RETurn, and by setting register values with X before you start. The form is

```
#Cstart,BCval,DEval
```

Either or both of the register values can be omitted, though you would need to insert an extra comma if you only wanted to set DE.

Employ (E) is an alternative to Insert and Read as a means of loading a file into memory for debugging. If need be, the symbol table can be loaded at the same time. The form is:

```
#Efilename,filename.SYM
```

Pass (P) sets a counter and breakpoint at a given address. The counter records the number of times that the program passes that point. Several passpoints can be set by successive use of the command:

```
#P180
#P200,1
```

In the second example of the command, a passpoint is set at 200hex, and an initial value of 1 is given to the counter.

Value (V) displays the current values of NEXT, MSZE, PC and END.

Write (W) is an alternative to SAVE in that it provides a means of saving the contents of memory on disk. As it can be done from within the debugger, it allows several different versions of a file to be saved during a single session. The command is given with the filename, start and end addresses:

```
WFILEVIEW.COM,100,1FF
```

CHAPTER 6

HIGH-LEVEL LANGUAGE PROGRAMMING

Introduction

There are times when programs have to be written in assembler; the times when every second is absolutely vital, or when an awful lot of programming has to be packed into a very small space. Nothing can match pure code for speed or compactness. Fortunately for most of us, these times are rare. Programming in assembler can be a tedious and frustrating business, though it does give a wonderful sense of achievement when you produce something that actually does what you intended.

Working with a high-level language is a different matter altogether. Your machine's native tongue, whether it is Locomotive or Mallard Basic, offers a friendly environment in which to write. You can test out what you are doing as you go along, and errors are reported back as and when they happen. The catch is that programs written in Basic can only be run on machines that have the right Basic interpreter, and the language itself is slow (though both of the Amstrad dialects are faster than most Basics.)

Standard Basic is slow because it is interpreted. This means that each of a program's instructions has to be converted into machine code every time it is executed. Interpretation involves first checking the instruction for syntax or logical errors; then working through a look-up table to find the appropriate routine to handle it; passing any parameters – the words that need printing, the numbers that need to be calculated, or whatever – to that routine; and finally executing the code. And it has to be done every time. An instruction in a 'FOR I=1 to 100' loop would have to be interpreted 100 times.

If all this processing could be done once only, and not during the program's run-time, then it would be much faster – and this is where compiling comes in.

A compiled language – compiled Basic, or Pascal or C – will produce programs that can approach the speed of fully coded ones. They won't be as compact as pure machine code, though that is rarely a problem in the CP/M systems where you have more than 40K (CP/M 2.2) or even more than 60K to spare (CP/M Plus). With Cbasic, the originators – Digital Research – claim an improvement of 8 to 10 times in the compiled version as compared to the interpreted one. In fact, as you will see below, the gain in speed can be even more than this. Pascal and C, as well as others such as Cobol, Fortran, Algol and Forth only exist as compiling languages and they will always be faster than an interpreted language like Basic.

There are non-CP/M versions of Pascal, C and Basic available for the older Amstrad machines and there is little difference in speed or compactness of programs written in these as compared to those produced by CP/M versions of these languages. The great advantage of CP/M is, as always, its portability. You have a wider selection of languages to choose from, and the programs you produce are not machine-specific.

Speed of compiled code with Cbasic

The next sample program was written in Cbasic to test the speed of the compiled code. Its basis is the Sieve of Eratosthenes. This ancient Greek mathematician had the first recorded interest in prime numbers – those which cannot be divided by any other without leaving a remainder. His 'sieve' (see Figure 6.1) filters out those numbers which are multiples of other numbers, leaving only primes. The program works by setting up an 8K block of numbers. Each prime number, from 2 upwards, is taken in turn, and its multiples are ticked off from the block. From a mathematical point of view, the results are interesting in themselves; while from a programming point of view, it is a good test of speed. In Locomotive Basic, an 8K block can be checked, and the prime numbers printed out, in just over 102 seconds. If you cut out the routine that prints the primes on the screen, the run time goes down to a little over 67 seconds. For Mallard Basic, the equivalent figures are 108 and 71 seconds.

In compiled Cbasic, the run time is down to around 43 seconds on a 6128 or half of that on the PCWs, and if the print out of prime numbers is skipped, it drops to a stagggering 3 seconds. This is the speed gain of a compiled program. Calculations are done in a tiny fraction of the time that they would have taken in an interpreted program.

The Cbasic listing given below was written using ED, though any other text editor or word processor that produces ASCII files would have done. The most striking feature of compiler Cbasic, compared to the interpreted version, is its lack of line numbers. In compiler Cbasic, they are only used as labels to mark jump-points within the program. The actual choice of numbers is arbitrary. Line '100' in PRIME.BAS could just as well have been 1, 42, 0.023 or 99999.

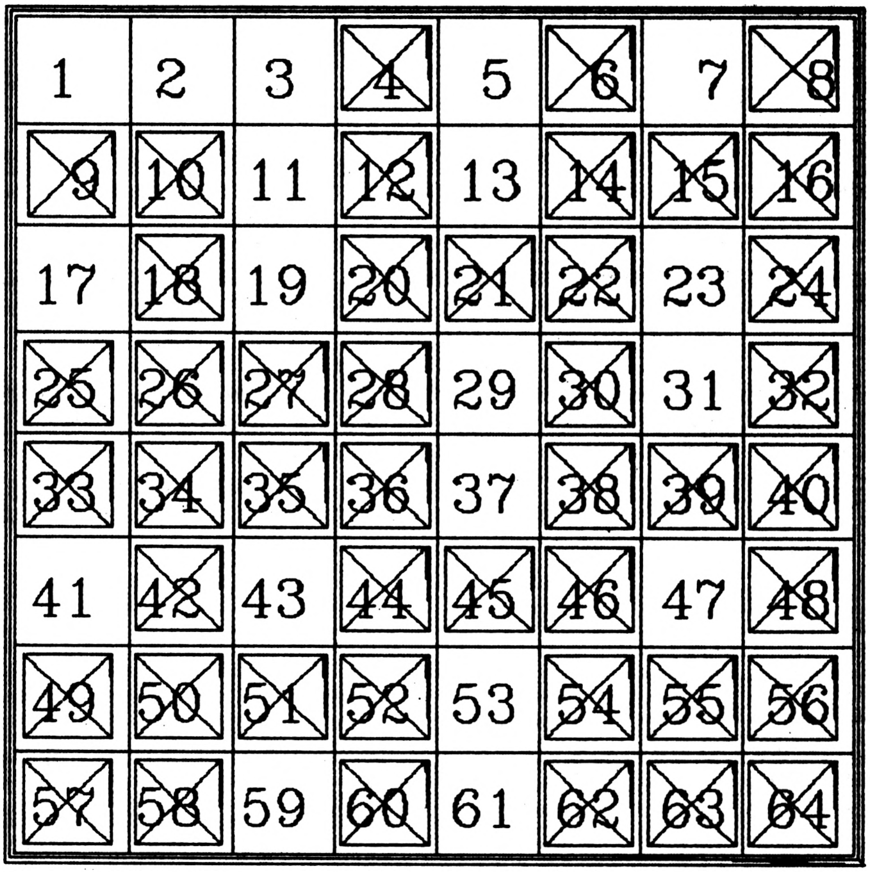


Figure 6.1 *The Sieve of Eratosthenes*

PRIME.BAS

```
REM PRIME NUMBER FINDER
REM to show speed of compiled Basic
REM program takes 102 secs in Locomotive Basic

DIM A%(8191)
REM set up 8K block
FOR N% = 2 TO 8191
    IF A%(N%) = 1 THEN GOTO 100
    REM If A%(N%) = 0 it is a prime
    FOR T% = N% + N% TO 8191 STEP N%
        A%(T%) = 1
        REM mark off all multiples of N%
        NEXT T%
        PRINT N%
100 NEXT N%
    PRINT 'END OF RUN'
    STOP
```

The listing was processed by Digital Research's Cbasic Compiler. It is a two-stage process. First the listing is compiled into a block of relocatable machine code by the CB80 utility. This is then converted into a .COM file with the Linker (LK80) which adds standard routines from the compiler's library file to handle those instructions that are used in the program. These add considerably to the program's bulk. That short PRIME.BAS program was compiled into a relocatable file (PRIME.REL) of under 1K, then linked into a .COM file of 5K. The rather longer file-handling program that appears later in the chapter started as a 1K Basic file, and became a 4K .REL file before reaching a final size of 12K. Compiled programs are not compact!

The Linker can also take a set of .REL files and join them together to produce a larger program. The advantage of this linking is obvious – the programmer can build up a library of compiled routines and functions, and draw on these at will for use in a succession of programs.

The PRIME.COM program takes around 43 seconds to run (around 20 on a PCW) – a fraction of the time of the equivalent (interpreted) Locomotive version – and most of this time is absorbed by the 'PRINT N%' line. Any input or output instructions slow the CP/M system down, as they entail filtering information through the BDOS and BIOS, and the interactions with peripherals, whether monitor, keyboard, printer or anything else, also take time.

Basic file handling with Cbasic

The second sample program should demonstrate the simplicity with which files can be accessed in Cbasic. The language can support two types of data file; either sequential with variable-length records, or random access with fixed-length records. In both types, the file is organised as a set of records, separated from each other by carriage return/line feed characters; and each record consists of one or more fields separated by commas. Data is written to the file by PRINT commands, and READ back into memory.

In a sequential file, the length of a record is entirely dependant on the number of characters in its fields, and can therefore vary constantly. Data is stored in a continual stream, with no gaps between records, and must be read back in the same sequence and the same structure of variables with which it was written. Sequential files are economic in their use of disk space, but not convenient to handle. To update such a file, you must either read all the data into an array, make the alterations then write it all out again; or open a second file and copy the records across, updating them in the process. Either way, the whole file has to be read and rewritten, even if you only want to alter a single field of a single record.

```
REM Telephone number sequential file
CREATE 'TEL.NUM' AS 1
PRINT #1, 'GEORGE','22345'
PRINT #1, 'AUNT AGATHA','OSWESTRY 777349'
PRINT #1, 'AMSTRAD CONSUMER ELECTRONICS','0277 230222'
CLOSE 1
```

The file from this is a continual stream of data divided only by the field (,) and record separators ([CR/LF]).

```
-----------------------------------------------------------------------------------------
'GEORGE' , '222345'    [CR/LF]          'AUNT AGATHA' , 'OSWESTRY
-----------------------------------------------------------------------------------------
777349'       [CR/LF]  'AMSTRAD CONSUMER ELECTRONICS' , '0277
-----------------------------------------------------------------------------------------
230222'           [CR/LF]
-----------------------------------------------------------------------------------------
```

It could be read from disk and displayed by a routine like this:

```
    OPEN 'TEL.NUM' AS 1
    IF END #1 THEN 200
    REM 'IF END' causes a jump when End-Of-File is reached
100 READ #1,NAME$,TEL$
    PRINT NAME$
    PRINT TEL$
    GOTO 100
200 CLOSE 1
    STOP
```

Fixed-length files allow much simpler and quicker updating. In these, each record has the same predefined length, so that the position of any given one within the file can be calculated. This means that the location of a record on the disk can also be calculated, and therefore individual records can be read and rewritten without having to access the whole file. The short program above can be converted to produce random-access files by adding a record-length specification in the first line, and by including the record number in the line that writes to the disk. The rest of the routine remains the same:

```
CREATE 'TEL.NUM' RECL 40 AS 1
PRINT #1,1; 'GEORGE','22345'
PRINT #1,2; 'AUNT AGATHA','OSWESTRY 777349'
PRINT #1,3; 'AMSTRAD CONSUMER ELECTRONICS','0277 230222'
CLOSE 1
```

The resulting file has a much clearer structure:

```
--------------------------------------------------------------------------------
'GEORGE' , '222345'                                                     (CR/LF)
--------------------------------------------------------------------------------
'AUNT AGATHA' , 'OSWESTRY 777349'                                       (CR/LF)
--------------------------------------------------------------------------------
'AMSTRAD CONSUMER ELECTRONICS' , '0277 230222'                          (CR/LF)
--------------------------------------------------------------------------------
```

When reading back in, the record numbers are used to find the right data:

```
OPEN 'TEL.NUM' RECL 40 AS 1
INPUT 'RECORD NUMBER?';R%
READ #1,R%;NAME$,TEL$
PRINT NAME$
PRINT TEL$
CLOSE 1
```

The BOOKFILE.BAS program given below illustrates the essential principles of random-access files and could form the basis of a more sophisticated one of your own. The program is geared up to handle the single file 'BOOK.DAT', and the record length is fixed at 80 characters. If the filename and record length were treated as variables, then the program could have wider application. To do this you will need to add a new routine that INPUTs the filename and length; then start each section with a line like this:

```
OPEN file$ RECL rl% AS 1
```

You will, of course, also have to keep a record of the filenames and record lengths. It might be an idea to create a master file in which details of any others are stored.

The first record in the file is used to store the number of records. This means that the first record from the user's point of view is actually the second, and so on through the file. You will find the +1 adjustment in every section of the program.

The point made earlier about line numbers crops up again here. The subroutines are numbered 100, 200, 300, 400, 500 and 600. These numbers are purely arbitrary. They could have been anything, and they don't have to be in any kind of order.

BOOKFILE.BAS

```
      REM to demonstrate file handling in Cbasic

      REM menu
1     FOR I% = 1 TO 25:PRINT:NEXT I%
      REM clear the screen
```

```
     PRINT '**********************************************'
     PRINT '*                                            *'
     PRINT '*           ADDRESS BOOK FILE MENU           *'
     PRINT '*                                            *'
     PRINT '**********************************************'
     PRINT
     PRINT 'CREATE NEW FILE ...............................1'
     PRINT
     PRINT 'DISPLAY FILE CONTENTS .........................2'
     PRINT
     PRINT 'UPDATE SINGLE RECORD ..........................3'
     PRINT
     PRINT 'ADD NEW RECORD ................................4'
     PRINT
     PRINT 'FIND GIVEN RECORD .............................5'
     PRINT
     PRINT 'EXIT PROGRAM ..................................6'
     PRINT
     PRINT

     REM get choice character
     I% = INKEY IF CHR$(I%) = '6' THEN STOP
     WHILE CHR$(I%)<'1' AND CHR$(I%)>'5'
          I% = INKEY
     WEND
     I% = I%-48
     ON I% GOSUB 100,200,300,400,500
     GOTO 1
     REM loop back to start


     REM create new random access file

100  CREATE 'BOOK.DAT' RECL 80 AS 1
     REM file has records 80 characters long
     INPUT 'NUMBER OF RECORDS:';R%
     PRINT #1,1;R%
     REM first record stores file length
     FOR N% = 2 TO R% + 1
          INPUT 'NAME:';NAME$
          INPUT 'ADDRESS:';ADD$
          INPUT 'TEL.NUMBER:';TEL$
          PRINT # 1,N%;NAME$,ADD$,TEL$
          REM each record is stored at position N%
     NEXT N%
     CLOSE 1
     GOSUB 600
```

```
    REM wait for a key routine
    RETURN


    REM display all records

200 OPEN 'BOOK.DAT' RECL 80 AS 1
    READ #1,1;R%
    PRINT 'CURRENT NUMBER OF RECORDS ';R%
    PRINT
    FOR N% = 2 TO R% + 1
        READ #1,N%;NAME$,ADD$,TEL$
        PRINT 'RECORD NUMBER: ';N%-1
        PRINT 'NAME:';NAME$
        PRINT 'ADDRESS:';ADD$
        PRINT 'TEL.NUMBER:';TEL$
        PRINT
        I% = INKEY
    NEXT N%
    CLOSE 1
    GOSUB 600
    RETURN


    REM update existing record

300 OPEN 'BOOK.DAT' RECL 80 AS 1
    PRINT 'UPDATE ROUTINE'
    PRINT
    INPUT 'RECORD NUMBER:';Q%
    READ #1,1;R%
    REM validity check
    WHILE Q%>R%
        PRINT 'INVALID NUMBER.'
        INPUT 'RECORD NUMBER? ';Q%
    WEND
    REM get the record
    READ #1,Q% + 1;NAME$,ADD$,TEL$

    REM check each field

    PRINT 'NAME:';NAME$
    INPUT 'IS THIS RIGHT? (Y/N)';A$
    IF UCASE$(A$) = 'N' THEN INPUT 'NEW NAME: ';NAME$

    PRINT 'ADDRESS:';ADD$
    INPUT 'IS THIS RIGHT? (Y/N)';A$
```

```
     IF UCASE$(A$) = 'N' THEN INPUT 'NEW ADDRESS:';ADD$

     PRINT 'TEL.NUMBER: ';TEL$
     INPUT 'IS THIS RIGHT? (Y/N)';A$
     IF UCASE$(A$) = 'N' THEN INPUT 'NEW TEL/NUMBER:';TEL$

     PRINT  #1,Q% + 1;NAME$,ADD$,TEL$
     REM rewrite updated record
     CLOSE 1
     GOSUB 600
     RETURN

     REM add new record at end of file

400  OPEN 'BOOK.DAT' RECL 80 AS 1
     READ  #1,1;R%
     REM get old number of records

     PRINT 'ADD NEW RECORD'
     INPUT 'NAME: ';NAME$
     INPUT 'ADDRESS: ';ADD$
     INPUT 'TEL.NUMBER: ';TEL$
     R% = R% + 1

     PRINT  #1,R% + 1;NAME$,ADD$,TEL$
     PRINT  #1,1;R%
     REM save new record and new total
     CLOSE 1
     GOSUB 600
     RETURN

     REM find a matching record

500  OPEN 'BOOK.DAT' RECL 80 AS 1
     INPUT 'NAME TO FIND: ';FIND$
     READ  #1,1;R%
     FOUND = 0
     REM flag to signal result of search
     FOR N% = 2 TO R% + 1
          READ  #1,N%;NAME$,ADD$,TEL$
          IF  MATCH(FIND$,NAME$,1)  THEN GOSUB 550
     REM 'MATCH' will check NAME$ to see if FIND$ is anywhere within it
     REM so if FIND$ = 'FRED', it will find any name containing 'FRED'
     NEXT N%
     CLOSE 1
```

```
    IF NOT FOUND THEN PRINT 'NO MATCHING NAME'
    GOSUB 600
    RETURN


    REM print matching records

550 PRINT 'NAME: ';NAME$
    PRINT 'ADDRESS ';ADD$
    PRINT 'TEL.NUMBER: ';TEL$
    PRINT 'RECORD NUMBER: ';N%-1
    PRINT
    FOUND = 1
    RETURN

600 PRINT 'PRESS ANY KEY TO CONTINUE'
    I% = INKEY
    WHILE NOT I%
        I% = INKEY
    WEND
    RETURN
```

For the experienced Basic programmer, Cbasic is probably the simplest way to create new CP/M utilities. It doesn't take long to get the hang of the compiling routines, and the worst problems you are likely to meet are those that relate to disk space and memory size. Digital Research's Cbasic compiler, linker and associated utilities and library files take up approximately 143K of disk space. This isn't a problem where high-capacity floppy or hard disks are part of the set-up, but owners of Amstrads (other than the 512s) are limited to 170K per side and if the complete Cbasic suite is present it doesn't leave enough space on a disk to do anything useful.

There are several solutions of which the simplest is to create a working disk that lets you write and compile the program on one side, and link it – perhaps adding graphics – on the other. On the first side you will need the compiler itself and its overlays, and also a text editor plus either PIP or FILECOPY. One of these last utilities must be there so that the compiled .REL file can be transferred to the second side for the final part of the process. The disk will have about 100K free space on either side, which should be ample for developing a program. When the program is complete it can be copied to a disk of its own, and the working space on the development disk can be cleared for the next program.

PCW owners can speed up compiling or linking by copying the working disk (or as much of it as will fit and is needed) into drive M and then switching to that drive while they edit and compile, or link, the program. At the end of the session, the day's work can be copied out onto the development disk. Drive operations carried out in RAM are significantly quicker than when an actual drive is used.

We will return to Cbasic in the next chapter when we look at graphics, as it offers one of the simplest ways to produce graphic programs. Before then let's turn to another language that is readily available to Amstrad owners – Pascal.

Pascal

This language was originally designed for programming students. It is highly structured, but uses common words and logical syntax so that programs are generally easier to read than those written in other languages. It is also highly flexible, leaving the programmer a lot of freedom over how he organises data; and if you have a good compiler the final coded version will run at high speed. Today, Pascal has gained wide acceptance as a programming language for students, but it has also been taken beyond the colleges and universities to be used in many scientific and commercial applications.

In Pascal you start with four types of simple variable: CHAR – characters; INTEGER – whole numbers between – 32767 and +32767; REAL – floating-point numbers; and BOOLEAN – logical TRUE or FALSE. Many versions now also recognise a fifth type – STRING, used to store variable-length strings of characters. These simple data types can be used within structured variables – arrays, sets, files and records. Arrays are much the same here as in Basic; sets are based on the concept of sets in Boolean algebra and are largely used for logical testing: files and records come into play in disk-based filing programs. If you want to arrange your data in a special way, then the language also allows you to define your own simple types and structures.

A Pascal program will always have a shape similar to that of the sample program given below. At the start, any constants – the Basic programmer might like to think of these as variables with a fixed value – used in the program will be given; and if new data types are needed they will then be defined. These are followed by the declaration – names and types – of the variables that will be used. The next part of the listing will be the definition of functions and procedures, which are themselves structured with variable declarations given before the operational routines. Finally the main program draws it all together. The structure is based on the concept that you should never use anything which you have not already defined.

While Pascal is not a hard language to learn, the need to organise and structure everything from the very beginning can be a nasty experience for the confirmed Basic hacker!

The sample program exploits one of the attractive features of Pascal – the ease with which recursive functions can be written. A recursive function, or procedure, is one that will pass a problem back to itself for processing and do it again and again, until it has reduced it to such a simple state that it can be solved. It means that a recursive function may be deeply nested within itself, but as each level can create its own separate set of variables, data is not lost in the process.

REROOT works out any root of a number by a series of calculations, each based on the previous one. The concept can be seen at its simplest with the square root. You can work out the square root of a number by starting with a guess. Divide the number by the guess, add the guess to the result, and divide it all by two. Take the result and use that as the next guess:

```
NEXTGUESS = (NUMBER/GUESS + GUESS)/2
```

For example, to find the square root of 16, start with a guess – perhaps 2. (Yes, it is a bad guess, but it allows us a nice simple example!) 16/2 + 2 is 10, and half of this is 5. Use that as the next guess, and you have the sum (16/5 + 5)/2. This gives you 4.2. Run that through again and you have 3.9. Carry on a couple more times and you finish up with 4.

The same basic formula is used to find any root. It can be summarised as:

$$\text{NEXTGUESS} = \left(\frac{\text{NUMBER}}{\text{GUESS} \wedge \text{(ROOT-1)}} + \text{GUESS} * \text{(ROOT-1)}\right) \Big/ \text{ROOT}$$

So, to find a cube root, the formula for the 'Nextguess' would be:

$$\text{NEXTGUESS} = \left(\frac{\text{NUMBER}}{\text{GUESS} \wedge 2} + 2 * \text{GUESS}\right) \Big/ 3$$

The calculations could be written in a single line in Pascal, but for ease of reading it is broken down into several lines in the ROOT function in this program:

```
PROGRAM REROOT;
  (* to calculate roots of numbers *)
VAR
  NUMBER : REAL;
  GETROOT : REAL;
  FIRSTGUESS : REAL;
  ANSWER : REAL;
  AGAIN : CHAR;

FUNCTION ROOT (GUESS : REAL) : REAL;

VAR
    FRACTION : REAL;
    I : INTEGER;
    NEXT : REAL;
    TIMES : INTEGER;
    GUESSTIMES : REAL;

BEGIN
    FRACTION: = 1.0;
    TIMES: =TRUNC(GETROOT)-1;
    (* convert real to integer for loop counter *)

    FOR I: = 1 TO TIMES DO
      FRACTION: =FRACTION * GUESS;

    GUESSTIMES: =GUESS * (GETROOT-1.0);
    NEXT: =(NUMBER/FRACTION + GUESSTIMES)/GETROOT;

    WRITELN('CURRENT APPROXIMATION = ',NEXT:6:3);

    IF ABS(NEXT-GUESS)<0.001 THEN
        ROOT: =NEXT                    ; the solution
    ELSE ROOT: =ROOT(NEXT)             ; go back into the function again
END;

(* main program starts here *)

BEGIN
    REPEAT
      WRITE('PLEASE ENTER NUMBER: ');
      READLN(NUMBER);
      WRITE('PLEASE ENTER ROOT REQUIRED: ');
      READLN(GETROOT);
      FIRSTGUESS: =NUMBER/GETROOT;
      ANSWER: =ROOT(FIRSTGUESS);
      WRITELN(NUMBER:6:2,' TO THE ROOT ',GETROOT:6:2,' =
',ANSWER:6:2);
      WRITELN;
      REPEAT
        WRITE('ANOTHER? (Y/N)');
        READLN(AGAIN);
      UNTIL AGAIN IN ['N','n','Y','y']
    UNTIL AGAIN IN ['N','n']
END.
```

Of course, this program also shows one of the limitations of Pascal. It does have fairly few built-in mathematical functions. You could find a root of a number in Basic with the expression:

```
ANSWER = NUMBER ^ (1/ROOT)
```

Digital Research's Pascal compiler is PASCAL MT+. Turning your Pascal listing into a .COM file is a two-stage process, just as it is in compiled Cbasic. The compiler, MTPLUS, produces a relocatable file which must then be linked with routines in the library files to get the final program.

Both compilers are efficient and well-documented, but they are designed for professional use. On the one hand, this means that you have the tools to do a very good job, but it also means that it takes time to master them. If you haven't tried this kind of programming before, you could be floundering for a while as the documentation does not make many concessions to inexperience.

The problems of disk and memory space that we met earlier, when looking at Digital Research's Cbasic, apply here with even more force. The compiler and its overlays take 108K of disk space – and you still need PIP and a text editor at hand. The linker and its associated files occupy a similar amount of space. Before you start to do any serious Pascal programming you must first establish an efficient system of disk management.

Pascal MT+ could be used for graphics programming, but not easily. There are no built-in graphics commands or functions, though you can access the GSX system via assembler language routines linked into the Pascal program.

At the time of writing, there are several other Pascal compilers on the market, and of these Turbo-Pascal produces the fastest code in the shortest time. Hisoft's Pascal 80 is also well worth looking at and a fraction of the price of Turbo-Pascal. The ability to access GSX graphics, through Logo-style routines is an extra bonus with Hisoft's version.

Compilers for Pascal, Basic as well as Forth, Lisp, C, Pistol, Pilot and Cobol can be obtained via the CP/M User Group. Documentation (on disk) and sample programs in most of these languages are also available. All it will cost you is the annual membership, the price of the disks, a small copying charge and probably a little more effort than would be needed with commercial software. You may find that the version you get will need some adjustments to install it on your machine, especially with the more esoteric languages.

The opportunity to explore other languages is one of the great attractions of CP/M for the serious hobbyist or student. And when you do find one that is suitable for the applications you have in mind, you will have the satisfaction of knowing that your program can be used on any other CP/M machine.

CHAPTER 7

GRAPHICS AND CP/M

Introduction

A major difference between CP/M Plus and the earlier 2.2 version is that Plus gives access to the GSX graphics system. Owners of 464s and 664s do not have this facility, and if they need to use graphics then they have four choices. The simplest, but most expensive, alternative is to flog the old machine and buy a 6128 or a PCW. Get a 6128 if you want carry on using your old software, or a PCW if you would prefer the speed and convenience of the RAM disk.

The second choice is to upgrade the 64K micro. The Dk'Tronics 64K RAM pack adds the extra memory and also includes the software that gives your old machine the same banked memory and commands as a 6128. It is very effective, but leaves you with the problem that you still need a set of CP/M Plus master disks. At the time of writing, there is no legal solution to this problem as the system is only sold with the machines for which it is intended, and Digital Research do not like people copying their disks to give to friends.

Another possibility is to write your own graphics routines, either in assembler or in a compiling language. It is perfectly possible to peek and poke into all parts of memory, including screen memory, and create graphic effects this way. The catch is that the resulting program will only work on 464 and 664 machines, as all the other Amstrads have different memory maps when running CP/M. As the program is therefore machine-specific, there seems to be little point in using the CP/M system at all.

The last alternative is to abandon CP/M altogether and use the native graphics routines and non-CP/M graphics packages – of which there are plenty on the market. In fact, some of these are faster, more flexible and easier to use than the leading CP/M graphics applications DR Draw and DR Graph.

GSX – the Graphics System eXtension

The structure of the graphics system is essentially the same as that of CP/M itself. It is divided into two sections – the GDOS (Graphics Device Operating System) and the GIOS (Graphic Input Output System). The GDOS, like the BDOS, is standard in all versions and intercepts GSX calls from transient programs. The GIOS is machine- and device-specific, turning each GDOS function into screen, printer or plotter output.

Under GSX, every device is assumed to have a coordinate system that runs from 0 to 32,767 along both axes. All positional data is passed to the GDOS in this form where it is scaled down to suit the particular device. Colour-handling is similarly flexible, as colours are defined by giving the intensity of red, green and blue needed for each. This means that the images produced on different devices are identical for all practical purposes.

GDOS occupies 2K of space immediately below the resident CP/M system. Once installed, it intercepts all operating system calls, processing the graphics ones and passing the other function calls through to the BDOS. All RSXs (Resident System eXtensions) – of which GSX is a prime example – work in this way. At the end of a

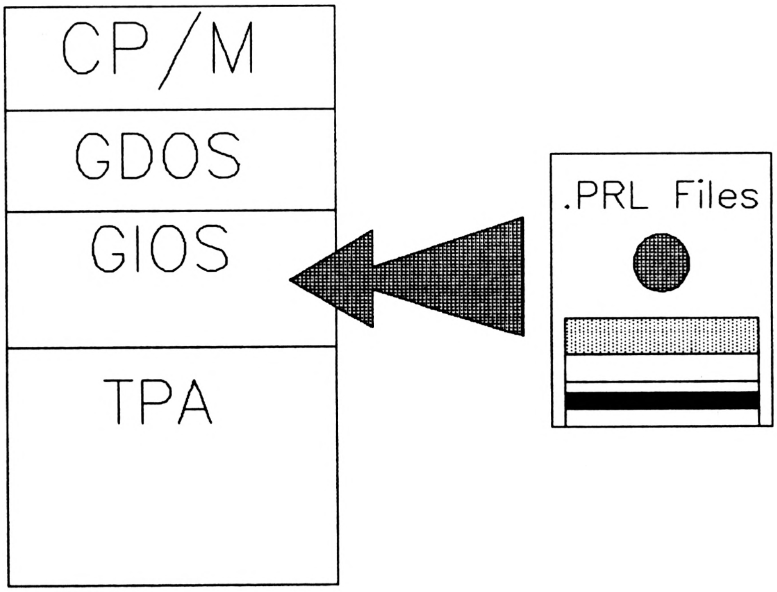


Figure 7.1 *GSX memory map*

program that uses GSX the system will remain in memory until a cold start is performed. A warm start leaves GSX intact.

The GIOS is organised as a set of device drivers, each containing the routines needed for a particular peripheral and each stored separately on the distribution disks. When a GSX program is running, only the driver needed at that moment is brought into memory, but the others must be present on the working disk. This is essential as some of the drivers need a lot of space – the typical printer driver is around 12K in length – and the GSX system must leave room for the transient program that uses it.

The drivers are identified by .PRL filetypes, and the set that you have will depend upon your machine. The 6128 version of Plus includes drivers for Epson printers (DDFXLR7.PRL), the Amstrad DMP1 printer (DD-DMP1.PRL) and the screen in each of its modes (DDMODE0.PRL, DDMODE1.PRL and DDMODE2.PRL). The PCW models have drivers for the printer in low-and high-resolution modes (DDFXLR8.PRL and DDFXHR8.PRL). Drivers for HP-compatible pen-plotters (DDHP7470.PRL) and for Shinwa printers are also available. If you own, or are thinking of getting any other sort of printer or plotter to use with the CP/M system, then be sure that a driver is supplied with it.

The set of drivers that will be accessed by GSX on your system depends upon the ASSIGN.SYS file. This gives a list of the drivers, with GSX identifying numbers and filenames. The driver that needs the largest amount of memory must be given first, as this determines the space that will be reserved for the drivers. By convention, numbers beginning with 0 are screen drivers, while 1 indicates a plotter, and 2 a printer.

The standard assignments for a 6128 are:

```
21 a:ddfxlr7    ; Epson 7 bit printer
11 a:ddhp7470   ; Pen plotter
01 a:ddmode2    ; Screen in mode 2
02 a:ddmode1    ; Screen in mode 1
03 a:ddmode0    ; Screen in mode 0
```

The default ASSIGN.SYS file on a PCW lists the drivers for the printer and monitor only. If you wish to add other devices, the file can be rewritten with any text editor.

Some applications programs are supplied with a suitable GSX system already installed, but more often than not you will have to install it yourself. Any good commercial software will include an installation program that will take you through the stages of this process.

When you have written and compiled programs of your own that are to use GSX extensions, you will need to add the system after the link stage of compilation. To do this, you must have on the same disk the following files:

Your own program
GENGRAF.COM
GSX.SYS
ASSIGN.SYS
The .PRL files listed in ASSIGN.SYS

The command 'GENGRAF program_name' will attach the GSX system to the program.

GSX-using programs can be written in assembler, but it is a complex and highly specialised job. A far more satisfactory alternative for most programmers is to use Cbasic which has a set of commands that call up the GSX functions. We will return to programming in the next chapter, after we have had a look at the graphics application programs DR Draw and DR Graph.

DR Draw

DR Draw is intended for use in producing business graphics such as illustrated reports, file covers, advertising leaflets, letterheads, charts and diagrams. It was used to create the very crisp illustrations in the DR Draw manual, and many of the diagrams in this book. It is not intended for producing any statistical displays – these are handled far better by the companion piece, DR Graph.

The package consists of a well-produced and clearly written manual, a machine-specific leaflet to show you how to get it all running, and a disk full of files. The main DRAW.COM program is supported by a whole bank of overlays and a number of different font (typeface) files. There are also the utilities that you will need to install the package for your machine.

DR Draw can be used with either a 6128 or a PCW, though 6128 owners will find that they have to do rather a lot of disk switching when they use the program. This is far less of a problem on the PCWs where all the overlays and driver files can be stashed in the RAM disk for instant access, leaving disk space free for picture and font files.

The program is menu-driven, and is best operated with a mouse. The keyboard can be used, but if you are at all heavy-fingered – or simply used to hitting RETURN at the end of an operation – you could find that you keep leaping out of the drawing screen back to the main menu! Most selections are made by moving the cursor across the screen until it is level with one of the options listed along the top line. Pressing the space bar, or the PICK button on the mouse then takes you into that option. There will often be a second menu to work through at this stage. DONE, or the RETURN or ENTER key, takes you back to the level of the previous menu. Keyboard cursor control is relatively slow, and the constant shuffling back and forth across the screen can become tedious. All in all, the program controls are not idiot-proof, and are barely user-tolerant at times, but with practice you can get used to its ways of doing things and produce some interesting displays.

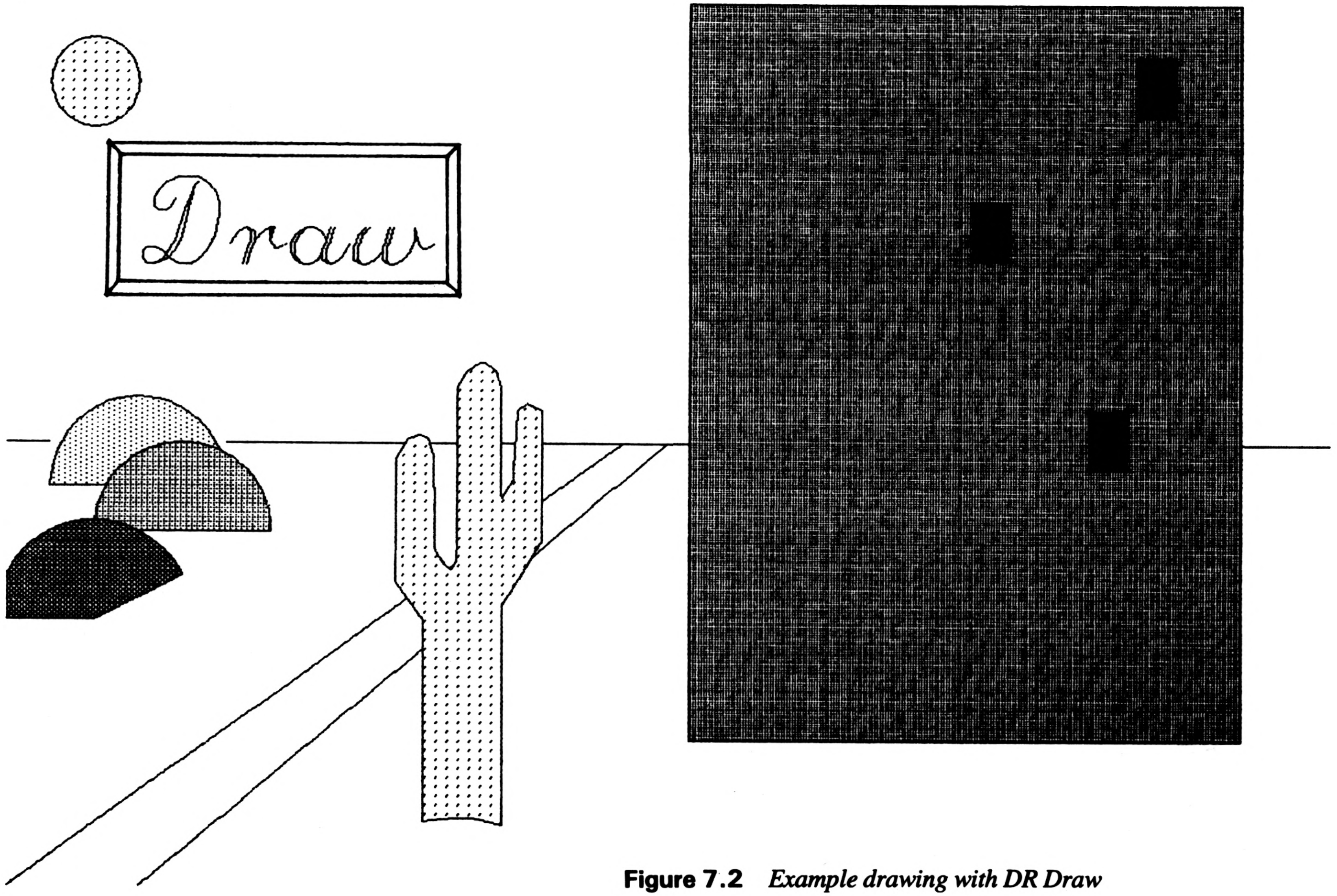


Figure 7.2 *Example drawing with DR Draw*

Pictures are built up out of a selection of drawing 'primitives' – line, bar, circle (open or filled), arc or segment and polygon. Polygons can be any closed shape, and in practice can have an almost unlimited number of vertices, allowing very complex images. Line styles – solid or a variety of broken patterns – and fill colours or shading densities can be selected at will, and changed afterwards if needed. Text can be written in simple machine characters or in one of the supplied fonts. Up to three fonts may be accessed at any time, out of a total of eight supplied with the suite. Text size is infinitely variable, though the relation of height to width remains constant and text is always written horizontally across the screen or paper.

One of its most striking and attractive features is its enormous flexibility over scale and layout. It is so simple, once you have got the hang of it, to move and alter an image, making large or small adjustments, until it is exactly the right size and in exactly the right place. The variety of font styles, and the possibility of continuous scaling with all but machine typefaces, adds interest even to simple textual presentations. In theory, the image that you see on the screen will be almost identical to the one that is transferred to paper by printer or plotter. In practice this is not quite so, as text does not always appear in the same scale as other graphics, and you may need to adjust your design after a trial print out.

This flexibility of scale is of course a direct result of the GSX coordinate system. When your axes run from 0 to 32767, then copying with a pixel move is no problem. Another, and less attractive result of the GSX system is that the program runs slowly. Points have to be recalculated every time they are referenced and even the simplest operations have to travel the tortuous routes through GDOS and GIOS. Owners of 6128s who have used non-CP/M screen designers will certainly notice the difference in speed.

When everything works, the results can be impressive. The catch is that sometimes things don't work. The version available at the time of writing – nearly three years after its first release and several months after its issue on Amstrad 3-inch disks – is not fully debugged and it is quite possible to crash it. Those functions that access the disks seem to be most prone to error, which is very bad news. It is quite likely that from time to time you will find that the program hangs when you are trying to save a picture file. At other times, you will think that you have saved a picture, but discover when you recall it that some of its elements are missing. There are other irritating little quirks too, like the way that text sometimes appears below, rather than in, the area to which you were trying to move it.

The sheer scale of the program means that you have to be efficient in your use of disk space. Picture files are not particularly compact – the simple diagram in Figure 7.1 took 3K, and the sketch in Figure 7.2 took 7K – and when working on a picture, you will need space for the current .PIX file, the last version of it (.BAK) and the working copy (.TMP). This could mean well over 30K for a moderately complex design. As the program requires that some utility files are on the same side of the disk as the picture file, you don't have much space to play with. It is the same problem that we met earlier with Cbasic and Pascal, and reflects the fact that CP/M programs were often originally designed for use on professional systems with several drives or with a hard disk. PCW

512 owners, of course, with their second, high-capacity, drive do not have the same problems in this respect.

When the last few bugs are cured, DR Draw will offer an efficient display utility for anyone who is prepared to spend time learning how to get the best out of it. The current version can best be considered a professional tool for those professionals who get paid for their time whether they do anything useful or not. The self-employed user may well resent the hours lost through those occasional but highly damaging bugs.

DR Graph

DR Graph is designed to produce the kind of clear, meaningful statistical displays that can add so much to a presentation or a report. It can be used to create line, bar, scatter, stick or step graphs and pie charts. These can be output singly or with up to four on a page. Four different font styles and great variability of character size are also available to highlight the displays.

The package is similar to DR Draw, comprising a well-produced manual, an installation leaflet and a disk crammed with files. The program is largely controlled by keyboard selection from numbered menus. Data is entered, and display options selected, in a series of screens where movement is by simple keystrokes. The layout of the screens is well organised, and the flow of the program is clear and simple to follow. It therefore takes far less time to learn how to use this program than it does with DR Draw.

There are plenty of options, allowing fine tuning of your displays and a wide variety of graphics effects, but the core of the program is essentially simple. This means that you can produce adequate graphs almost on first contact. These can be easily altered afterwards if a different or a better quality of graph is required. Individual graphs can be drawn together into multiple displays, and while you lose a little of the clarity and fine detail of the larger scale outputs, multiple displays do allow sets of data to be compared more easily.

As you can see from the sample graphs in Figure 7.4, the statistics that you are displaying do not have to be business ones. DR Graph is flexible enough to handle any type of data, though it is geared to commercial applications. Visicalc and Supercalc files can be handled directly by DR Graph – a very attractive bonus to those who are already using one of these spreadsheet packages.

The infrequent, but irritating bugs that spoil DR Draw appear to be quite absent here, though the problem of disk space remains for 6128 owners. With the PCWs, there is no such problem as all the utility files can be transferred to RAM disk, and once the program is running it does not need to access the drive again except to store and recall picture files.

As a rough guide to file size, each of the four sample graphs shown here took 4K of disk

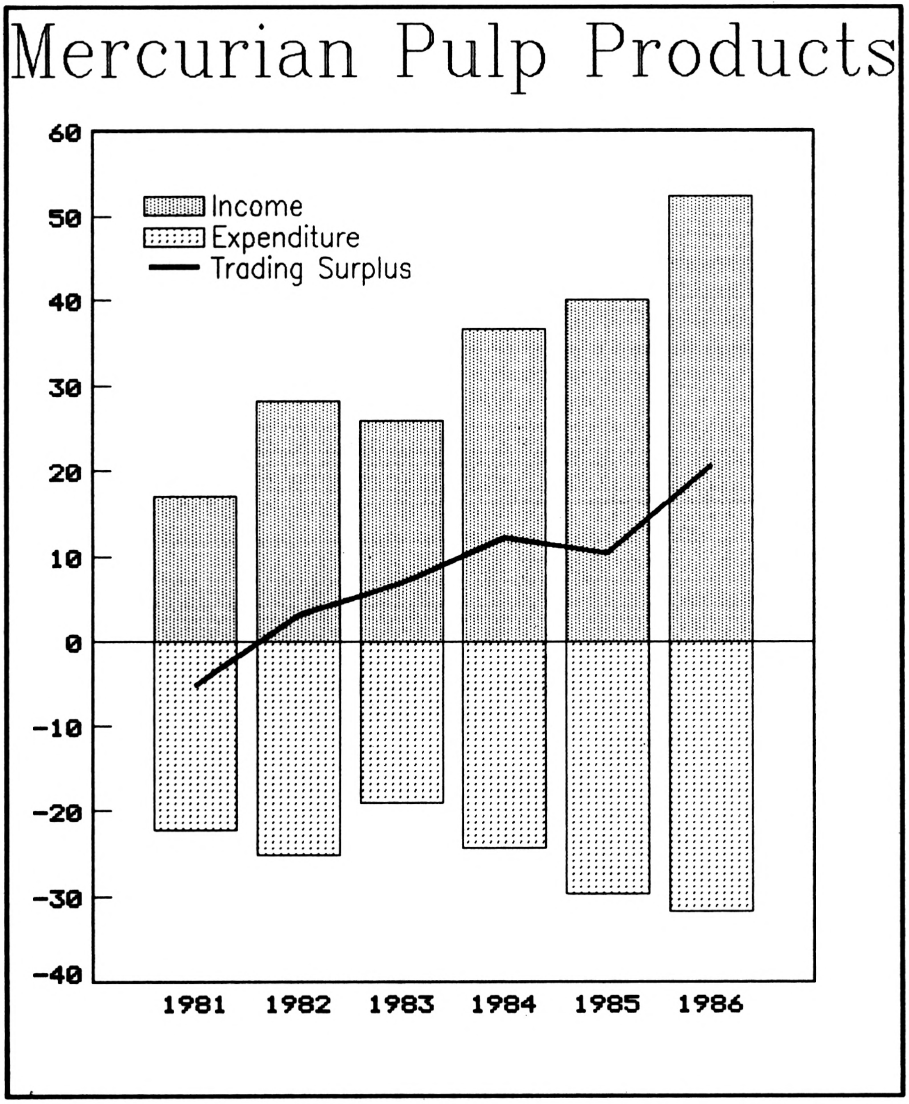


Figure 7.3 *Example graph with DR Graph*

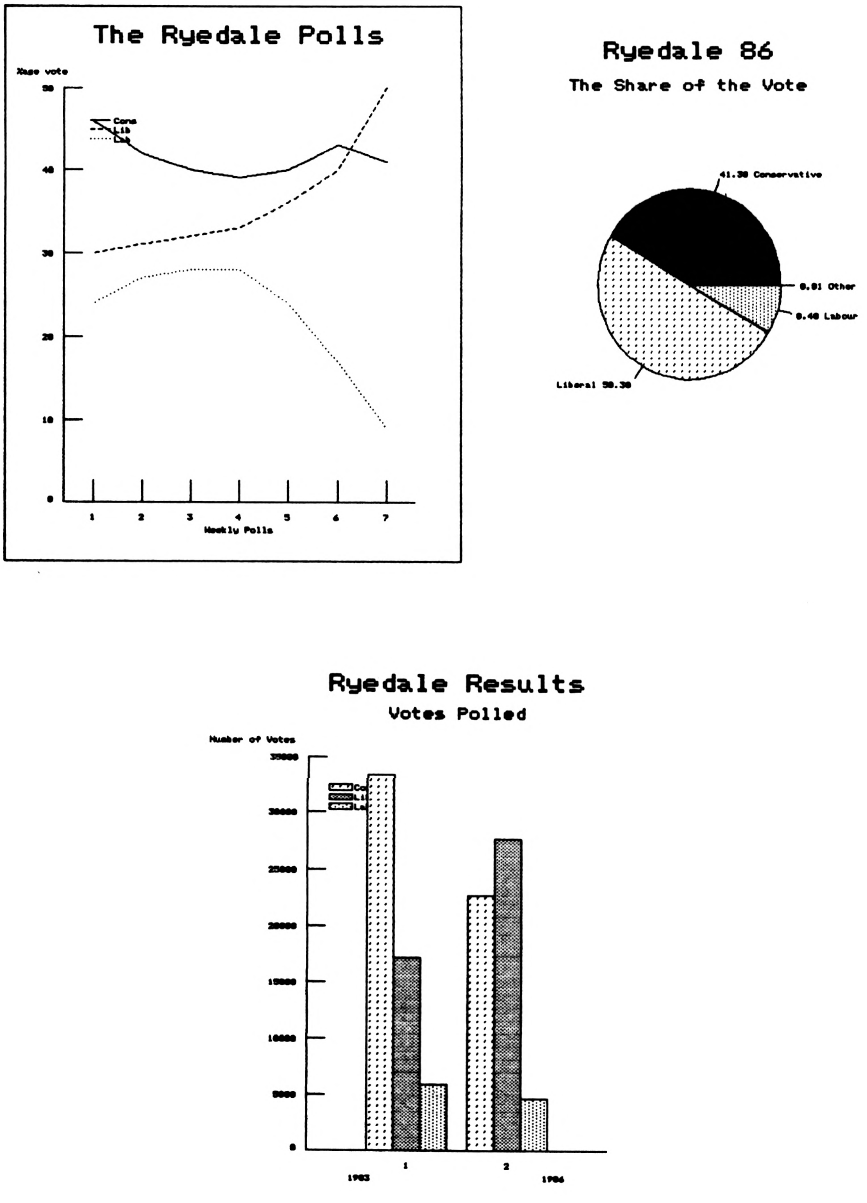


Figure 7.4 *Statistical results in several forms produced by DR Graph*

space to store, though notice that they use relatively simple layouts and none of them has the quantity of data that might be needed for a full annual analysis.

The program is significantly faster than DR Draw, except for printer output which remains as slow as ever. This shouldn't matter too much as the printer needs no further attention once it has started, and will happily plod along by itself while you do something entirely different.

Even at £50, DR Graph represents reasonable value for money, especially if it is to be used alongside Visicalc or Supercalc, or another compatible spreadsheet. It is robust and easy to use, and the printed output is impressive. If the professional finish that it gives to your presentations will help to secure an extra sale or two, or convince the bank manager over that loan you wanted, then it will soon pay for itself.

CHAPTER 8

GRAPHICS PROGRAMMING

Digital Research's Cbasic is probably the most satisfactory means of creating graphics programs to run on a CP/M Plus system. It offers an efficient and simple set of graphics commands, and the rest of the language is a fairly standard Basic. The results are not blindingly fast, but that is more a reflection on CP/M than on Cbasic. The output to the printer is so slow that at times you may think that the program has hung! Owners of 6128 may prefer to drop out of CP/M altogether, and stick to standard Locomotive Basic for their graphic work, unless they have a particular need for the long-term portability of CP/M. PCW owners do not have a similar simple alternative as Mallard Basic has no natural graphic capabilities. GSX routines can be added in the same way that any RSXs can, but their use requires a full understanding of the system and painstaking attention to detail.

There is an alternative for PCW (and 6128) owners who like to program in Pascal or C. Hisoft produce versions of both these languages with GSX functions accessed through Logo-style commands. The results are equally as good as those achieved through Cbasic, though the speed is inevitably much the same. The packages are slightly cheaper than Cbasic, but the manuals are definitely not for beginners. If you understand the languages then they are excellent value. If not, then you must add the price of a good book to the total cost.

Cbasic graphics commands

Cbasic graphics commands reflect GSX function calls both in style and scope. They may be divided into three sets: device-based commands, key drawing commands and text-handling commands.

Device-based commands

These commands get and set the parameters of the output device:

GRAPHIC OPEN n takes the driver for the device number n into memory, and prepares for output.

GRAPHIC CLOSE ends the use of a device.

CLEAR clears the screen.

WINDOW imposes the user's own co-ordinate system on the device. The default is 0 to 1 in each direction, but any scales may be used.

VIEWPORT selects the portion of the device that is to be used for graphics – very similar to creating a WINDOW in Locomotive Basic.

BOUNDS sets the aspect ratio – the relationship between the X and Y scales. On a PCW monitor the ratio needs to be around 0.73 to 1 if circles are to circular and squares really square. The printer's BOUNDS need to be about 1.2 to 1 to achieve properly proportioned graphics.

CLIP can be used to prevent the graphics overrunning the allotted area.

ASK DEVICE tells you the physical X,Y ratio of the current device.

COLOR COUNT and STYLE COUNT reveal the range of display options.

Key drawing commands

PLOT draws a line – not the single point which you might expect. As the 'line' could be one point long, the command can be used for both lines and points. The nature of the line can be set by LINE STYLE.

BEAM offers another way of drawing a line. When the BEAM is turned on, a line will be drawn between the current graphic position and the next position to which it moves. When it is off, the graphic cursor can move invisibly.

SET and ASK POSITION will change or return the current graphic coordinates.

MAT PLOT joins together an array (or MATrix) of points. The very similar command

MAT FILL joins points, creating a closed shape, which is then filled with the current COLOR.

MAT MARKER is a third array-using command. The markers that are drawn may be any of the set [. + * O X] and can be scaled to different sizes.

MARKER HEIGHT and MARKER TYPE define the nature of the markers.

Text-handling commands

GRAPHIC PRINT and GRAPHIC INPUT display and accept text at the current graphic position.

JUSTIFY can be used alongside the PRINT command to tidy up display.

CHARACTER HEIGHT and TEXT ANGLE are commands that give flexibility and variety to the output of text. They only work with the printer and do not alter the display on the monitor. Alternative character sets are not available.

Cbasic graphics use

As you can tell, the essential graphics functions are all there, but you may miss the higher facilities that most good Basics provide. A particular lack is the absence of a circle function. Digital Research try to correct this by including a sample circle routine in their manual. An alternative, faster and more flexible, circle routine is given below as an example of graphic Cbasic programming.

Good disk space management is essential for happy programming with Cbasic graphics. You need a total of almost 150K of utility files ready at hand. PCW users can stash must of the utilities into RAM and make their lives much easier, but the poor 6128 owner must have all or most of these on the work disks. The files can be grouped according to use in the four stages of producing a program.

Stage 1: The text file. [ED filename.BAS] Use your favourite text editor to produce the file. The editor could be called up from a second disk, but it is more convenient to have it on the work disk where you store your text files.

Stage 2: Compilation. [CB80 filename] CB80, its three overlays and the core routines held in GRAPHCOM.BAS will take 59K, leaving 110 for text files and the .REL files that are produced.

Stage 3: Linking. [LK80 filename] LK80, CB80.IRL (a library of essential functions) and LIB.COM (the library manager) are needed, and you may also want to store a debugger. Total taken, between 50 and 60K – more if you build up your own libraries and use the linker to its full.

Stage 4: Adding the GSX routines. [GENGRAF filename] For this you need GENGRAF, GSX.SYS, ASSIGN.SYS, and the drivers for each device in the system. With the normal PCW set-up, these take 25K in all. It is quite feasible to group Stages 1 and 2 on one side of a disk, and the remaining two on the other, but you do have to tidy the disk regularly, removing unwanted files and transferring completed programs to permanent storage elsewhere. You have more working space if each stage of the process is managed on a separate side, but it does involve rather more disk changing. Using one disk or more, you will also need PIP at hand somewhere – preferably on every side for greatest convenience.

Arcs and circles

The following program should give some idea of the flavour of Cbasic, and if you do use the language, I hope that the routines may be useful in your own programs.

It produces four types of curved shape – simple arcs, solid segments (or rather slices), open rings and filled circles. The position and size – and the shapes of arcs and segments – are infinitely variable. The program uses a 0 to 100 coordinate system, but this can be set to any range that suits you. The actual ranges of X,Y values that are set in the WINDOW command do not affect the degree of detail that can be displayed on screen or printer. Any kind of curve drawing will always be slowed down by the need to calculate the x and y coordinates of the points on the circumference – and the slowest part of these calculations involves the use of sines and cosines. To speed matters up, tables of sine and cosine values are calculated once at the beginning of the program, and then referred to later when needed. The table creation is itself speeded up by only calculating the sines and cosines of the angles between 0 and 90 degrees (0 to 1.57 radians), and copying these values to the other three-quarters of the table. The absolute values of the sine and cosine of an angle between a line and an axis are always the same, whether the line points up, down, left or right. All that changes is the positive/negative sign. For example, SIN (45) is the same as –SIN (135), –SIN (225) and SIN (315).

The MAT PLOT and MAT FILL commands, which are used to draw the shapes, work in almost identical ways. The format of the command is:

```
MAT PLOT num: x,y
```

where num is the number of points in the arrays, and x and y are arrays of coordinates. This corresponds to the structure of GSX function calls where drawing functions are again based on arrays rather than single points. As the coordinate values passed to the GSX calls have to be converted by Cbasic from the user-defined system to GSX's Normalised Device Coordinates in the range 0 to 32767, there is always a delay before the image appears.

```
REM ARC AND CIRCLE FUNCTIONS DEMONSTRATION PROGRAM
REM to include in other programs use only the definition sections

  % INCLUDE GRAPHCOM.BAS : REM essential common variables

     DIM C(63): REM table of cosines
     DIM S(63): REM table of sines
     DIM X(63): REM array of X and...
     DIM Y(63): REM Y coordinates
```

```
PRINT "PLEASE WAIT - MEDITATING TRANSCENDENTALS"
REM in maths jargon sines and cosines are transcendental functions
I=0
FOR N=0 TO 15
  C(N)=COS(I)           :REM 0 to 90 degrees or 0 to PI/2 radians
  S(N)=SIN(I)
  C(31-N)=-C(N)         :REM 90 to 180 degrees - only cosines negative
  S(31-N)=S(N)
  C(N+32)=-C(N)         :REM 180 to 270 degrees - both negative
  S(N+32)=-S(N)
  C(63-N)=C(N)          :REM 270 to 360 degrees - only sines negative
  S(63-N)=-S(N)
  I=I+0.1               :REM calculations actually in radians
NEXT N

DEF CALC
  FOR I=0 TO 63
    X(I)=XC+R*C(I)      :REM X displacement from centre
    Y(I)=YC+R*S(I)
  NEXT I
  RETURN
FEND

DEF ADJUST (ARCEND,START)
  ST=START
  NUM=ARCEND-START
  FOR I=0 TO NUM        :REM move all values to start of array
    X(I)=X(ST)
    Y(I)=Y(ST)
    ST=ST+1
  NEXT I
  RETURN
FEND

DEF ARC
  IF START <> 0 THEN CALL ADJUST(ARCEND,START)
  NUM=ARCEND-START
  MAT PLOT NUM: X,Y
  RETURN
FEND

DEF SEGMENT
  IF START<>0 THEN CALL ADJUST(ARCEND,START)
  NUM=ARCEND-START
  NUM=NUM+1
  X(NUM)=XC             :REM make centre the last point
  Y(NUM)=YC
```

```
    MAT FILL NUM:X,Y
    RETURN
  FEND


REM demonstration starts here

  DEF LIMITS
    PRINT ''ENTER START ANGLE IN RADS (0 - 6.28)''
      WHILE START <0 OR START>6.28 :REM checked INPUT routine
        INPUT START$
        WHILE START$ = '' ''
          INPUT START$:START = VAL(START$)
        WEND
      WEND
    START = START * 10: REM DERIVE ARRAY START NUMBER FROM
    ANGLE
    PRINT ''ENTER END ANGLE''
      WHILE ARCEND <0.1 OR ARCEND >6.27
        INPUT ARCEND
      WEND
    ARCEND = ARCEND * 10
    RETURN
  FEND


REM loop to repeat

  AGAIN = 0
  WHILE AGAIN = 0
    XC = 0:YC = 0:R = 0:START = -1:ARCEND = 0: REM reinitialise
    CLEAR
    CENTRE$ = ''OPEN'': REM DEFAULT VALUE
    PRINT ''SELECT ARC, SEGMENT, RING OR CIRCLE (A/S/R/C)''
    INPUT A$
    WHILE MATCH(A$,''ASRC'',1) = 0 :REM check input
      INPUT A$: A$ = UCASE$(A$)
    WEND
    IF A$ = ''S'' OR A$ = ''C'' THEN CENTRE$ = ''FILLED''
    PRINT ''ENTER X COORDINATE OF CENTRE''
    WHILE XC<2 OR XC>100
      INPUT XC
    WEND
    PRINT ''ENTER Y COORDINATE OF CENTRE''
    WHILE YC<2 OR YC>100
      INPUT YC
    WEND
    IF A$ = "A" OR A$ = "S" THEN CALL LIMITS
    PRINT "ENTER RADIUS"
    WHILE R<1 OR R>50
      INPUT R
    WEND

REM calculate x,y plot values

    CALL CALC
    SCREENCODE = 1    :REM device driver codes in ASSIGN.SYS
    PRINTERCODE = 21
    DEST = SCREENCODE: REM default to screen
    PRINT "OUTPUT TO SCREEN OR PRINTER (S/P)"
    INPUT D$
    WHILE D$<>"S" AND D$<>"P"
      INPUT D$:D$ = UCASE$(D$)
    WEND
    IF D$ = "P" THEN DEST = PRINTERCODE
    CLEAR

    GRAPHIC OPEN DEST
    SET CLIP "ON" : REM keeps all graphics within window area
    SET WINDOW 0,100,0,100 :REM set coord system
    SET BOUNDS 0.75,1 : REM set aspect ratio - default to screen
    IF D$ = "P" THEN SET BOUNDS 1.2,1
    IF A$ = "A" THEN CALL ARC
    IF A$ = "S" THEN CALL SEGMENT
    IF A$ = "R" THEN MAT PLOT 63:X,Y
    IF A$ = "C" THEN MAT FILL 63:X,Y
    IF D$ = "P" THEN GRAPHIC CLOSE :REM no output to printer before
    close
    INPUT "ANOTHER ONE ? (Y/N) ";ANS$
    IF UCASE$(ANS$) = "N" THEN AGAIN = 1
    IF D$ = "S" THEN GRAPHIC CLOSE
  WEND
STOP
END
```

To produce a working .COM file, start by creating the text file. It must have a .BAS filetype:

```
ED CIRCLES.BAS
```

Transfer the file to the disk with the compiler and overlays, and call up CB80:

```
CB80 CIRCLES
```

Do not give the filetype – .BAS is assumed, but do use the (B) option as it will suppress the output of the list to the screen, displaying only those lines containing errors. If you allow the listing to occur, the error messages will scroll off the screen faster than you can read them – and there will usually be at least one odd error, unless you are an extremely accurate typist.

CB80 will produce a .REL file – a relocatable module – ready for the linker program. Move this to the disk with LK80. No filetype is needed here either:

```
LK80 CIRCLES
```

At this stage, there is a dramatic change in the size of the file. Both the .BAS and .REL files were about 4K in length. Linking adds a further 16K, bringing it up to 20K. The amount added varies with the library routines that the program needs, and many of these are almost always essential. A much larger .REL file will not necessarily grow by more than 16K during linking.

The .COM file output by LK80 would be ready to run if it didn't have graphics. As it does, it needs the final processing to add the GSX header and routines. This increases the file's size by a further 2K. Even if you are only producing short, exploratory programs, you must have something like 40K of free space available on the disk.

GSX programming

An introduction to CP/M is not the place in which to go deeply into the techniques of using GSX functions within machine-code programs. There simply isn't space here for all the information that you would need. It is useful, though, to have a look at the range of functions available and the way in which they are accessed. Understanding more about the possibilities and limitations of the system will help you to get more out of Cbasic and graphics packages.

The GSX functions

Many of these are device-dependant, and this list only includes those that are implemented on the 6128 and PCW monitors and printers.

1 OPEN WORKSTATION This loads the relevant device driver and prepares the system to use that device.

2 CLOSE WORKSTATION If the current device is the screen, it is cleared. If it is a printer, then the printout is done.

3 CLEAR WORKSTATION This clears the screen or performs a line-feed and print out.

4 UPDATE WORKSTATION This executes any pending graphics commands, but without the clear-screen or line-feed.

5 DEVICE-SPECIFIC OPERATIONS These are identified within the call. Many of

them you will recognise as having the same effect as the normal control characters:

1 Inquire addressable character cells – how many rows and columns
2 Enter graphic mode – if different from standard mode
3 Exit graphic mode
4 Cursor up
5 Cursor down
6 Cursor right
7 Cursor left
8 Home cursor
9 Erase to end of screen
10 Erase to end of line
11 Locate cursor at row and column
12 Display text at cursor
13 Reverse video on
14 Normal video
15 Get cursor position
16 Auxillary input from mouse, joystick or graphic tablet
17 Hardcopy – copy screen to printer
18 Locate graphic cursor
19 Erase graphic cursor

6 POLYLINE draws lines between an array of points.

7 POLYMARKERS displays markers at an array of points.

8 TEXT writes text at a given position.

9 POLYGON displays a filled polygon defined in an array.

10 CELL ARRAY draws a table of boxes

11 GENERALISED DRAWING PRIMITIVE which may be selected from:
1 Bar
2 Arc
3 Pie Slice
4 Circle
5 Graphic characters

12 SET CHARACTER HEIGHT with the size given in device units – which means rasters (points of light) on a monitor or dot sizes on a printer

13 SET TEXT ANGLE Identical to the Cbasic command, except that figure are given as integers corresponding to tenths of degrees (0 to 3600).

14 SET COLOR by selecting a specific intensity of red, blue and green to be allocated to a given colour index.

15 SET POLYLINE TYPE as in Cbasic's LINE STYLE.

16 SET POLYLINE WIDTH with the width given in device units.

17 SET POLYLINE COLOR selects a colour index for line drawing.

18 SET POLYMARKER TYPE as in Cbasic's MARKER TYPE.

19 SET POLYMARKER SCALE as in Cbasic's MARKER HEIGHT.

20 SET POLYMARKER COLOR selects the colour index for markers.

21 SET TEXT FONT selects the character set.

22 SET TEXT COLOR selects the colour for future text printing.

23 SET FILL STYLE – hollow, solid, half-tone or hatch. (Printer only)

24 SET FILL STYLE INDEX sets the pattern for half-tone and hatch fill.

The GSX functions are all accessed in the same way through a call to the BDOS function 115. This is not normally resident, but is added as part of the GSX system. When the call is made, the C register is, of course, loaded with 115 and the DE pair contains the address of a parameter block. This holds the addresses of five arrays in which data is passed between the graphics system and the calling program. These arrays are conventionally called CONTRL, INTIN, PTSIN, INTOUT and PTSOUT.

CONTRL has a common structure, though the meaning of most of its elements is dependant on the operation that it controls.

CONTRL(1) holds the opcode – the number that identifies the function
CONTRL(2) – number of points in the array PTSIN
CONTRL(3) – number of points in PTSOUT
CONTRL(4) – number of elements in INTIN
CONTRL(5) – number of elements in INTOUT
CONTRL(6 on) – opcode-dependant. The number of elements and the way that they are used varies considerably.

INTIN holds the input parameters – the values that select colour, fill patterns, line type and such. 'INT' here refers to INTeger, as all values are such.

PTSIN is the array of points, set in x,y coordinates. These are each given as two-byte integers in the Normalised Device Coordinate range of 0 to 32767. The array will therefore be four times the number of vertices in length.

INTOUT returns parameters (as integer values) to the calling program. The most substantial INTOUT array is that returned by the OPEN WORKSTATION call, which gives all the relevant specifications of the current device.

PTSOUT returns other numeric values. Again the most significant data is produced by the OPEN WORKSTATION call, where minimum and maximum character heights and line widths are returned in PTSOUT.

The amount of data that has to be set in CONTRL and the other arrays depends entirely upon the function that is being called. Take, for example, the POLYLINE and SET POLYLINE LINETYPE functions.

POLYLINE needs the opcode 6 in CONTRL(1) and the number of points in CONTRL(2). The x and y coordinates of each point are of course needed in PTSIN. No other data is required in this call, though any changes in width, style or colour must have been set separately by other calls. On return from the function, CONTRL(3) will be set to 0.

SET POLYLINE LINETYPE needs the opcode 15 in CONTRL(1), 0 in CONTRL(2) and the linestyle index number in INTIN(1). On return, there will be 0 in CONTRL(3) and the number of the linestyle that was used in INTOUT(1). If the two linestyle numbers are different, it shows that the one requested was out of range for that device and the default linestyle was used instead.

Full details of the GSX functions are given in the *GSX Programmer's Guide* published by Digital Research. It should be stressed that successful programming at this level requires a detailed technical understanding of the devices used as well as a good grasp of assembly language and the GSX functions.

CHAPTER 9

CP/M SOFTWARE

Introduction

As we noted at early on in the book, one of the great attractions of CP/M is that you are not limited to the marketplace for your software. Join the CP/M User Group and you will have access to a whole library of public domain software. The library catalogue is a closely typed book of over 140 pages, listing around 300 disks, each with an average of well over 200K of files! Its contents are very much a mixed bunch, nearly fifteen years of the accumulated output of CP/M professionals and amateur enthusiasts from the United States, Europe, Australia and elsewhere.

There are games ranging from the original Crowther and Woods 'Colossal Adventure' (the granddaddy of all adventure games, and it takes three disks), 'Dungeons and Dragons' (another three-disk set), and other text adventures, through Othello and Chess and other boardgames, down to simple word and keyboard reaction games written in one or other type of Basic.

The languages in the library include various dialects of Pascal, C, Basic, Lisp, Forth, Algol and Pistol. As well as compilers for these, there are also document files explaining their use, example programs and useful routines – quick sorts, random number generators, text formatters, number crunchers and file-handling aids. Most of these are in Pascal or C, the two most popular languages. Some of the less common languages, for example PISTOL the Portably Implemented Stack Oriented Language, are not readily available other than through the Group.

CP/M SOFTWARE

CP/M utilities figure very largely among the files. This is hardly surprising as anyone who uses CP/M (or any other operating system for that matter) for any length of time is likely to want to improve some features of the system. The utilities are mainly upgrades of the normal command set and additional file and directory handling routines; others are designed to transfer data between different machines, or to simulate other operating systems. Other programmers' utilities include many modem and communications programs, and a variety of improved assemblers and disassemblers. There are also clock and calendar generators (one has a pin-up too!), as well as home control programs and many varied others.

Business software does have its place in the library, and you can find several accounts packages, databases, mail-merge, inventory and point-of-sale systems and other standard types here. There is even a complete suite of programs – Businessmaster II – taking five disks and covering everything from Depreciation Calculator, through Raw and Finished Goods Inventories, to Federal Tax programs. (Yes, it's American and would need some tailoring for British businesses.) What you also find are the more unusual packages produced for special situations. Do you need, perhaps, a program to design a balanced feed for your cows? Or analyse the energy performance of a building?

The educational software side is rather limited. There are some Maths practice programs, text exercises and typing tutors and a few assorted others. At a higher level there are statistical and scientific tools, the Yale Catalogue of Bright Stars and even a financial modelling program called THEFED. Overall, the selection probably reflects the academic/professional/enthusiast nature of most CP/M users and the restricted graphics capabilities of most CP/M systems, but should certainly include items of interest for many new users.

At the time of writing, the Group charge a £3 per disk copying fee – with you providing the disk. Annual membership – currently £7.50 – is the only other cost. This gives you access to the library and can help to put you in touch with other users in your area, or at the other end of a modem line. It is very much an enthusiasts' club, not a commercial organisation. As such it welcomes contributions of programs for its library and articles for its journal, and encourages members to be active in their local areas.

CP/M User Group (UK) is at 72 Mill Road, Hawley, Dartford, Kent DA2 7RZ.

Commercial software

At the time of writing there is a good range of CP/M applications software available on the market, though almost all of this is business software or programming languages. The most notable exceptions are the games 'The Hitch-Hiker's Guide to the Galaxy' and '3D Clock Chess'. As most PCW owners are probably business or professional people or programmers, and all 6128 owners can use the standard Amstrad games and utilities, this situation is likely to remain much the same in future. What we can reasonably expect is a significant increase in the variety of specialist programs designed to be used in particular types of businesses – packages for video hire shops, estate agents, newsagents, garages, dentists and so on. Some of these already exist in other CP/M formats, and it is largely a question of time before they are prepared for sale on 3-inch disks.

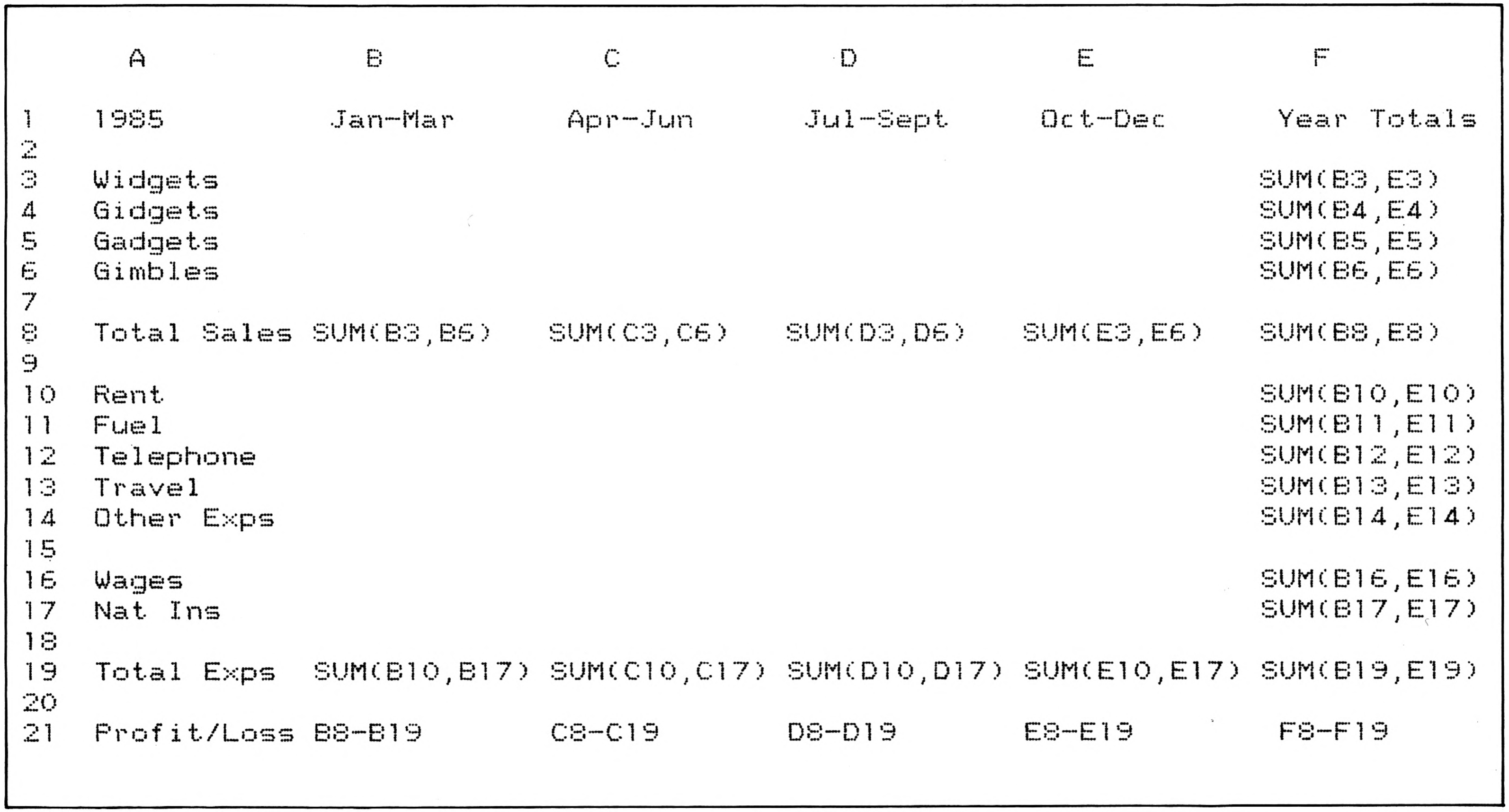
A key issue in the decision to buy a software package must be cost. When you are going to have to pay anything between £50 and £150 for a package, you have to be reasonably sure that it will be worth it. There are then two aspects to this: is the package the best value in its class, and does it make sense to use the computer at all for the particular job you have in mind? There is another, and often greater, cost which may not be immediately obvious. With any package, you are going to spend time learning how to use it, and more time setting it up for use in a business. Where it is necessary to transfer a lot of data from ledgers or card files, then there could be many hours of typing involved. It can add up to several hundred pound's worth of managerial and secretarial time. If at the end of the process, the software begins to save valuable time or to improve the profitability of the business in other ways, then computerisation will have been worth the effort. This isn't necessarily the case, and there is some evidence to suggest that introducing computers can be counter-productive in some situations. Because data can be stored and manipulated so easily, there is a tendency for people to store and manipulate more information than they need, thereby wasting time on unproductive work.

The question of compatibility is also important in two ways: the transfer of data from program to program, and from computer to computer. We have already noted how files produced by a spreadsheet such as Supercalc or Visicalc can be processed by DR Graph, and how some word processors and databases can be linked to create personalised letters. Anyone computerising an office must be sure that the programs can use each others' files if they need to – and as it may not be clear at first just what cross-use there could be, it is as well to buy those programs that give greatest compatibility with others. As for passing data between machines – this may not seem important where there is only a single computer in the office, but in the long term there will come a point where it is useful or necessary to buy additional or newer machines. No one can say what the future holds in this field, but those who use a CP/M system can have the confidence of knowing that there are very many other CP/M (and CP/M-compatible) machines out there. There will always be suitable computers that can be used to upgrade a system and continue to use all that data that has been so laboriously typed into the disks over the years.

Choosing and using CP/M business software is covered in a companion volume in this series, but let's have a look at some of the key types of programs and their uses.

Spreadsheets

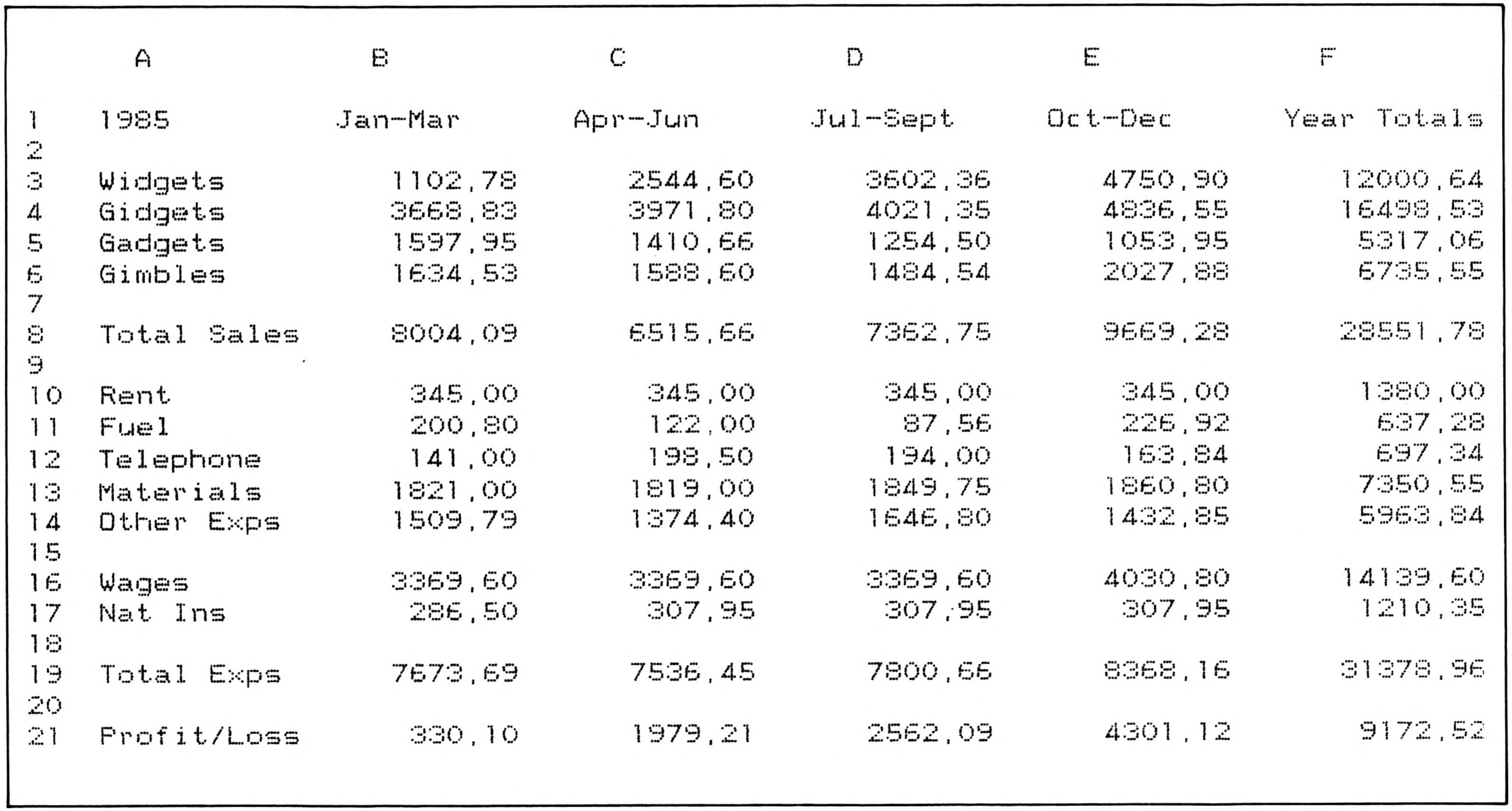
Spreadsheets derive their name from the large sheets of paper on which accountants used to assemble the financial records of a year's trading, with sales, costs, profits and losses organised by week or month, and totalled over time or by category. It is an essential part of drawing up the balance sheet for the end of the year accounts, and also gives a clear picture of the progress of the business and of the interrelations of different aspects of trading.



```
      A            B            C            D            E             F

1   1985        Jan-Mar      Apr-Jun      Jul-Sept     Oct-Dec      Year Totals
2
3   Widgets                                                         SUM(B3,E3)
4   Gidgets                                                         SUM(B4,E4)
5   Gadgets                                                         SUM(B5,E5)
6   Gimbles                                                         SUM(B6,E6)
7
8   Total Sales SUM(B3,B6)   SUM(C3,C6)   SUM(D3,D6)   SUM(E3,E6)   SUM(B8,E8)
9
10  Rent                                                            SUM(B10,E10)
11  Fuel                                                            SUM(B11,E11)
12  Telephone                                                       SUM(B12,E12)
13  Travel                                                          SUM(B13,E13)
14  Other Exps                                                      SUM(B14,E14)
15
16  Wages                                                           SUM(B16,E16)
17  Nat Ins                                                         SUM(B17,E17)
18
19  Total Exps  SUM(B10,B17) SUM(C10,C17) SUM(D10,D17) SUM(E10,E17) SUM(B19,E19)
20
21  Profit/Loss B8-B19       C8-C19       D8-D19       E8-E19        F8-F19
```

Figure 9.1 *Initial spreadsheet*



	A	B	C	D	E	F
1	1985	Jan-Mar	Apr-Jun	Jul-Sept	Oct-Dec	Year Totals
2						
3	Widgets	1102,78	2544,60	3602,36	4750,90	12000,64
4	Gidgets	3668,83	3971,80	4021,35	4836,55	16498,53
5	Gadgets	1597,95	1410,66	1254,50	1053,95	5317,06
6	Gimbles	1634,53	1588,60	1484,54	2027,88	6735,55
7						
8	Total Sales	8004,09	6515,66	7362,75	9669,28	28551,78
9						
10	Rent	345,00	345,00	345,00	345,00	1380,00
11	Fuel	200,80	122,00	87,56	226,92	637,28
12	Telephone	141,00	198,50	194,00	163,84	697,34
13	Materials	1821,00	1819,00	1849,75	1860,80	7350,55
14	Other Exps	1509,79	1374,40	1646,80	1432,85	5963,84
15						
16	Wages	3369,60	3369,60	3369,60	4030,80	14139,60
17	Nat Ins	286,50	307,95	307,95	307,95	1210,35
18						
19	Total Exps	7673,69	7536,45	7800,66	8368,16	31378,96
20						
21	Profit/Loss	330,10	1979,21	2562,09	4301,12	9172,52

Figure 9.2 *Spreadsheet showing quarterly accounts*

The computerised spreadsheet can be used in exactly the same way, but without the labour of the calculations. These can be written into the sheet and performed by the program. You can see this in the simplified examples shown in Figures 9.1 and 9.2. The first shows the spreadsheet as it was first set up, with the headings and the formulae for calculations written in. In the second figure the quarterly accounts have been keyed in, and totals produced by the program.

Each location, or cell, in the sheet is identified by a column letter and a row number. So, the sales of widgets between January and March are put in B3. The formulae are simply sums written with reference to selected cells on the spreadsheet. To calculate the profit or loss for the first quarter, you take the total expenses (shown in cell B19) form the sales for that time (in cell B8). The formula, B8 − B19, is put into cell B21, and when the spreadsheet performs its calculations the resulting figure will appear there. A good spreadsheet should not limit the nature of the formulae that can be used. If you can express it in a mathematical form, then you should be able to use it. All sheets also include special terms like SUM which will add up totals along a set part of a row or column, and the better ones even allow the use of logical expressions.

It took less time and typing than you might think to set up the spreadsheet. Only four formulae were typed in. The rest were replicated. For instance, the first of the right-hand totals was written in full 'SUM(B3,F3)', then the cursor was moved down to the lowest cell (F21) and the Replicate function called up. This copied the formulae to each of the cells in that column, adjusting the row numbers to the appropriate values. Any cell contents can be replicated, not only formulae but also text and given values so that standard headings and regular costs can be rapidly filled in. Any good spreadsheet will have this facility to copy individual cells, rows, columns or even whole blocks from one part of the sheet to another.

If you only use your spreadsheet to perform simple accounting work, of the sort illustrated, then you will get some benefit from it, but there is a lot more that this type of program can do. Our hypothetical firm, Mercurian Knick-Knacks Unlimited, could break down their total wage and material costs and relate them directly to the unit cost of producing widgets, gidgets and the rest. The total sales figures for these should also be produced by a 'number times unit price' formula, rather than simply writing them in. It is then possible to do 'What if' calculations on the sheets. What if they put more into widgets, and less into gimbles? Does this give more profit? What if they agree to the wage rises demanded by the labour force (young Jim and the Gaffer)? By how much will they have to increase prices to maintain profitability? What if they took out a loan to install a new gadget-press? How does the increase in output affect the rest of the position? You can do this kind of forward planning without a spreadsheet but not as easily or quickly. The sheer work involved in recalculating all of the figures means that you couldn't afford to do it very often. A spreadsheet will run through even the most complex analysis while you watch, allowing you to try out a whole range of possible futures in a short time.

Spreadsheets can also be used for generating invoices, handling stock control, and many other routine tasks that require analysis, calculation and data storage. A colleague of the author uses hers for assessing horses' diets, and it is an excellent tool for the job.

The main differences between the most-used programs are in size, flexibility and ease of use. Supercalc is well established and scores highly in its presentation – the manuals are excellent – but it is limited in memory. The 30K or so that is available sounds a lot to start with, but soon gets used up, especially if you use a lot of formulae. Scratchpad Plus, from Caxton, has almost unlimited memory as it uses the disk, either RAM or real, for storage; it is as easy to use and has the added features of windowing and special function keys. The Cracker is the most adaptable, and simple to get started on. In theory you can even use it to control your central heating system, though this is perhaps best left to the more advanced users!

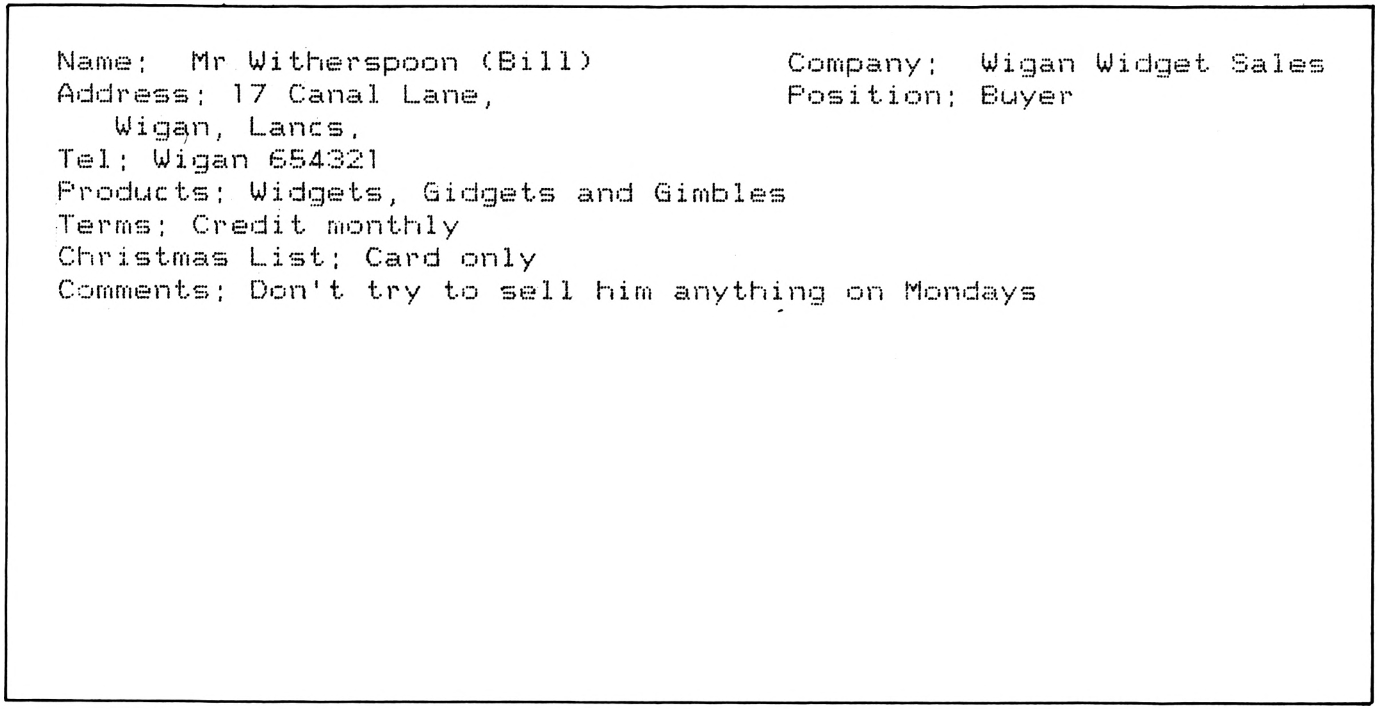
Databases

Database management programs have done for filing what spreadsheets have done for accounts. At the simplest level they are the electronic equivalent of card filing systems, but score over them in the speed and simplicity with which they can access individual 'cards' or sets. A good card index program will also allow you to link the data with a word processing program for 'Mail-merging' – creating personalised circulars.

The beauty of the cardbox-style database managers is that they relate so closely to existing card systems. This makes it very easy for the novice to understand and use, and keeps to a minimum the amount of time spent in setting up the program. The advantages of the system will soon become obvious. In a card file, the way that the cards are stored limits the ways in which you can access them. In a simple alphabetical name and address file, you have to search by name, but this may not be the focus of your search. If you want to find the people that live in a particular area, or those customers that only buy specific products, then there is no easy way to do it on this kind of manual system. Of courses, the card index can be structured by area or product interest, but makes it awkward in other ways.

Figure 9.3 shows the screen appearance of a typical record in the database at Mercurian Knick-Knacks. As you can see, its layout is much the same as it would be on a card, and in fact it is exactly the same as their old card files. It was deliberately designed this way to make it easier for the secretary to transfer across all the data, and for the people who would use the database to find things on the screen. In the old manual system, you could find a named person by rifling through the drawer, and it only took a few seconds. When the secretary drew up the Christmas card list, or the salesman worked out an itinerary, they were faced with the chore of reading through all of the cards. In the computerised system, they can set up a search for those cards containing the word 'Card' or 'Lancs' and all the relevant ones will be grouped together in less than a minute.

The other obvious advantages of electronic over manual card files is that they can be updated easily and neatly; the 'cards' don't get dog-eared with use; and they take far less



```
Name:  Mr Witherspoon (Bill)            Company:  Wigan Widget Sales
Address: 17 Canal Lane,                 Position: Buyer
   Wigan, Lancs,
Tel: Wigan 654321
Products: Widgets, Gidgets and Gimbles
Terms: Credit monthly
Christmas List: Card only
Comments: Don't try to sell him anything on Mondays
```

Figure 9.3 *Screen appearance of a typical record*

space to store. The record on Figure 9.3 has approximately 200 characters on it. Over 800 such records could be stored on a single side of a standard 3-inch disk!

The Quest Mailbox is a good cheap example of this type, and Caxton's Cardbox, though more expensive, has many attractive features – not the least being size. It can store up to 65,500 different records, each of which may contain over a thousand characters. Of course, if you actually wanted a maximum size file, you would need a hard disk drive to store it, and a team of typists to key it all in and keep it up to date. (There's about 2000 skilled typing hours in creating a 65 megabyte database.)

The more sophisticated database managers give you far greater flexibility in the way that you can handle data. The best current example is probably Condor (Caxton Software), which allows you to define the structure of the database and to design your own screen and paper layouts. It is easy to use – the commands are given in English not jargon – but most importantly it has very extensive file-handling facilities, including the ability to filter and merge files together to create a new database.

Spreadsheets and databases are the most striking examples of programs that don't merely replace paper-based office work, but also create new opportunities for increasing efficiency and profitability. There are many other types of commercial software which can do much to cut out routine clerical and accounting tasks and free staff to do more useful work. Payroll, stock control, invoicing and ledger programs suitable for small to medium-size business are readily available. Sage, Camsoft and Caxton are currently the leading names in the business software field and you should look closely at their products before committing yourself to any purchase.

As long as you start out with a very clear idea of what you want to do, its not a difficult task, given the range and nature of today's software, to find a suitable program and to configure it to your needs.

APPENDIX A

CONVERTING ASSEMBLER MNEMONICS

Z80	8080
ADC A,n	ACI n
ADC A,r	ADC r
ADD A,n	ADI n
ADD A,r	ADD r
ADD HL,rr	DAD rr
AND n	ANI n
AND r	ANA r
CALL addr	CALL addr
CALL C,addr	CC addr
CALL M,addr	CM addr
CALL NC,addr	CNC addr
CALL P,addr	CP addr
CALL PE,addr	CPE addr
CALL PO,addr	CPO addr
CALL Z,addr	CZ addr
CCF	CMC
CP r	CMP r
CP n	CPI n
CPL	CMA
DAA	DAA
DEC r	DCR r

Z80	8080
DEC rr	DCX rr
DI	DI
EI	EI
EX DE,HL	XCHG
EX (SP),HL	XTHL
IN A,(n)	IN n
INC r	INR r
INC rr	INX rr
JP addr	JMP addr
JP (HL)	PCHL
JP C,addr	JC addr
JP M,addr	JM addr
JP NC,addr	JNC addr
JP P,addr	JP addr
JP PE,addr	JPE add
JP PO,addr	JPO addr
JP Z,addr	JZ addr
LD r,n	MVI r,n
LD r,r	MOV r,r
LXI rr,nn	LD rr,nn
LD (addr),A	STA addr
LD (addr),HL	SHLD addr
LD (BC),A	STAX B
LD (DE),HL	STAX D
LD A,(addr)	LDA addr
LD A,(BC)	LBAX
LD A,(DE)	LDAX
LD r,(HL)	MOV r,M
LD HL,(addr)	LHLD
LD SP,HL	SPHL
NOP	NOP
OR n	ORI n
OR r	ORA r
OUT (n),a	OUT n
PUSH AF	PUSH PSW
PUSH rr	PUSH rr
POP AF	POP PSW
POP rr	POP rr
RET	RET
RET C	RC
RET M	RM
RET NC	RNC
RET P	RP
RET PE	RPE
RET PO	RPO

Z80	**8080**
RET Z	RZ
RLA	RAL
RLCA	RLC
RRA	RAR
RRCA	RRC
RST n	RST n
SBC A,n	SBI n
SBC A,r	SBB r
SCF	STC
SUB n	SUI n
SUB r	SUB r
XOR n	XRI n
XOR r	XRA r

n – number given as immediate data
nn – two-byte number
addr – address
r – single register (A,B,C,D,E,H,L) or the contents of the byte addresses by HL. In 8080, M is used instead of (HL).
rr – register pair. These are identified by BC, DE, HL, SP in Z80 mnemonics, but by B, D, H and SP in 8080 mnemonics.

Absent friends
The Z80 CPU is more complex than the 8080, and this is reflected in the limitations of 8080 assembler language. There are no 8080 mnemonics that can handle the following:

Any use of the index registers IX and IY;
Block compare, input or load
BIT, RES and SET •
Relative jumps, including DJNZ
Access to the alternate register set or interrupts
Rotate and shift in registers other than A
Two-byte arithmetic other than DAD rr

Code generated via Z80 mnemonics can, of course, run under CP/M, but it cannot be assembled, disassembled or debugged by the packaged development utilities.

APPENDIX B

ASCII NUMBER EFFECTS

In CP/M 2.2 on either the 464, 664 or 6128 the normal control codes remain effective and may be used to move the cursor, set windows, change modes and ink and border colours. Where only a single character is involved, send it via Function 2 – Console Output. If the control code needs parameters, for example the ink number and colours, then the code and parameters should be sent to the console in an unbroken stream with Function 9 – Print String.

ASCII No.	*Parameters*	*Effect*
0	-	None
1	char code	Print character at current cursor position
2	-	Do not display text cursor
3	-	Display text cursor
4	mode no.	Set new screen mode
5	char code	Print character at current graphics position
6	-	Enable text screen
7	-	Bleeper
8	-	Cursor left
9	-	Cursor right
10	-	Cursor down
11	-	Cursor up
12	-	Clear screen and move cursor to top left
13	-	Cursor to left side of window
14	Ink no.	Set Paper to given Ink number
15	Ink no.	Set current Pen to Ink number
16	-	Delete last character
17	-	Clear from left edge to current cursor position
18	—	Clear from current cursor position to right edge
19	-	Clear from top left of window to current position
20	-	Clear from current position to bottom right of window
21	-	Turn off text screen. (Restore with code 6)
22	0/1	Use 1 for 'transparent' printing, 0 for normal
23	Print mode	0 – normal graphic inks, 1 – XOR, 2 – AND, 3 – OR
24	-	Inverse print – swap paper and pen inks
25	UDC data	Acts as SYMBOL command. Needs character code and 8 bytes to define the image.
26	corners	Acts as WINDOW command. Needs left/right column numbers and top/bottom row numbers.
27	-	Escape code – ignored
28	Ink data	Acts as INK command. Needs ink number plus two colour codes
29	colours	Changes Border colour. Needs two colour codes
30	-	Home cursor (to top left)
31	row,col	Acts as LOCATE command. Needs Row and Column numbers

APPENDIX C

ASCII AND HEXADECIMAL CONVERSIONS

ASCII stands for American Standard Code for Information Interchange. The code contains 96 printing and 32 non-printing characters used to store data on a disk. Table C1 defines ASCII symbols and Table C2 lists the ASCII and hexadecimal conversions. The table includes binary, decimal, hexadecimal and ASCII conversions.

Table C1 **ASCII symbols**

Symbol	*Meaning*	*Symbol*	*Meaning*
ACK	acknowledge	FS	file separator
BEL	bell	GS	group separator
BS	backspace	HT	horizontal tabulation
CAN	cancel	LF	line-feed
CR	carriage return	NAK	negative acknowledge
DC	device control	NUL	null
DEL	delete	RS	record separator
DLE	data link escape	SI	shift in
EM	end of medium	SO	shift out
ENQ	enquiry	SOH	start of heading
EOT	end of transmission	SP	space
ESC	escape	STX	start of text
ETB	end of transmission	SUB	substitute
ETX	end of text	SYN	synchronous idle
FF	form-feed	US	unit separator
		VT	vertical tabulation

Table C2 ASCII conversion table

Binary	*Decimal*	*Hexadecimal*	*ASCII*	
0000000	0	0	NUL	
0000001	1	1	SOH	(CTRL-A)
0000010	2	2	STX	(CTRL-B)
0000011	3	3	ETX	(CTRL-C)
0000100	4	4	EOT	(CTRL-D)
0000101	5	5	ENQ	(CTRL-E)
0000110	6	6	ACK	(CTRL-F)
0000111	7	7	BEL	(CTRL-G)
0001000	8	8	BS	(CTRL-H)
0001001	9	9	HT	(CTRL-I)
0001010	10	A	LF	(CTRL-J)
0001011	11	B	VT	(CTRL-K)
0001100	12	C	FF	(CTRL-L)
0001101	13	D	CR	(CTRL-M)
0001110	14	E	SO	(CTRL-N)
0001111	15	F	SI	(CTRL-O)
0010000	16	10	DLE	(CTRL-P)
0010001	17	11	DC1	(CTRL-Q)
0010010	18	12	DC2	(CTRL-R)
0010011	19	13	DC3	(CTRL-S)
0010100	20	14	DC4	(CTRL-T)
0010101	21	15	NAK	(CTRL-U)
0010110	22	16	SYN	(CTRL-V)
0010111	23	17	ETB	(CTRL-W)
0011000	24	18	CAN	(CTRL-X)
0011001	25	19	EM	(CTRL-Y)
0011010	26	1A	SUB	(CTRL-Z)
0011011	27	1B	ESC	
0011100	28	1C	FS	
0011101	29	1D	GS	
0011110	30	1E	RS	
0011111	31	1F	US	
0100000	32	20	(SPACE)	
0100001	33	21	!	
0100010	34	22	"	
0100011	35	23	#	
0100100	36	24	$	
0100101	37	25	%	
0100110	38	26	&	
0100111	39	27	'	
0101000	40	28	(	
0101001	41	29	)	
0101010	42	2A	*	

Binary	*Decimal*	*Hexadecimal*	*ASCII*
0101011	43	2B	+
0101100	44	2C	,
0101101	45	2D	-
0101110	46	2E	.
0101111	47	2F	/
0110000	48	30	0
0110001	49	31	1
0110010	50	32	2
0110011	51	33	3
0110100	52	34	4
0110101	53	35	5
0110110	54	36	6
0110111	55	37	7
0111000	56	38	8
0111001	57	39	9
0111010	58	3A	:
0111011	59	3B	;
0111100	60	3C	<
0111101	61	3D	=
0111110	62	3E	>
0111111	63	3F	?
1000000	64	40	@
1000001	65	41	A
1000010	66	42	B
1000011	67	43	C
1000100	68	44	D
1000101	69	45	E
1000110	70	46	F
1000111	71	47	G
1001000	72	48	H
1001001	73	49	I
1001010	74	4A	J
1001011	75	4B	K
1001100	76	4C	L
1001101	77	4D	M
1001110	78	4E	N
1001111	79	4F	O
1010000	80	50	P
1010001	81	51	Q
1010010	82	52	R
1010011	83	53	S
1010100	84	54	T
1010101	85	55	U
1010110	86	56	V

Binary	*Decimal*	*Hexadecimal*	*ASCII*
1010111	87	57	W
1011000	88	58	X
1011001	89	59	Y
1011010	90	5A	Z
1011011	91	5B	[
1011100	92	5C	\
1011101	93	5D	]
1011110	94	5E	^
1011111	95	5F	_
1100000	96	60	'
1100001	97	61	a
1100010	98	62	b
1100011	99	63	c
1100100	100	64	d
1100101	101	65	e
1100110	102	66	f
1100111	103	67	g
1101000	104	68	h
1101001	105	69	i
1101010	106	6A	j
1101011	107	6B	k
1101100	108	6C	l
1101101	109	6D	m
1101110	110	6E	n
1101111	111	6F	o
1110000	112	70	p
1110001	113	71	q
1110010	114	72	r
1110011	115	73	s
1110100	116	74	t
1110101	117	75	u
1110110	118	76	v
1110111	119	77	w
1111000	120	78	x
1111001	121	79	y
1111010	122	7A	z
1111011	123	7B	{
1111100	124	7C	|
1111101	125	7D	}
1111110	126	7E	~
1111111	127	7F	DEL

APPENDIX D

THE BDOS FUNCTIONS

All BDOS functions are accessed in the same way, through a call to 0005.

Before the call is made, the C register must be loaded with the function number. If data has to be passed to the function, it will be through either the E register, or the DE pair. E is used for a single byte – normally a character code. Sometimes the value in E selects an option within the function. The DE pair is used for passing the address, which may be of a block of data for printing, a File Control Block, a buffer in which input will be stored or of the location to which a block of memory is to be moved.

Data may be returned by the function call through either A or HL. If the HL pair is used, then A will contain the same value as L, and B the same as H.

All the functions will corrupt the registers and flags to a greater or lesser extent, though the Stack Pointer and Program Counter will be safeguarded. It is up to the programmer to protect any other registers that are needed later.

BDOS input/output functions

The way that the control characters (CTRL S, CTRL P, CTRL Q and CTRL C) are treated depends upon the Console Mode, which can be reset by function 110. In the following function descriptions, the default Mode is assumed.

Function 0 – System Reset

This is the normal means of ending a transient program. It effects a warm start, restoring the CCP operation. In 2.2, but not in Plus, it also resets the disk system, logging in drive A.

Function 1 – Console Input

On Exit: A contains ASCII character code

This waits for a character to be input from the logical device CONIN: – which is usually the keyboard. Any printing character and carriage return, line feed, and delete will be echoed on the screen, and some other non-printing characters are also detected and acted upon:

CTRL S will stop the screen scrolling. In 2.2 any keypress will restart the scroll, but only CTRL Q can be used for this in Plus.

CTRL P is the on/off toggle for printer output.

CTRL I and TAB move the cursor to the next 8-column TAB place.

Function 2 – Console Output

On Entry: E contains ASCII character code.

This outputs a character to the logical device CONOUT: – normally the screen – and also to LST: if the printer has been activated.

In 2.2 you can use function 2 to send the control characters (0 to 31) to move the cursor, set windows, inks, etc. as usual. In CP/M Plus only TAB characters are acted upon. The rest are ignored.

The function checks for stop and start scroll commands.

Function 3 – Auxiliary Input

On Exit: A contains ASCII character code.

This is equivalent to Function 1, waiting for a character from the device assigned to AUXIN: or RDR: in 2.2. The character is not echoed on the screen, and there is no check for the printer toggle or scroll controls.

Function 4 – Auxiliary Output

On Entry: E contains ASCII character code.

The character in E is sent to the device assigned to AUXOUT: or PUN:.

Function 5 – List Output

On Entry: E contains ASCII character code.

This is identical to Function 4, except that output is to the device – normally a printer – assigned to LST:.

Function 6 – Direct Console Input/output

On Entry: E contains:	0FFhex – for input and status check, or
	0FEhex – for status check only, or
	0FDhex – for input, or
	ASCII character code

On Exit: A contains ASCII character code or status indicator.

Function 6 allows flexible and unrestricted input/output with the console. If E contains any value below 0FDhex, it will be treated as an ASCII code and sent to the screen – perhaps with unpredictable results.

Higher values give alternative ways of reading the keyboard. You can check its status – whether or not a key has been pressed – by loading E with 0FEhex. If there has been a key press, A contains 0FFhex on exit, otherwise it contains 00.

Load E with 0FDhex and the function will wait until a key is pressed, then return its character code in A. Characters input in this way are not echoed to the screen, and this may be useful in some situations.

The third option combines these two. Load E with 0FFhex and the function will read the keyboard – without waiting – and load A with either the ASCII code of a pressed key or 00 if no key is down.

Function 7 – Auxiliary Input Status

On Exit: A contains status.

The value in A will be 0FFhex if there is a character ready for input from AUXIN:, otherwise 00.

Function 8 – Auxiliary Output Status

On Exit: A contains status.

Use this to check if the auxiliary device is ready to receive a character. If it is then A will contain 0FFhex, if not it will contain 00.

Function 9 – Print String

On Entry: DE contains address of string.

This takes the string of characters stored at the given address and transfers them to the screen, and to the printer if selected by CTRL P. The end of the string is usually marked by a '$', though this delimiter can be changed by Function 110 in CP/M Plus. The function checks for stop and start scroll commands.

Function 10 – Read Console Buffer

On Entry: DE contains buffer address.

Characters are read in from the keyboard (CONIN:) and stored in the buffer addressed by DE. Input ends when a carriage return (ENTER or CTRL M) or line feed (CTRL J) is entered. If an attempt is made to enter more characters than the buffer can hold, then the extra ones are discarded and the bell is sounded.

The normal CCP editing commands are active within the function, and the buffer can be initialised with a string of characters for the user to edit. This string must be terminated with a null character – ASCII 00 – rather than the usual string delimiter '$'.

The buffer address and size is under the control of the programmer. It will always take the format:

```
ADDRESS   DE    DE+1  DE+2 DE+3 DE+4...  DE+max+2
CONTENTS max : num :  ch1 :  ch2 :  ch3 : ....  chmax:
```

The required buffer size is stored in the first location before the function is called. On exit, address+1 will contain the number of input characters.

If the function is to be used to print a string on screen for editing, then DE should be set to 00, and the function will treat th current Direct Memory Address as the buffer address. The DMA can be set by func tion 26.

The function does not respond to stop/start scroll commands, but it does act upon the printer toggle, and if CTRL C is entered as the first character in the line then the program will be aborted and control returned to the CCP.

Function 11 – Get Console Status

On Exit: A contains status.

The keyboard is checked to see if a character has been entered. On exit A will contain 01 if one has, or otherwise 00. The function can be altered so that it checks for CTRL C only, by resetting the Console Mode using Function 110.

Function 12 – Return Version Number

On Exit: HL contains version number.

H will contain 00 for either version of CP/M, and L will have either 22hex or 31hex if it is a Plus system. Where a program is intended for use on several machines, the variations in the operating systems will mean either that you need different subroutines for the different versions, or that some aspects of the program may need to be switched off. If the program is version-specific (it might make heavy use GSX graphics) then a call to function 12 at the start should be used to check that the program can run.

Function 13 – Reset Disk System

This resets all drives to Read/Write, selects drive A as the default and restores the DMA to 80hex. It is used where a program requires a disk to be changed, as for example in FILECOPY and the DISCKIT copy routine. Function 37 offers an alternative way of resetting.

In CP/M 2.2, this function also re-initialised the BIOS.

Function 14 – Select Disk

On Entry: E contains drive number.

On Exit: A and H contain error codes.

This selects the default disk drive and logs it in. The 'drive number' is 0 for A, 1 for B, and so on up to 15 for P. Notice that this is different to the drive numbers used in File Control Blocks, where 0 identifies the default drive, 1 refers to A, 2 to B, etc.

If the drive is activated successfully, A is returned containing 00, otherwise it will be loaded with FFhex. The treatment of errors is otherwise as covered immediately below.

BDOS file-handling functions

The next set are all concerned with aspects of file handling, and have a number of features in common. They all require a File Control Block (see Chapter 5) to be initialised in memory before the call is made. The FCB's address must normally be given in the DE register pair.

After a successful call, A will either contain 00, or a directory code in the range 0 to 3. This indicates the position of the file's block within the DMA area (see Chapter 5).

Errors are treated in one of two ways, depending on the BDOS error mode. In default mode, an error message will be displayed on the screen, and the calling program will be aborted. If the error mode has been changed by Function 45, then the function will return to the calling program with error codes in A and H. The A code will normally be FFhex, other codes will be given in the appropriate function description. H codes are standard.

Physical Error Codes – H Register

01 – disk I/O error
02 – read-only disk
03 – read-only file
04 – invalid drive number
07 – invalid password
08 – file already exists
09 – wildcard in file specification not acceptable

Function 15 – Open File

On Exit: A = 00 or FFhex; H = 00, 01, 04, 07 or 09.

The file to be opened must already exist. New files are opened by Function 22 – Make File. The user has full access to his own files, and may also open system files in the area of User 0, though these may be only opened for reading.

Password-protected files can only be opened if the correct password has been written at the start of the current DMA, or if a default password has been established.

Access data and time stamping will take place at this stage if the directory label requires it, and if byte 12 of the FCB (the extent field) is set to 0.

Function 16 – Close File

On Exit: A = 00 or FFhex; H = 00, 01, 02 or 04.

This closes a file which has been opened with either Open File or Make File. The directory will be updated if any write operations have taken place.

The close can be permanent or partial, where the directory is updated but the file remains open. The choice is made via the attribute bit on byte 5 of the filename. Set bit 7 if a partial close is wanted; leave it at 0 for a permanent close.

Function 17 – Search for First

On Exit: A = directory code or FFhex for error; H = 00, 01 or 04.

This searches the directory for a match with the data in the FCB. This may be an unambiguous filename, or wildcards may be used. If the first byte of the block is 63 (?), then the every file in the directory will be a match. During the search operation, directory data is copied into the DMA one record – four files – at a time. This is not normally relevant to the programmer, but may be accessed if required.

The function is ended when the first match is found, or if the search fails. A will contain FFhex to indicate failure, or a value from 0 to 3. Multiply this by 32 and add to the DMA address to get the location of the directory copy of the FCB. (See the example program in Chapter 5.) After a successful search, further matches may be sought by Search for Next, as long as no other file functions are called in between.

Function 18 – Search for Next

This function is virtually identical to Search for First, which it must follow. The FCB address is not needed on entry.

Function 19 – Delete File

On Exit: A = directory code or FFhex for error; H = 00, 01, 02, 03, 04 or 07

The file, or files (wildcards may be used) to be deleted must be in the user's area, and not write-protected. If the file has a password, this must be given in the DMA or have been established as the default as with Open File operations.

A file is deleted from the directory by setting F5hex as the user number. Disk space allocated to the file becomes free space again.

Function 20 – Read Sequential

On Exit: A = error code: 01 – if end of file reached, 09 – error in FCB data, 10hex – disk has been changed, FFhex – physical error; H = 00, 01 or 04.

This reads into DMA memory the 128-byte record identified by the current record and extent bytes of the FCB. The function then increments the current record number, and the extent count if the current record overflows. Both bytes should be set to zero in the Open File call if the read operations are to start from the first record.

Function 21 – Write Sequential

On Exit: A = error code: 01 – no free directory space for update, 02 – no free disk space for record, 09, 10hex or FFhex as above; H = 00, 01, 02, 03 or 04.

This writes the 128-byte record at the current DMA address onto disk at the position identified by the current record and extent numbers in the FCB. These values are then updated to point to the next record.

The first Write operation in a session will also call up the date stamping routine if the directory label requests Update stamps.

Function 22 – Make File

On Exit: A = 00 or FFhex; H = 00, 01, 02, 04, 08 or 09.

This creates a new disk directory entry from the data given in the FCB. If a password is to be assigned to the file, it must be present at the start of the DMA, and the attribute bit (bit 7) of byte 6 in the FCB must be set to 1. A Create and/or Update time stamp will be set if required by the directory label.

Function 23 – Rename File

On Exit: A = 00 or FFhex; H = 00, 01, 02, 03, 04, 07, 08 or 09.

The new filename is written into the FCB between bytes 17 and 27. Wildcards may not be used and the name must not already exist in the directory. If the file is password-protected, the password must be given in the DMA or have been established as the default.

Other function calls

The remaining function calls are more general drive and memory functions, but also include a number that handle files, and their entry and exit conditions are as above.

Function 24 – Return Login Vector

On Exit: HL contains Login Vector.

This 16-bit value is bit-significant, with one bit for each drive and '1' indicating that the drive is active. The least significant bit (bit 0 of L) refers to drive A, and bit 7 of H refers to drive P.

Function 25 – Return Current Drive

On Exit: A contains number of default drive.

Here, 0 corresponds to drive A , 1 to B and so on up to 0Fhex for drive P.

Function 26 – Set DMA Address

On Entry: DE contains new DMA address.

The Direct Memory Address locates the the area in memory normally used as a buffer for storing data during read and write operations to the disk. The DMA is also for passing data to or from functions – passwords are written here, for example.

The default DMA address of 80hex is restored when the disk system is reset.

Function 27 – Get Address of Allocation Vector

On Exit: HL contains address or FFFFhex to show error.

The allocation vector in memory can be used to calculate the amount of free space on a drive. In CP/M Plus, the vector may well be in banked memory and therefore inaccessible. If the drive is Read-Only, the allocation data may not be accurate. In either case, Function 46 will provide a simpler way to determine free space.

Function 28 – Write Protect Disk

This sets temporary write-protection for a drive. It can be removed by resetting the disk system.

Function 29 – Get Read-Only Vector

On Exit: HL contains Read-only vector.

The 16-bit value has the same structure as the Login Vector. A bit set to '1' indicates Read-only status for a drive.

Function 30 – Set File Attributes

On Entry: DE contains FCB address

On Exit: A = 00 or FFhex; H = 00, 01, 02, 04, 07 or 09

The resulting status of the file attributes will depend upon the setting of the high-order bits of bytes 1 to 4 of the filename, and the three bytes of the filetype. Type 1 selects Read-only, type 2 selects System and type 3 selects Archive. The filename attribute bits have no special meaning to CP/M, but can be used by the programmer.

Function 31 – Get Address Disk Parameter Block

On Exit: HL contains address of DPB or FFFFhex to show error.

The DPB, containing the parameters of the current drive, is located within the BIOS area and may be read by the programmer. The information contained there is an unprocessed form of that given by STAT DSK or SHOW [DRIVE]. If you create and load this short assembler routine, you can then trace it with DDT or SID and pick up the DPB location from the HL pair after the call:

```
MVI C,31
CALL 5
RET
```

The block's structure and values under 2.2 and Plus are shown here.

Byte	*Purpose*	*CP/M 2.2*	*CP/M Plus*
0-1	Sectors per track	26	26
2	Block Shift Factor	3	3
3	Block Mask	7	7
4	Extent Folding Mask	0	0
5-6	Maximum number of blocks – 1	170	179
7-8	Max. directory entries – 1	63	63
9-10	Directory allocation map	192	192
12-13	Directory Check size	16	16
13-14	Reserved Track Offset	2	0

The block size can be calculated either by multiplying the shift factor (+1) by 256, or by multiplying the block mask (+1) by 128. Either way, you get a block size of 1K.

Function 32 – Get/Set User Code

On Entry: A contains User code or FFhex to Get.

On Exit: A contains User code after Get.

Load A with FFhex to use this function to get the current user number, or select a new user by loading the value into A. The code range is 0 to 15.

Function 33 – Read Random

On Entry: DE contains FCB address.

On Exit: A contains error code = 01 – end of file, 03 – current extent cannot be closed, 04 – extent not created for file, 06 – record number too large, 10hex – disk has been changed, FFhex – physical error; H = 01 – Disk I/O, 04 – invalid drive.

The record addressed by the current record number is read into DMA memory. The current record number is in the range 0 to FFFF (65535), and is stored in bytes 33 and 34 of the FCB, low byte first. (At 128 bytes a record, this would access an 8 megabyte file! In practice the range will be much lower.) Larger CP/M systems can access even larger files, and there the record number also occupies byte 35. This must be set to 0, to maintain compatibility.

To read a random record, the file must have been opened with the current extent set to zero.

The function does not change the current record number. That is the programmer's responsibility.

Function 34 – Write Random

On Entry: DE contains FCB address.

On Exit: A contains error code = 02 – no data space, 05 – no directory space, 03, 06, 10hex, FFhex as above; H = 01 – Disk I/O, 02 – R/O Disk, 03 R/O Drive, 04 – invalid drive.

The record is accessed via the current record number as in Read operations, and the data to be written must first be stored at the current DMA address.

The file will be date stamped on the first Write of a session if Update stamps are requested by the directory label.

Function 35 – Compute File Size

On Entry: DE contains FCB address.

On Exit: A = 00 or FFhex; H = 00, 01 – Disk error, 04 – invalid drive.

The FCB must have random file format, i.e. bytes 33 to 35 must be present. The function finds the last record in the file and sets the current record number in the FCB to one more than this. Multiply the 16-bit value in 33 and 34 by 128 to get the file size in bytes.

Function 36 – Set Random Record

On Entry: DE contains FCB address.

On Exit: Bytes 33-35 of FCB contain current record number.

This may be used after sequential read or write operations to find the current record number, and is therefore of value in cataloguing files with variable length records, and in switching from sequential to random file access.

CP/M Plus function calls

Function 37 – Reset Drive

On Entry: DE = drive vector

This function allows individual drives to be reset. The 16-bit drive vector is bit significant, with bit 0 of E controlling drive A, and the high bit of D controlling drive P. Any bit set to 1 will reset the associated drive.

Function 40 – Write Random with Zero Fill

On Entry: DE = FCB address.

On Exit: A and H = error codes.

When a random file is created, disk space allocated to that file may contain data from previous use. This function will erase residual garbage, and could also be used to wipe records. It is identical to Write Random in all other respects.

Function 45 – Set BDOS Error Mode

On Entry: E contains error mode: FFhex – Return Error
FEhex – Display and Return
any other - Display and Abort (default).

In the default mode, any BDOS errors will result in the display of an error message and the calling program will be aborted. In Return Error, error codes are passed back to the calling program through the A and H registers. In Display and Return, the BDOS will display the error message, but then continue with the execution of the calling program.

Function 46 – Get Free Space

On Entry: E = drive number.

On Exit: A = 00 or FFhex; H = 00, 01 or 04.

The drive numbers here go from 0 for drive A through to 15 for P. On return, the first three bytes of the DMA should contain the number of free bytes on the disk. The three byte figure is given, as usual, low byte first. If the drive is set to Read/Only the free space may not be calculated correctly.

Function 47 – Chain to Program

On Entry: DMA = command line.

This allows one program to call up the next directly. The command line in the DMA is what would otherwise by typed at the A> prompt, and is terminated by a null character (00hex). If E is set to FFhex, then the current drive and user number are retained in the new program, but if any other value is present the normal default values are used. Function 108 – Get/Set Program Return Code – may be used with this function to pass a two-byte value to the next program.

Function 48 – Flush Buffers

On Entry: E = FFhex to flush all buffers.

On Exit: A = 00 or FFhex; H = 00, 01, 02, 04.

The function copies to disk any write-pending records. If E contains FFhex, all buffers are cleared. This is necessary if write operations are followed by re-reading to verify data. If the buffers were not flushed, there would be no way of ensuring that had been written and read back properly.

Function 49 – Get/Set System Control Block

This allows direct access to the 100-byte System Control Block in BDOS memory where essential system data is stored, and should not normally be used. All parameters can be altered more safely through other specific functions.

Function 50 – Direct BIOS Calls

This allows a BIOS function to be called directly, rather than via a BDOS function. As all BIOS functions, except some specialised disk access, can be performed more safely and conveniently through the BDOS, function 50 should rarely be used.

Function 59 – Load Overlay

On Entry: DE = FCB address.

On Exit: A = 00, 01 – end of file, 09 – FCB error, 10hex – disk changed, FEhex – bad address or not enough memory, FFhex – LOADER missing; H = 00, 01, 04.

This can only be used with programs that have an RSX header, as the LOADER RSX is needed to load in the overlay. The overlay modules may be relocatable – identified by the .PRL type – or load to an absolute address. The load address is given in bytes 33-34 of the FCB, and must be a page boundary if the overlay is relocatable.

Function 60 – Call RSX

On Entry: DE = RSX Parameter Block address.

On Exit: A = FFhex if RSX not found.

This calls an RSX and passes parameters to it through the RSX PB. The block may be located anywhere in main memory, and has this format:

```
PARMS: DB function_number
DB parameter_count
DW parameter_word_1 ;every parameter occupies two bytes
DW parameter_word2
....
DW parameter_word_last
```

Function 98 – Free Blocks

On Exit: A = 00 or FFhex – error; H = 00 or 04.

This returns to free space any disk blocks that have been written but not yet recorded in the directory by a Close File operation. As this function is called at a warm start, it is essential to close files properly to preserve data.

Function 99 – Truncate File

On Entry: DE = FCB address.

On Exit: A = 00 or FFhex if error; H = 01, 02, 03, 04, 07, 09.

Bytes 33–35 of the FCB must contain the number of what will become the last record in the truncated file. In a sparse random file i.e. one that has some unwritten records, the specified last record must be a written one.

The password must be written in the DMA, or set by default, if the file is protected.

Function 100 – Set Directory Label

On Entry: DE = FCB address.

On Exit: A = 00 or FFhex if error; H = 01, 02, 04 or 07.

This creates or updates the directory label which determines the nature of date/time stamps and password protection. Byte 12 of the FCB gives the data in bit-significant form:

Bit 7 – Password protection
6 – Access stamps
5 – Update stamps
4 – Create stamps
0 – New directory password

If Bit 0 is set, then the function will look in the second eight bytes of the DMA for the new password to be assigned to the directory label. If time stamps are to be activated, the directory must have been processed with the INITDIR utility to create the SFCB fields in which stamps are maintained.

Function 101 – Get Directory Label Data

On Entry: E = drive code.

On Exit: A = Label byte, 0 – no label,or FFhex if error; H = 01 or 04.

The drive codes are 0 for A, 1 for B, up to 15 for drive P. The bit significance of the label byte is as in Function 100.

Function 102 – Read File Date Stamps and Password Mode

On Entry: DE = FCB address.

On Exit: A = 00 or FFhex if error; H = 01, 04 or 09.

After the function has been executed the FCB will contain password and stamp data. The password mode is coded into byte 12:

Bit 7 – Read protect
6 – Write protect
4 – Delete protect

The Create or Access time stamp is in bytes 24–27, and the Update stamp in bytes 28–31. Both have the same structure. The first two bytes hold the number of days since 1 January 1978. The next two hold the hour and minute in binary coded decimal. (To read BCD numbers, split each byte into two and convert each half separately – 10010110 gives 0011 0110 or 36.)

Function 103 – Write File XFCB

On Entry: DE = FCB address.

On Exit: A = 00 or FFhex; H = 01, 02, 04, 07, 09.

This creates or updates the eXtended FCB for a file, giving or altering a password and/or the protection mode. The mode is set in byte 12 of the FCB, with bits 7,6 or 4 set to indicate the mode, as in Function 102. If bit 0 is set, then a new password is assigned. This must be present in the second eight bytes of the DMA. XFCBs can only be created for files if the directory label supports passwords, and if there is space in the directory in which to write the XFCB.

Function 104 – Set Date and Time

On Entry: DE contains DATE block address.

The DATE block consists of four bytes arranged as in Function 102. Seconds are not given, and this field will be set to zero by the function.

Function 105 – Get Date and Time

On Entry: DE = DATE block address.

On Exit: A contains seconds; DATE block contains date, hours and minutes

The DATE block structure is as above. The seconds are given in BCD digits in A. If the time has not been set during the session, then the time will be calculated from system start-up.

Function 106 – Set Default Password

On Entry: DE = password address.

When a function tries to access any password protected file, it looks for that password in the DMA and also compares it with the default string. This function assigns the eight byte string addressed by DE to the default.

Function 107 – Get Serial Number

On Entry: DE = buffer address.

A six-byte buffer is needed to store the CP/M Plus serial number. It is held in simple ASCII character digits.

Function 108 – Get/Set Program Return Code

On Entry: DE = code to Set, or FFFFhex to Get code.

On Exit: HL = code after Get.

Where programs are chained together, it may be important that one has been successfully completed before the next starts. If the command line in the DMA starts with a colon (:), then the program will only be run if the return code shows that the previous one ended without error. Any code beginning FF.. indicates an error. FFFD is a BDOS error. FFFE shows that the user exited the previous program with CTRL and C. Other error codes, or any values below FF00 may also be passed to the next program through the return code function.

Function 109 – Get/Set Console Mode

On Entry: DE = new mode or FFFFhex to Get.

On Exit: HL = current mode after Get.

The Console Mode data is bit-significant.

```
Low Byte: Bit 0 – if set read only CTRL C status in Function 11
              1 – if set disable stop/start scroll controls
              2 – if set disable tab expansion and printer echo
              3 – if set disable CTRL C abort
```

Function 110 – Get/Set Output Delimiter

On Entry: DE = FFFFhex to Get, or new character code in E.

On Exit: A = delimiter character after Get.

The default delimiter – used to mark the end of strings when calling printing functions – is '$'. This can be altered if another character is preferred. As it stands, the dollar sign cannot be printed in a string.

Function 111 – Print Block

On Entry: DE = Character Control Block address.

This is an alternative to Print String, as a means of creating console display. The first two bytes of the CCB block contain the address of the string to print, and the next two bytes give its length.

Function 112 – List Block

On Entry: DE = CCB address.

This is the same as Function 111 except that output is directed to the printer.

Function 152 – Parse Filename

On Entry: DE = Control Block address.

The function takes a file specification and converts it into standard FCB format. The control block consists of two pairs of bytes, the first giving the address of the input file specification, and the second of the FCB to be created.

APPENDIX E

BDOS FUNCTIONS: QUICK REFERENCE GUIDE

Note: . indicates the address of

Function Number/Name	*Input Parameters*	*Returned Values*
0 System Reset	none	none
1 Console Input	none	A = char
2 Console Output	E = char	A = 00H
3 Auxiliary Input	none	A = char
4 Auxiliary Output	E = char	A = 00H
5 List Output	E = char	A = 00H
6 Direct Console I/O	E = 0FFH/ 0FEH/ 0FDH/ char	A = char/status/ none
7 Auxiliary Input Status	none	A = 00/0FFH
8 Auxiliary Output Status	none	A = 00/0FFH
9 Print String	DE = .String	A – 00H

Function Number/Name	*Input Parameters*	*Returned Values*
10 Read Console Buffer	DE = .Buffer0	Characters in buffer
11 Get Console Status	none	A = 00/01
12 Return Version Number	none	HL = Version (0031H)
13 Reset Disk System	none	A = 00H
14 Select Disk	E = Disk Number	A = Err Flag
15 Open File	DE = .FCB	A = Dir Code
16 Close File	DE = .FCB	A = Dir Code
17 Search for First	DE = .FCB	A = Dir Code
18 Search for Next	none	A = Dir Code
19 Delete File	DE = .FCB	A = Dir Code
20 Read Sequential	DE = .FCB	A = Err Code
21 Write Sequential	DE = .FCB	A = Err Code
22 Make File	DE = .FCB	A = Dir Code
23 Rename File	DE = .FCB	A = Dir Code
24 Return Login Vector	none	HL = Login Vector
25 Return Current Disk	none	A = Cur Disk #
26 Set DMA Address	DE = .DMA	A = 00H
27 Get Addr(Alloc)	none	HL = .Alloc
28 Write Protect Disk	none	A = 00H
29 Get R/O Vector	none	HL = R/O Vector
30 Set File Attributes	DE = .FCB	A = Dir Code
31 Get Addr(DPB)	none	HL = .DPB
32 Set/Get User Code	E = 0FFH/ user number	A = Curr User/ 00H
33 Read Random	DE = .FCB	A = Err Code
34 Write Random	DE = .FCB	A = Err Code
35 Compute File Size	DE = .FCB	r0, r1, r2 A = Err Flag
36 Set Random Record	DE = .FCB	r0, r1, r2
37 Reset Drive	DE = Drive Vector	A = 00H
38 Access Drive	none	A = 00H
39 Free Drive	none	A = 00H
40 Write Random with Zero Fill	DE = .FCB	A = Err Code
41 Test and Write Record	DE = .FCB	A = 0FFH
42 Lock Record	DE = .FCB	A = 00H
43 Unlock Record	DE = .FCB	A = 00H
44 Set Multi-sector Count	E = # Sectors	A = Return Code
45 Set BDOS Error Mode	E = BDOS Err Mode	A = 00H
46 Get Disk Free Space	E = Drive number A = Err Flag	Number of Free Sectors
47 Chain to Program	E = Chain Flag	A = 00H
48 Flush Buffers	E = Purge Flag	A = Err Flag

Function Number/Name	*Input Parameters*	*Returned Values*

49 Get/Set System Control	DE = .SCB PB
	HL = Returned Word
		A = Returned Byte Block

50 Direct BIOS Calls	DE = .BIOS PB	BIOS Return
59 Load Overlay	DE = .FCB	A = Err Code
60 Call Resident System Extension	DE = .RSX PB	A = Err Code
98 Free Blocks	none	A = Err Flag
99 Truncate File	DE = .FCB	A = Dir Code
100 Set Directory Label	DE = .FCB	A = Dir Code
101 Return Directory Label Data	E = Drive	A = Dir label data byte
102 Read File Date Stamps and Password Mode	DE = .FCB	A = Dir Code
103 Write File XFCB	DE = .FCB	A = Dir Code
104 Set Date and Time	DE = .DAT	A = 00H
105 Get Date and Time	DE = .DAT	Date and Time A = seconds
106 Set Default Password	DE =.Password	A = 00H
107 Return Serial Number	DE = .Serial # field	Serial Number
108 Get/Set Program	DE = 0FFFFH/Code	HL = Program Ret Code
109 Return Code	none	
Get/Set Console Mode	DE = 0FFFFH/ Mode	HL = Console Mode none
110 Get/Set Output	DE = 0FFFFH/	A = Output Delimiter
111 Delimiter	E = Delimiter	none
112 Print Block	DE = .CCB	A = 00H
152 List Block	DE = .CCB	A = 00H
Parse Filename	DE = .PFCB	See definition

APPENDIX F

SYSTEM CONTROL BLOCK

The System Control Block (SCB) is a CP/M Plus data structure located in the BDOS. CP/M Plus uses this region primarily for communication between the BDOS and the BIOS. However, it is also available for communication between application programs, RSXs and the BDOS. Note that programs that access the System Control Block are not version independent. They can run only on CP/M Plus.

The following list describes the fields of the SCB that are available for access by application programs and RSXs. The location of each field is described as the offset from the start address of the SCB (see BDOS Function 49). The RW/RO column indicates if the SCB field is Read-Write or Read-Only.

Offset	*RW/RO*	*Definition*
00 – 04	RO	Reserved for system use.
05	RO	BDOS Version Number.
06 – 09	RW	Reserved for user use. Use these four bytes for your own flags or data.
0A – 0F	RO	Reserved for system use.

Offset	*RW/RO*	*Definition*
10 – 11	RW	Program Error Return Code. This 2-byte field can be used by a program to pass an error code or value to a chained program. CP/M Plus's conditional command facility also uses this field to determine if a program executes successfully. The BDOS Function 108 (Get/Set Program Return Code) is used to get/set this value.
12 – 19	RO	Reserved for system use
1A	RW	Console Width. This byte contains the number of columns, characters per line, on your console relative to zero. Most systems default this value to 79. You can set this default value by using the DEVICE utility. The console width value is used by the banked version of CP/M Plus in BDOS function 10, CP/M Plus's console editing input function. Note that typing a character into the last position of the screen, as specified by the Console Width field, must not cause the terminal to advance to the next line. 1BROConsole Column Position. This byte contains the current console column position.
1C	RW	Console Page Length. This byte contains the page length, lines per page, of your console. Most systems default this value to 24 lines per page. This default value may be changed by using the DEVICE utility.
1D – 21	RO	Reserved for system use.
22 – 2B	RW	Redirection flags for each of the five logical character devices. If your system's BIOS supports assignment of logical devices to physical devices, you can direct each of the five logical character devices to any combination of up to 12 physical devices. The 16-bit word for each device represents the following: Each bit represents a physical device where bit 15 corresponds to device zero and bit 4 corresponds to device 11. Bits zero through 3 are reserved for system use. You can redirect the input and output logical devices with the DEVICE command.
22 – 23	RW	CONIN Redirection Flag.

Offset	*RW/RO*	*Definition*
24 – 25	RW	CONOUT Redirection Flag.
26 – 27	RW	AUXIN Redirection Flag.
28 – 29	RW	AUXOUT Redirection Flag.
2A – 2B	RW	LSTOUT Redirection Flag.
2C	RW	Page Mode. If this byte is set to zero, some CP/M Plus utilities and CCP built-in commands display one page of data at a time; you display the next page by pressing any key. If this byte is not set to zero, the system displays data on the screen without stopping. To stop and start the display, you can press CTRL-S and CTRL-Q, respectively.
2D	RO	Reserved for system use.
2E	RW	Determines if CTRL-H is interpreted as a rub/del character. If this byte is set to 0, then CTRL-H is a backspace character (moves back and deletes). If this byte is set to 0FFH, then CTRL-H is a rub/del character, echoes the deleted character.
2F	RW	Determines if rub/del is interpreted as CTRL-H character. If this byte is set to 0, then rub/del echoes the deleted character. If this byte is set to 0FF, then rub/del is interpreted as a CTRL-H character (moves back and deletes).
30 – 32	RO	Reserved for system use.
33 – 34	RW	Console Mode. This is a 16-bit system parameter that determines the action of certain BDOS Console I/O functions. (See BDOS Function 109, Get/Set Console Mode, for a thorough explanation of Console Mode.)
35 – 36	RO	Reserved for system use.
37	RW	Output delimiter character. The default output delimiter character is $, but you can change this value by using the BDOS Function 110, Get/Set Output Delimiter.

Offset	*RW/RO*	*Definition*
38	RW	List Output Flag. If this byte is set to 0, console output is not echoed to the list device. If this byte is set to 1 console output is echoed to the list device.
39 – 3B	RO	Reserved for system use.
3C – 3D	RO	Current DMA Address. This address can be set by BDOS Function 26 (Set DMA Address). The CCP initialises this value to 0080H. BDOS Function 13, Reset Disk System, also sets the DMA address to 0080H.
3E	RO	Current Disk. This byte contains the currently selected default disk number. This value ranges from 0–15 corresponding to drives A–P, respectively, BDOS Function 25, Return Current Disk, can be used to determine the current disk value.
3F – 43	RO	Reserved for system use.
44	RO	Current User Number. This byte contains the current user number. This value ranges from 0–15. BDOS Function 32, Set/Get User Code, can change or interrogate the currently active user number.
45 – 49	RO	Reserved for system use.
4A	RW	BDOS Multi-Sector Count. This field is set by BDOS Function 44, Set Multi-sector Count.
4B	RW	BDOS Error Mode. This field is set by BDOS Function 45, Set BDOS Error Mode. If this byte is set to 0FFH, the system returns to the current program without displaying any error messages. If it is set to 0FEH, the system displays error messages before returning to the current program. Otherwise, the system terminates the program and displays error messages. See description of BDOS Function 45, Set BDOS Error Mode, for discussion of the different error modes.
4C – 4F	RW	Drive Search Chain. The first byte contains the drive number of the first drive in the chain, the second byte contains the drive number of the second drive in the chain, and so on, for up to four bytes. If less than four drives are to be searched, the next byte is set to 0FFH to signal the end of

Offset	*RW/RO*	*Definition*
		the search chain. The drive values range from 0 – 16, where 0 corresponds to the default drive, while 1 – 16 corresponds to drives A – P, respectively. The drive search chain can be displayed or set by using the SETDEF utility.
50	RW	Temporary File Drive. This byte contains the drive number of the temporary file drive. The drive number ranges from 0 – 16, where 0 corresponds to the default drive, while 1 – 16 corresponds to drives A – P, respectively.
51	RO	Error drive. This byte contains the drive number of the selected drive when the last physical or extended error occurred.
52 – 56	RO	Reserved for system use.
57	RO	BDOS Flags. Bit 7 applies to banked systems only. If bit 7 is set, then the system displays expanded error messages. The second error line displays the function number and FCB information. (See Section 8.3.13). Bit 6 applies only to non-banked systems. If bit 6 is set, it indicates that GENCPM has specified single allocation vectors for the system. Otherwise, double allocation vectors have been defined for the system. Function 98, Free Blocks, returns temporarily allocated blocks to free space only if bit 6 is reset.
58 – 59	RW	Date in days in binary since 1 Jan 78.
5A	RW	Hour in BCD (2-digit Binary Coded Decimal).
5B	RW	Minutes in BCD.
5C	RW	Seconds in BCD. 5D – 5EROCommon Memory Base Address. This value is zero for non-banked systems and non-zero for banked systems.
5F – 63	RO	Reserved for system use.

INDEX

CP/M Control Characters for CP/M Editing

Keys	Meaning
CTRL-C	System restart(warmboot),restores system prompt.
CTRL-E	Moves cursor to next line to continue command line (without executing or transmitting line).
CTRL-H	*Backspaces cursor to erase last character typed.
CTRL-I	Moves cursor a "tab"space(7 columns long).
CTRL-J	*Performs a RETURN
CTRL-M	*Performs a RETURN
CTRL-L	Replacement for Carriage Return sequence generated by RETURN in strings used with search and substitute commands.
CTRL-R	Retype current line (types a clean line)
CTRL-U	Delete current line
CTRL-X	*Backspace to beginning of current line and erase line.
RETURN	Transmit(execute)the current line, or generate a Carriage Return to seperate lines of text file.
DELETE	Delete the last character typed (echoes the character).
CTRL-Z	Terminate the I commands inserting,or separate strings of text vin search and substitutions, or place as marker at end of text file.
CTRL-P	Echo everything typed or displayed at the line printer.
CTRL-S	Temporarily halt a long display(strike any key to continue display).
BREAK	Discontinue execution of currently-executing ED command.

AN INTRODUCTION TO CP/M PLUS ON AMSTRAD COMPUTERS

P. K. McBride

The aim of this book is to provide a complete introduction to CP/M Plus as available on many Amstrad computers.

The book begins with an explanation of the nature of CP/M as an operating system and traces its history and development as well as explaining the difference between CP/M versions. The basic system is explained with coverage of memory allocation (CCP, BDOS, BIOS, TPA) and system commands (DIR, USER, RENAME, ERASE, SAVE, TYPE). The book then proceeds to coverage of useful utilities such as STAT, FORMAT, PIP, SETUP and SYSGEN; text editors and word processors; writing software under CP/M; high level languages and CP/M; CP/M Plus; graphics under CP/M; installing programs; adapting the system. The book then concludes with an overview of available programs and cost via the User Group as well as coverage of commercial software. Appendices include Z80 to 8080 code conversion table and comments; Keyboard control summaries – CCP and ED; and the BDOS call chart.

About the author
P. K. McBride has a wealth of experience in writing not only computer books, but also games and commercial software. He is editor of the Daily Mail Home Computing Guide as well as author of the companion book to this title, Choosing and Using CP/M Business Programs.

£7.95

GLENTOP
PUBLISHERS LIMITED

Glentop Publishers Ltd
Standfast House
Bath Place
High Street Barnet
Herts EN5 5XE

ISBN 1-85181-054-4

9 781851 810543

INTRODUCTION TO CP/M PLUS ON AMSTRAD COMPUTERS McBride

GLENTOP
PUBLISHERS ■ LIMITED