Master 512 Technical Guide

by Robin Burton

DABS PRESS

Master 512 Technical Guide

ISBN 1-870336-80-1
First edition, first printing September 1990
Editors: Syd Day, Bruce Smith, David Atherton
Typesettng and proofreading: L Atherton, J. Horrocks, A. Wygladala
Cover Design. Atherton Clare Designs (0204 654207)

The publishers wish to acknowledge with gratitude the assistance provided by Acorn Computers Ltd. in the preparation of this book, without whom publication would have been impossible, and for allowing the publication of the Master 512 circuit diagram. The publishers wish also to thank Richard Russell for allowing his BBCBASIC(86) program to be included on the programs disc.

Published by Dabs Press, PO Box 48, Prestwich, Manchester M25 7HF
Telephone 061-7738632. Fax 061-7738290.

Typeset in 10/11 point Palatino by Dabs Press using an Apple Macintosh desktop publishing system.
Author's text prepared on a Master 512 system using PC-Write version 3.0.

Printed and bound in the UK by BPCC Wheaton, Exeter, Devon EX2 8RP.

Master 512 Technical Guide

Contents

Appendices

F: 6502.SYS code

6502.SYS source listing
OSWORD &FA source listing

G: Hardware Projects

Clock add on
Master 512 RAM expansion board
Building your own hard disc

H: Third Party Products

Beebug

512 articles

Dabs Press
Essential Software

Various utility packages
512K expansion board

Interactive Software Services

Fortran development package

Margolis & Co.

Communications Software

Permanent Memory Systems

Real Time Clock

Shibumi Soft

Problem Solver

Introduction

Purpose and Scope

The intention of this book is to provide a definitive reference work which provides information to those who wish to know more about the specifics of the symbiosis between the Master 512 co-processor and its BBC Micro host.

This volume is concerned mainly with details of both the hardware and software interfaces between the two systems, examining how the system operates internally rather than how to use it. It complements its companion volume, the *Master 512 User Guide* by Chris Snee, also published by Dabs Press, which details day-to-day operation of the system at the user interface.

It is assumed that you are conversant with the essential functions carried out by the host processor's Operating System in native mode since many of these facilities are used directly, either in their standard form or modified, to allow the 512's Operating System to communicate with the outside world. Familiarity with 6502 assembler code is necessary to allow a full understanding of the machine code programs which reside in the host, and which collectively are responsible for handling all inter-processor dialogue across the Tube, the data bus which connects the two machines.

While all the appropriate host operating system calls are fully documented, the information supplied relates to the function as used by DOS Plus. Details of how these MOS calls are implemented within the host can be found in BBC Micro technical publications such as the *Advanced User Guide* by Bray, Dickens and Holmes and *Master Operating* System: *A Dabhand Guide* by David Atherton.

It is assumed that you are an experienced user of DOS Plus and the 512, and explanations of standard user operations are not included in this book. Knowledge of 8086/80186 architecture or assembly language, while being helpful, is not essential. Such technicalities of the operation of the 80186 as are needed are explained, though it is not the intention of this book to teach you to write assembly code programs for DOS Plus.

We are unable to reproduce the source code for any of the versions of DOS Plus as used by the 512. The material is the property and copyright of Digital Research, not of Acorn Computers. This is further complicated by the fact that four different versions have actually been issued. Inclusion of such listings, quite apart from taking up more space on their own than the total size of the book, would, with the addition of copyright licenses, have increased the cost of this volume by an unacceptable amount.

There have been several versions of DOS Plus for the 512, version 2.1 being the latest issue. All detailed references to the functions of DOS Plus are based on version 2.1, and while many functions may be identical from the user point of view, the code itself or the memory addresses concerned can differ

considerably from version to version. Users of earlier DOS Plus versions should take this into account.

Terminology

Throughout this book the use of technical terminology is unavoidable, but to experienced programmers of the 6502 only a few new concepts will be introduced. The main difficulty of understanding the operation of the system is likely to stem from the fact that, at times, two different processors must be considered simultaneously.

To avoid confusion between the two processors, operations that take place in the BBC Micro, regardless of the variant employed, will be referred to as 6502, BBC, host or MOS operations, while those relevant to the 512 will be referred to as 512, 80186 or DOS functions. Where differences in the host's operation are relevant to the type of machine employed (Model B, B+ or Master 128) these will be highlighted.

Within the book the term '512' will variously refer to software and hardware in the Master 512 system, and the term 'PC' will variously refer to the hardware and software systems of standard PC clones.

The *raison d'être* for this book is that the Master 512 is *not* a standard clone, and as such conventional technical PC books are inappropriate.

Acknowledgements

Both the author and the publishers would like to express their sincere thanks to Acorn Computers Ltd. for making available a great deal of copyright material, much of which was previously Acorn restricted and without which this volume could certainly not have been produced.

In particular thanks are due to Andy Smith, who, although he has now moved on to pastures new within Acorn, has until recently and for quite some time been virtually the sole 'front-line' support available to the ten thousand or so users of the 512. We should all be grateful.

His has been the unenviable task of handling the updates to, and production of, many hundreds of copies of the 512 Application Notes (including the Applications Compatibility List) and the Technical Reference Notes, from which many users and this book have benefited.

Without his efforts in extracting, collecting and collating (and sometimes finding) much of the Acorn material required for this book, the result would have taken even longer and might very well have proved impossible.

It would also be remiss of me to fail to mention that he is largely responsible for designing, building, testing and documenting the prototype of the 512K byte memory expansion, together with the other hardware projects we are pleased to be able to include in this book.

Finally I must offer my thanks to both you, the reader, and to Dabs Press for your patience. That this volume has been longer in preparation than was originally intended or envisaged is no secret. I would be less than honest were I not to admit that virtually all the delays up to the present time are entirely my responsibility [sincerely hope that you find the wait was worthwhile.

Robin Burton
Leicestershire
September 1989.

1: System Overview

This part of the book serves as an introduction to the various separate elements of the 512 hardware system. The software techniques employed to connect the hardware together into a working DOS Plus system are considered in detail in subsequent chapters.

Functional Requirements

The 512 second processor consists of a single, self-contained circuit board with only one means of communicating with the outside world, via the connecting pins at either end of the board. It follows that all support for these functions must fall upon the host processor. The 512's host is responsible for providing all peripheral connections and services, including the screen, discs, printer, keyboard and mouse.

While it is true that this arrangement has some disadvantages and limitations, it also provides a highly efficient system by dividing the workload between two separate processors, which can, to some extent, work independently and simultaneously. The extremely respectable performance of the 512, particularly in some benchmarks, is due in part to this two-processor configuration.

Host Connection

The 512's host processor may be a BBC model B, B+ or a Master 128. In all three cases (and it is the only possibility for the first two) the 512 may be fitted into an external housing such as an Acorn Universal Second Processor Unit or some other proprietary co-processor adapter. In this case the second processor housing must also include a power supply unit to cater for the 512's 5 volt needs.

When fitted in this way, the only physical and electrical connection between the 512 and its host is provided by the external tube. This is an Acorn proprietary fast data bus, rated at 2 Mhz (theoretically up to two million bits per second) and 32 bits (four bytes) wide. The tube is provided specifically for second processor connection, though not only for the 512, as it has been used for *at* least four other commercially available boards. (The others are 6502, Z80 CP/M, 32016 scientific and Acorn RISC Machine development Systems).

Uniquely, in the Master 128, there is a second option of using the internal tube, when the 512 does not require any additional housing since it is fitted directly to the Master's mother board within its own case. In this type of installation the 512 also takes its power from the Master's power supply unit. Because there is no separate on-off switch by which to enable or disable the 512, software switching is employed, provided by the host's MOS. Selection between the external and internal tube is achieved by the Master's `*CONFIGURE` command, as is enabling or disabling the tube interface itself.

While there is no essential difference between the host's support of the second processor whichever connection is used, two particular points are worthy of note when the internal tube is employed.

First, the 80186 processor chip in the 512 becomes extremely hot after even a short time. This is completely normal, but after an hour or so it is likely to be too hot to touch. This does very significantly increase the ambient temperature within the Master's case, and any ICs that may be on the limits of their operating temperature tolerance have an increased probability of transient failure. It is vital that the ventilation slots in the case are not obstructed.

Cases are known of users who, after fitting a 512 internally, begin to experience problems after an hour or two of operation, even when the 512 is not in use. This is because the 512 consumes some electrical power whether it is selected by software or not.

There are two solutions to the problem. One is to have suspect ICs replaced, but identification can prove difficult and expensive. The other is to switch the connection of the 512 to the external tube. Similar problems are rarely experienced when a 6502 second processor is fitted internally, as these ran very much cooler.

It is worth noting that the WD1772 floppy disc controller fitted to some Masters has proved to be more troublesome than other chips. If problems are experienced which seen to be related to the floppy disc system after fitting a 512 (especially internally) changing to the more reliable, though slightly slower, WD1770 may well provide a solution.

The second point about the internal tube is that fitting one type of second processor to it does not preclude the simultaneous attachment of yet another, connected via the external tube. It is possible to have both a 6502 and a 512 second processor permanently attached, switching between them as required, with the command *CONFIGURE
EXTUBE/INTUBE.

Limitations

The 512 with DOS Plus 2.1 runs virtually all application packages which **are** legally **written** to **be** compatible with MS-DOS version 2.1, **PC-DOS** 2.1 or CP/M86, which was the forerunner of DOS Plus. Arguably the most notable aspect of the 512 is that much of the software written for PCs, which have very different architecture, does run satisfactorily.

Inevitably some software will not function correctly or even at all in the 512. The Dabs Press *Master 512 User Guide* contains a good overview of the general problems and some of the steps you can take when trying out the compatibility of a new package.

Memory Size

Not surprisingly, some incompatibilities are more easily predicted than others. The prime requisite is that the package does not require more memory than the 512 can provide. This is typically about 358k bytes in an empty 512 running DOS Plus 2.1 and no other program, but it can be further reduced by background programs, the memory disc and other memory-resident utilities.

Some more recent packages are written to expect 640k, which is virtually the standard memory size for PCs these days. This problem is further compounded by the fact that DOS Plus takes about 90k more memory than MS-DOS for its memory resident portion. The result is that, even if software is suitable for a 512k MS-DOS machine, it's still entirely possible that it wilt not run in the 512 because of lack of memory.

Expanded memory 512s offer more directly addressable memory than any true PC so, however severe this problem is, it can be overcome. The PC Plus by Solidisk is no longer available (except second hand), therefore a 512k bytes expansion project is included in this volume to overcome this problem.

Memory constraints aside, if software claims compatibility with DOS 2.1, the majority of any remaining problems stem from the limitations imposed by the two processor hardware configuration of the 512 system, or to the capabilities of the 6502 host. Most of these problem areas can be identified and are virtually all concerned with screen handling, the keyboard or discs.

Screen Facilities

Screen support provided by the host 6502 is naturally limited to the facilities that are available when running in native BBC mode. Despite the fact that the 6845 CRTC (Cathode Ray Tube Controller) employed in the BBC Micro is the same chip as is used by machines, the display facilities available are not the same.

The major difference between the screen displays provided by the 512 and other DOS Systems is the restricted colour capability, especially when operating in 80 column text mode. In these modes the BBC Micro uses its native screen mode 3, which is a single colour mode. Although the 512 offers a CGA (Colour Graphics Adapter) compatible screen map to DOS applications, the true PC CGA screen of 80 columns in colour is ultimately restricted by the BBC mode 3 display.

The Keyboard

Keyboard support by the 6502 host can offer very particular problems in DOS, especially if a Model B or B+ is used. The prime difficulty is that the keyboards employed by PCs simply have more keys than those of BBC Micros. More recent PCs also have more programmable function keys, fifteen or twenty now being common.

Some, though by no means all, recent applications software expects to take advantage of these extra keys and, unless such software is configurable for earlier PCs, it will be difficult, if not impossible, to run satisfactorily in the 512.

Some of the extra keys provided by PCs are included for highly specialised purposes, directly related to hardware, or more properly firmware functions of other parts of the machine. In the BBC Micro no such hardware to firmware dialogue is provided, all keyboard scanning being entirely under the control of the host's MOS. To illustrate, the nearest equivalent operation in the BBC Micro is the effect of the shift and control keys which, when pressed simultaneously will normally prevent the screen from scrolling regardless of the application used.

In addition, the method of translating keyboard hardware matrix scans to an internal key representation is entirely different in PCs. Even for those keys that are common to both the BBC Micro and PCs there are differences because the actual internal number produced by most key presses is different. For the 512 system all key-presses passed across the tube must first be translated by software to the required IBM code before being passed onto DOS applications.

It follows that an application expecting to detect an IBM key at source by reading the keyboard hardware matrix will fail, but so will any package which expects to respond to one of the extra IBM keys, even if the legal DOS keyboard facilities are used.

For example, in most PCs the left and right SHIFT keys generate two different internal key numbers. While entering text into a word processor there may seem to be no operational difference between these two keys, but at other times they can serve entirely different purposes. Both keys can be detected and distinguished separately in PCs either by scanning the keyboard hardware matrix (strictly speaking illegal, but it doesn't matter in a PC) or by reading the infernal key number using legal DOS calls. Applications that use the first of these methods will always fail, while those that require the extra keys will fail to respond correctly (or at all) in the 512.

This type of low-level non-ASCII key detection is not possible with the BBC Micro because the keyboard matrix is much simpler and does not provide the fall range of PC keys, nor does it provide any hard-wired functions. Also the keyboard is on the wrong side of the tube for direct access by DOS or any application, therefore the keyboard is not directly addressable as it is in true PCs.

Model B or B+ Micros have additional problems in that they use an even more simple keyboard than the Master. Even for those PC keys which are emulated on the Master's numeric keypad when used in conjunction with shift-LOCK, the results may not always be reliable because they depend on the method of access employed by the application as well as the key number expected.

Discs

The Western Digital 1770 or 1772 FDC (floppy Disc Controller) employed by Acorn for ADFS is

eminently suitable for use with DOS disc formats but there are one or two specific points to note.

DOS expects two disc drives for a practical usable system. It is possible to run DOS Plus with only a single drive, provided that all applications leave enough free RAM for an adequate memory disc. In practice this would be highly restrictive, therefore to all intents and purposes two physical drives should be regarded as the minimum.

For floppy disc drives both must be 80 track double sided. There is no longer such a thing as a single sided DOS disc drive, although the ancient and rarely seen 160k discs were single sided 40 track devices. Although IBM 360k formats use 40 tracks, it should be noted that even if switchable drives are fitted to the host, they should always be set to 80 track operation as all format detection and track double stepping is catered for in the 512's software.

The third real disc that may be attached is a hard or Winchester disc. This runs as drive C:, and optionally it may or may not be the boot disc. Because of the non-standard (as far as DOS is concerned) interfacing of the Winchester to the host, control of drive C. is left to the ADFS ROM in the 6502. In the source for 6502.SYS it can clearly be seen that the Winchester control code has not been implemented.

Although CP/M (on which DOS Plus is based) is capable of supporting up to 13 physical disc drives, in the 512 implementation only two floppy drives (limited by the host), one hard disc and one memory disc are supported.

Apart from the hard disc and the memory disc, the formats of which are not variable from DOS's point of view, the MOS contains suitable code for support of several physical floppy disc formats as used by different versions of PC compatible micros. There are limitations to the effectiveness of this dynamic software configuring, forced by the host's hardware.

A common requirement is that of using discs to transport data a PC and the 512. The 512 is more flexible than most PCs in the range of disc formats supported and will successfully read and write the majority of PC discs. One might hope the task would prove to be straightforward, but the operation is almost certain to encounter problems if the discs are formatted by the 512 rather than the PC in question. Even though the format may be nominally of the correct specification and the 512 verifies that the disc is error free, successful reading or writing of the disc by a PC is unlikely.

Although this appears at first to be a serious problem, it is quite simple. The size of the inter-sector gap written by the host's WD1770/2 when formatting is too small for the majority of PCs, which are unable to read such discs. The solution is equally simple. If discs used to transport files are formatted by the PC, both the 512 and the PC will be able to successfully read and write the disc.

This assumes that both a disc size and format is available which is common to both the PC and the 512. For both sizes of disc (5.25" and 3.5") this will be either 360K (also called single density) or 720K (double density) format. It also means that if files are transported between several machines you may

need several discs, one for each, as not all PC disc formats are compatible with each other.

Under no circumstances will the 512 read or write either of the two high-density (HD) formats, which are now available on recent PC machines. These are 1.2 Mb for 5.25" and 1.44 Mb for 3.5" discs. This is not usually a problem, since such machines will also support both single and double density formats. However, it is possible that, even with their higher specification drives and faster controllers, these machines will still reject 512 single or double-density formatted discs-

The success of the technique of formatting discs on a target PC is not guaranteed for all machines. Some PCs appear to support standard disc formats, but will themselves produce discs that the 512 will neither read nor write, or more dangerously, will not read or write reliably.

The second possibility usually arises because the PC has a slightly non-standard catalogue or it stores the file allocation table in a different internal format. The disc may verify and can even respond to a DIR command correctly with no error reported. It may only be later, when you find that the information read from a file bears no relationship to the file's true contents that the problem becomes apparent. Always check by writing a couple of known files from each machine and reading them successfully on the other before you commit to this method of data transfer on an unknown machine.

Equally, some machines which appear to support a recognised standard disc format, produce discs which cannot be read at all by the 512. There are certain 360K and 720K PC formats which a real IBM PC will not read either.) The only definite statement that can be made on the subject of disc formats is that not all PC compatibles are compatible and trial and error is the only true test.

If data transfer is required between the 512 and a PC but cannot be achieved directly via a common disc format, two remaining possibilities exist. A third PC might produce a format which can he read and written by both of the other two. This strange situation arises because some PCs cannot be instructed to format IBM standard discs, but will correctly sell-configure to read or write them, like the 512, when the discs are already formatted.

If no other solution can be found, an alternative method of data transfer that might prove acceptable is via the serial port, either directly connecting the 512 to the PC or at a distance by means of a modem. (See RS232 below).

The Printer

Fortunately printers hold few problems, either for DOS or the 512. The vast majority of recent printers provide either an optional IBM mode (usually IBM Proprinter X24 emulation) or they provide adequate IBM block graphics character set support. The standard DOS Plus background print utility operates quite satisfactorily in the 512, with the small condition that you disable the printer's automatic line feed if you wish to avoid double spacing.

Whether you do this by means of the printer's DIP switches or the host's printer ignore character is a matter of personal preference, though regular users of the 512 will find it more convenient to disable the line feed in the printer's hardware, then issue (or configure)

```
*FX6, 0
```

in the BBC host in native mode, or *CONFIGURE IGNORE 0 on a Master. This avoids the need for use the relatively tedious and slow STAR command when the 512 is active.

The standard printer connection for DOS Systems is a Centronics (or parallel) interface. In DOS Plus the logical print device is PRN0, while in MS-DOS or PC-DOS it is usually LPT1. This may give rise to problems with some software in the 512, for example, GW-BASIC.

Within applications, owing to the proliferation of PC compatibles, a printer configuration section is included in the installation process of most packages. Generally such packages write to the printer legally, as it is a slow speed device, and leave the logical to physical device assignment to the operating system. However, in a limited number of cases it may be necessary to change the printer device assignment within the application. If so this should be set to PRN0 (or LPT0 if PRN is not accepted). The package should then print correctly.

A useful tip with GW-BASIC in which the LPRINT command will not work on a 512, is instead to 'open' the printer with the command

```
OPEN "PRN" FOR OUTPUT AS #1
```

at the start of the program, and then to use PRINT #1, a$ instead of LPRINT as when you need to print to the printer. If your program uses files, then the channel number may have to be changed from 1 to another free channel number.

The more capable packages also will include a list of different printers from which your model, or a 100% control code compatible alternative can be selected, ensuring correct control codes will be sent for each of the various typeface effects and enhancements. More recent packages will also cater for a range of laser printers too.

Serial connection of printers is not directly supported, either by DOS or by applications software. As is normally required when in native mode, the speed should be set correctly, either by using STAR or the MODE command. Also printer output should be directed to the serial port either by means of STAR again, or by logical to physical device assignment using DOS's DEVICE command.

If printing from an application proves impossible with a serial printer, you may find the package provides a print to file or a save ASCII data option. If no such option is provided, it may be possible to use DOS's re-direction facilities to achieve effectively the same result. In either case the resulting file

can be printed afterwards, by means of the standard DOS Plus PRINT command. This last 'trick' can also solve problems with some packages that refuse to communicate with 'PRN' or LPT0. If this proves unacceptably inconvenient or impossible, the only alternative is to change the printer's interface to parallel.

The RS232 Serial Port

Like the printer's Centronics interface, the RS232 serial port is a standard communications connection, which has been adopted by micros, rather than the reverse. There are few problems concerned with the 512's serial port except in the area of proprietary PC software.

Support for the RS232 port is notably weak in MS-DOS and PC-DOS, and access is very slow via the legal calls. In consequence, most communications software authors find it necessary to write their own hardware driver code. Because of this, virtually all DOS communications software uses illegal direct hardware addressing to provide more sophisticated facilities and to permit the use of the higher speeds. Since the 512's serial port is located on the far side of the tube it is impossible to use such techniques in the 512 as they are guaranteed to fail. There is no possible fix for this problem, so do not invest in DOS communications software, except as mentioned below, or which you have seen working in a 512.

Margolis & Co are the only company to have produced a communications package that operates legally and satisfactorily in the 512. A brief description and report of the package is included later in the book, together with the address for inquiries.

The well known public domain file transfer system, KERMIT, originally produced at Columbia University in the USA and available in the UK from Lancaster University, also works satisfactorily in the 512 if PC version 2.32 is used. KERMIT is produced for a vast array of machines, from home micros like the BBC up to and including all types of mainframe.

Proprietary communications software apart, there are no problems with the 512's serial port except those which also apply when the host is used in native mode.

The only points of relevance are that the physical connection in the BBC Micro uses a non-standard DIN type plug and the lead must either be made up from components or purchased from an Acorn dealer. Standard RS232 leads from other sources are of no use, as they will usually have a 25 way connector fitted at both ends.

The Acorn implementation does not conform to the full RS232C communications standard either, being a cut down version. In practice this is little problem. For connection to a modem it is usually possible to achieve success by using only three of the five available connections wired null modern, data-in to data-out, data-out to data-in and ground to ground.

If handshaking is not required, as for example with the 512 and are directly connected back-to-back so

that each machine's operator can know what the other is doing, the three-wire connection can also usually be used successfully to transmit files in either direction. Some machines will insist on a more sophisticated connection, when both the RTS and CTS will be needed too, and possibly one or both of these may also need to be connected to the DTE and DTR pins of the other machine. In this area PCs often differ, and some persistence with experimentation may be required before success is achieved.

The Mouse

Of all the facilities that might have been omitted from the 512, a suitable mouse driver is the one, which causes the most complaint. This is especially unfortunate since production of a mouse driver would have been much easier for the original supplier, but given the size of the 512 market and the decisions taken when customising the XIOS, it is commercially much less attractive after the event.

Although mouse driver software is included on the issue discs for the 512's GEM collection, it is neither a separate mouse driver, nor would it be suitable for use with anything but GEM if it were. The Acorn GEM mouse driver is a one-off program produced purely for GEM in the 512+ In fact there are two programs, depending on whether you're using colour or black and white.

Because there is no mouse port in the 6502 host, the user port was pressed into service. This introduces two problems for conventional DOS mouse software. First, there is no physical mouse port to address (it's the far side of the tube again), and if there were it would not have the normal characteristics. This means that no standard DOS mouse driver, whether generic or supplied with an application, will work successfully in the 512+.

Because of this, a further step was taken. In the interests of reducing the memory resident portion of DOS plus, since it would not be used, all the mouse pointer code was omitted at source, along with the mouse interrupt processing code.

The result is that implementing a mouse driver for the 512 involves writing the code to read the mouse's movements from the XIOS, decoding them and passing them on to an application via an interrupt which must also be implemented by the program.

Tull Computer Services, whose address is given later in the book, have completed this considerable task.

The 80186 Processor

The 512's microprocessor is an Intel 80186, which runs at a clock speed of 10 MHz. The 80186 is object code compatible with the 8086 processor, though there are extensions in the 80186 instruction set which are not present in the 8086. There are two later processors in this series, the 80286 and 80386. Code produced specifically for them or compiled on them to use the extra facilities will not run in an 80186 machine.

The most notable omission from the 80186 chip, when compared with some of its contemporaries, is the absence of support for an 8087 maths co-processor. This is a hardware floating-point processor, which has its own range of separate instructions. When a machine fitted with an 8087 performs large calculations, the program can be written so that both processors can execute their own code, to a large extent, independently.

The only implication of the lack of an 8087 should be that a system would run slowly when performing large or complicated calculations. The presence or absence of an 8087 can be detected in software, so applications which can use the 8087 should check for its presence and self configure in it's absence to use overlaid software routines instead. A few calculation intensive applications do not verify the presence of the 8087 and cannot be run in the 512. These will crash as soon as an 8087 instruction is encountered.

Programs in this category the heavyweight statistical analysis and modelling routines or pure scientific applications. It must be assumed that the authors expected that no one would attempt to run such applications in a machine without an 8087 co-processor and did not feel the need to check for it.

Processor Registers

The 80186 is a 16-bit processor, which means that each of its registers are 16 bits, or two bytes, wide. This has several direct implications for the capabilities of the system as a whole, governing factors as diverse as the total amount of memory that can be addressed, how so phisticated individual machine code instructions can be, and how comprehensively the processor can monitor events in the system by means of its range of status flags.

A further ramification is the maximum number of possible instructions in the instruction set. In the 6502, which can manipulate only two or three bytes at a time, the first byte always indicates the opcode, hence there could never be a repertoire of more than 256 instructions. The 86 series of processors has no such restrictions and more instructions have been added with each succeeding version, which is why code produced specially for the 80286 or 80386 processors is incompatible with the 80186.

There follows a brief introduction to the 80186 processor's architecture and an outline of some of its

instruction set's capabilities. This is not a programming course, but 6502 assembly programmers will be able to contrast the 80186 with the 6502, so it may serve to whet some appetites.

The registers essentially fall into four categories, which are general purpose, segment, index (also called offset) and status.

General Purpose Registers

In the first category are the registers, which are used for arithmetic and testing of values and results. The four general purpose registers are called A, B, C and D, but they can each be used both as a single two byte (word) register, or as two separate single byte registers. Depending on the current operation they have modified names in programs to indicate their current mode of use.

Note that there is no switch of mode. The portion of the register affected or used is controlled entirely by the form of the name used in each individual instruction in a program. This may vary from one instruction to the next.

When used as word registers, A, B, C and D are suffixed by x and are referred to as AX, BX, CX and DX. When coded into program instructions by these names, all sixteen bits in the register are involved in any operation.

Although there are two bytes in each of these registers, when used for memory address manipulation it is convenient to be able to manipulate the high address byte and the low address byte independently. When byte operations are performed a means is required to indicate which half of which register is to be used. The convention is to replace the x in each name by an H to refer to the high byte, or an L for the low byte. AX, BX, CX and DX therefore become AH and AL, BH and BL, CH and CL and DH and DL respectively.

This technique gives effective access to 8 single byte registers or four word registers, depending only on what they are called within each program instruction. Examine the following 80186 assembly instructions, noting that in 86 assembly code the target operand is always stated before the Source operand. That means that the second operand, the numeric value, is added to the first, which in each case here is a register:

```
ADD AL,1h
```

is the same as:

```
ADD AX,1h
```

since both of these increase the (low order byte) value of AX by one, but consider:

```
ADD AH, lh
```

which would be the equivalent of:

```
ADD Ax, l00h
```

In the example above, the second instruction would achieve the same result as the first, but would take more memory and time to do it. The h after a number is the 86 convention to indicate hexadecimal values and is directly equivalent to the '&' prefix used in native BBC notation, hence &FF and FFh represent precisely the same value. Some assemblers also accept a leading zero to indicate hex values so 0F would be taken to mean the same as Fh or 0Fh.

Most assemblers also accept decimal number literals, some requiring a suffix 'd', some assemblers that if neither a leading zero nor trailing 'h' is supplied the value is decimal. The last instruction above could equally have been written (in its 'long' version) as:

```
ADD AX, 256d
```

or:

```
ADD AX, 256
```

Upper case has been used here to clarify instructions and register names, but most 86 assembly language programs are written mainly in lower case, though like DOS Plus commands, either will be accepted. Lower case is used because there is a generally recognised programming convention that constant values or external references, such as literal identifiers and library filenames declared at the start of a program are written in upper case, while memory addresses (i.e. labels) whether used as jump destinations in the code or as data storage, are in lower case.

The idea is that within the body of the code it is immediately obvious at a glance whether a name refers to an external area or a fixed constant value, or to the address of a local memory location, the contents of which are subject to change.

Segment Registers

The second register category is segment registers. These are used to contain the number of the memory block that is the segment part of the RAM address for memory addressing operations (see Segmentation below). The four 16-bit segment registers are CS, DS, ES, and SS. These registers cannot be addressed as two separate bytes and so the following are the only recognised names for these registers. The segment register names mean:

```
CS Code segment
DS Default segment
SS Stack segment
ES Extra segment
```

By convention three of these registers have agreed uses within programs, but the purpose of Cs is fixed by the processor. CS always points to the segment in which code is currently being executed. In fad all the segment registers are set to this value automatically when a .COM or .CMD program first loads (as is the general purpose register DX), though they may be altered within the program or during the initialisation process.

CS cannot be directly altered, but a program jump to a different segment (usually initially set up in ES) automatically causes the value in CS to change appropriately.

Depending on the particular assembler used, it may be necessary to make declarations at the start of source code about initial segment registers set-up. This can often be seen in 86 assembler source program as a series of 'ASSUME' statements, which may look like this:

```
ASSUME CS = CODE
ASSUME DS = CS
ASSUME SS = CS
ASSUME ES = nothing
```

DS is under program control and usually initially defaults to the code segment. When used to indicate a different segment, DS can be pointed to a segment of memory where the program's data is stored, like the currently used part of a document in word processing. For most program operations involving addressing, if no other segment register is explicitly supplied the segment value in DS is used.

The stack segment is analogous to the stack page in the 6502, but it can be much larger. Programs in DOS have a local stack, that is, one unique to the program, and this may or may not be in the current code segment. This technique is employed to ensure, as far as possible, that no matter what happens within a program, nothing else in the machine should be affected. It is also necessary because, without local stacks, multiple simultaneous program operation, like foreground and background tasks, would be impossible.

The stack segment initially defaults to the code segment, but it can be located elsewhere if required. The maximum size of a program's stack is unlimited, but in practice is usually confined to the maximum address range of 16 bits (64k) and it can be anywhere within its available memory that a program chooses to put it. If required you can allocate a separate segment entirely for the stack. In this case the segment register is program controlled.

The extra segment register, ES, is conventionally used to point to any other area of memory not within a

currently identified segment. For example, with Ds pointing to your current working data segment, you might set ES to point to another segment from which you wish to transfer data into the DS segment. At the same time CS points to the current code segment, while SS might point to a stack segment for temporarily stored data that you may need to retrieve later.

All segment registers are used in conjunction with a two-byte offset to address the memory location within the segment. This can be a hard coded literal value, or it may be a derived or calculated value contained in a memory location or one of the index registers. In the case of CS, the offset address is always indicated by the instruction pointer (See IP), the contents of which cannot he varied directly.

Index and Pointer Registers

The two index registers, SI and DI are the third category of register. These are both 16 bit registers and can be used in conjunction with the segment registers. When used together, a segment register and a segment offset can define any address in any part of the memory. For this reason the index registers are sometimes also called offset registers.

The fourth register category is the pointer registers. These are also 16 bit register and are called:

```
SP Stack pointer
BP Base pointer
IP Instruction pointer
```

The stack pointer holds the address of the next free stack entry in the stack segment. The stack pointer must be initialised by the program to point to the top of the space reserved for the stack if the programmer wishes to specify a special stack area. On initial load the stack pointer is set to the address of the end of the code segment minus two bytes. Stack space is used from the top downward in the same manner as the 6502's stack. Note that it is up to the program itself to allocate stack space and to point SR to it (and SS too if necessary) before The stack is used. For example the instruction:

```
PUSH A
```

executed with SR set at 1000h would push the contents of registers AX, CX, DX, BX, SP, BP, SI and DI onto the stack and decrement SR by two for each, leaving it pointing to 0FF0h.

The base pointer is used as a pointer into memory, often as a sort of memo to an address in a local table, or as the starting value for an index register which will be varied. It is entirely under program control and is mainly intended to be used to address local variables stored on the stack. For this reason the default segment register for BP is SS, not Ds as it is for most other operations.

There is a third pointer register, the implicit instruction-pointer, IP. Arguably IP is not a register in the usual sense as, in conjunction with CS, if points to the address of the currently executing program

instruction. Like Cs it cannot be changed directly, but only as a result of executing jumps or returns within the program code.

The last register, the status register, is also known as the flags register. It performs the same function as in other processors. Various bits are set or unset as a result of arithmetic operations, comparisons and moves. These status bits, or flags, can then be used to govern the actions of following conditional instructions.

There is one particularly interesting extra feature of the 86 status register over that of the 6502. One of the flags is the direction flag. As will be seen shortly, numerous automatic indexing and counting instructions exist and two instructions, 'STD' (SeT Direction) and 'CLD' (CLear Direction) set or clear this flag respectively. The setting of this flag dictates whether the automatic increment for certain instructions is positive or negative. The word 'advancing' is conventionally used to denote either of these in describing instruction operations.

80186 Instructions

In terms of operations which can be performed by or on all the registers mentioned so far (except CS and IP), there is a small number that can't. This is why the explanations above are qualified by words like 'usually', 'often' or conventionally'. There are few differences between the capabilities of any of the registers, with the exceptions that multiply, divide, input and output are restricted to register A only. It is the programmer's responsibility to use registers sensibly and to adhere to accepted good practice, because there are often several different ways to perform any given operation.

The contents of some registers, like CS and IP are significant to the processor and are excluded from some of the following operations, but within the bounds of common sense; there are almost no limits. If you wanted to add the contents of the data segment register to the contents of the base pointer you could do so, but whether such an operation is an appropriate technique is a separate question. Such is the nature of a 16-bit complex instruction set computer that there is a convenient machine code instruction for just about every eventuality.

When using the general-purpose registers or memory locations, there are effectively two versions of a great many instructions, allowing the same operations both at byte and at word level - such is the flexibility of 80186-assembly code. Here are just a few illustrations as there is insufficient space to explain the vast array of possible combinations of instruction formats. Each of the operations described below can be performed by a single machine code instruction.

You can add a literal value, a value held in memory, indexed or not, or a register's contents to another register. You can move a register's contents to or from another register or to or from memory, again indexed or not. It is even possible to add a register value or a literal value directly into memory! It is not, however, possible to add one area of memory directly to another area of memory without involving a processor register.

There are numerous compare instructions, some of which compare bytes, some words. Of these, some only set flags, while others will compare, set flags and also automatically advance the index registers in the currently set direction at the same time.

There is even an automatic loop instruction, giving a direct equivalent to BASIC's 'FOR...NEXT' construct. All the programmer needs to do is to set the loop counter, (CX) once, execute the code which is to be repeated, and then issue the instruction loop, with a label. Depending on the version of the instruction employed, various additional tests can be carried out by the Loop instruction making it conditional too. Implicit in all versions is the testing and decrementing of the loop counter. Regardless of the form of the instruction used, the loop ceases when the count reaches zero.

The 6502 operations of decrementing a counter, possibly performing an extra status test as well as a zero test and branching are all performed in 80186 assembler code by a single instruction.

To round off this brief insight into the 80186 instruction set's capabilities and as final food for thought for 6502 programmers, here are a few more facts about 80186 machine code. There is a REP (repeat) instruction, which repeats the following instruction a specified number of times or until a specified condition is met. There are 25 forms of the MOV (move) instruction, several of which can simultaneously advance the index registers. There are 31 conditional and 5 unconditional JUMP instructions and 5 different versions of the CALL instruction (the equivalent of the 6502's single JSR) and of course numerous versions of multiply and divide instructions.

Memory Addressing - Segmentation

The range of addressable memory in microcomputers is limited by the size of the registers concerned with memory addressing. In 86 series machines specific terms are used to refer to different sized blocks of memory. To fully appreciate the reasons a small amount of history is called for, together with a little simple arithmetic.

As usual the smallest base unit is a binary digit, or a bit. Four bits are known as a nibble, which is the smallest memory unit that can hold sixteen numeric values, or the complete range of unitary hexadecimal values (0h to Fh inclusive). Two nibbles form a byte, which is the smallest directly addressable single unit of memory. It is also the amount of memory needed to represent a single character.

In memory addressing a nibble can address 2^4 locations or 16 bytes. In both CP/M and 86 terminology a block of 16 bytes is known as a paragraph. This addressing convention was developed for the original 8 bit CP/M machines as a useful abbreviation, since any single register could hold a numeric value representing 16 times as much memory in paragraphs as it would be in bytes. In CP/M system calls, various operations required that quantities of memory be expressed in units of paragraphs. This is still the case in the 5l2 and all DOS Plus Systems when using many CP/M system calls (as opposed to DOS

interrupts.)

Conventionally, one byte can address one nibble^2 bytes, which is 256 or FFh locations and, as in the 6502, this amount of memory is called a page. Two bytes can address 2^{16}, 256^2 or &FFFF locations, which is 65336 bytes, usually shortened to 64k. If an extra nibble (remember 8-bit CP/M) were combined with this addressing range, making it into paragraphs, any two byte register plus a nibble could address 16 times 64k, a million bytes (1 Mb). This is precisely what was done for the larger memory 16 bit processors first used for CP /M86 systems which succeeded 8 bit CP/M.

This concept, which was also directly inherited by DOS systems, forms the basis of all memory addressing in 86 based Systems. Because all the processor registers are two bytes wide, the largest single unit of memory which can be directly addressed by one register, but which can still be accessed byte by byte, is 64k. This is known as a segment.

All memory addressing is based on units of a segment, but the explicit use of the 'extra' nibble is optional (i.e. it can be defaulted). When specified, it precedes the sixteen-bit portion of the address. The standard adopted for writing the two parts of the address is to separate them by a colon (:). The segment is written before the colon; the byte address within the segment, or the segment offset (usually just called the offset) follows the colon. As all the processor's registers are two bytes wide, each part of the address is specified as four hexadecimal digits, the lowest byte of memory is therefore 0000:0000.

This addressing scheme has very significant implications for the convenience of the way programs can be written and memory locations can be expressed and manipulated. If anything less than a segment is addressed, only a single register is needed - if no explicit segment address is given, the value in DS is assumed (except in the case of BP, as noted). By only the simple addition of a nibble qualifier, up to a megabyte can be accessed in 64k blocks. Since the two parts of the address are conceptually separate, either can be varied without disturbing the other.

Bearing in mind that the contents of the general purpose registers can be altered as two separate bytes or as one two byte value, it is easy to see that, by altering each part of a register containing a memory address, RAM locations can he accessed a byte at a time, a paragraph at a time, a page at a time or a segment at a time.

Addressing Techniques

When the processor translates addresses to an internal value for setting up the data bus to access a particular location, the two parts of the address, the segment and the offset, are combined to give a single 20 bit value. Remembering that the segment value is realty a nibble, representing 16 bytes, it can be seen that a segment value of &0001 is 16 bytes, &0010 is a page, or 256 bytes, &0100 is 4k, or 4096 bytes and a segment value of &1000 is a complete segment of 64k.

Obviously segment address values and segment offset values can overlap, depending on how they are

expressed. This probably causes mere confusion to the new 86 programmer than almost any other aspect of the system. Examine the following segment and offset addresses (all are in hex and perfectly valid) Each pair gives the same address:

```
seg:offset seg:offset
0001:0000 0000:0010
000F:0000 0000:00F0
0010:0000 0000:0100
0F00:0000 0000:F000
```

It can be seen that the segment address simply represents a value sixteen times bigger than the offset. When working out a memory address from a segment:offset address statement, all that is required is to multiply the stated segment value by sixteen and add it to the offset to arrive at a complete absolute address.

In many tutorials it is pointed out that, as the segment address can point to any multiple of 16 bytes, any address can be expressed in 4K, or 4096 different ways (64k/16). For example:

```
0120:0000
011F:0010
011E:0020
```

all point to the same address. Such addresses though, even with simple examples like these, can be highly confusing and care should be exercised. Try adding these two addresses together if you are not convinced!

```
0B7E:CA0E
03EC:13FA
```

Ultimately it's a matter of personal choice, but when planning and designing large applications, perhaps using large tables, it is sensible to treat segment offsets as ranging from &0000 to &FFFF, and count segments in ones of Mk each¸ that is:

Segment zero	is	`0000:0000 to 0000:FFFF`
Segment one	is	`0000:0000 to 1000:FFFF`
Segment two	is	`2000:0000 to 2000:FFFF`
Segment ten	is	`A000:0000 to A000:FFFF`
Segment fifteen	is	`F000:0000 to F000:FFFF`

and so on. You will notice that the last value given has a value of one less than a megabyte, the largest address possible in the 512. In real PCs, which do not directly address more than 640k, (ignoring

extended memory systems, which use special mapping, much like sideways RAM does in the BBC Micro) the highest address possible is 9000:FFFF. The highest address in an unexpanded 512 is of course 8000:FFFF.

It leads to much more understandable addressing at the planning stage if you completely ignore the bottom three nibbles of the segment address. Always treat them as if they will be zero unless there are very special reasons for wanting to count in 4k page or paragraph sized chunks of memory in certain operations.

In practice, all DOS programs are relocatable, so in a live program DOS sets up the segment registers to point to the first available paragraph of the actual code segment. In this way, all offsets within the code segment of a live program will start from zero, unless you decide otherwise. No matter where the program loads (governed by what else is already running) all offset calculations therefore remain valid.

Going a step further, if you could work within a single segment addressing would be even easier, as you could, in effect, forget the segment address. This explains why many applications have a 64K limit on the size of memory-resident files. It is also the reason why .COM programs may not exceed 64k in size, and why some applications or languages include both a 'small' (64k maximum) and a large' system version.

Because of the 16 bit 80186 registers, it is convenient to work within whole segments if at all possible. On initial program load, all the segment registers are already set up to point to the code segment. If your program can confine all its code and its data within the code segment, you can ignore the segment addressing nibble altogether throughout the entire program.

The offset becomes the only relevant portion of addresses so far as memory accesses are concerned, and the resulting program will be both simpler and more efficient. In cases where 64k is not sufficient for all code and data, other techniques can be employed to minimise segment address calculations and this is why several segment registers are provided.

For example, while CS points to the executing instructions, DS can be set to point to the working data segment. ES can then be used to point to a segment containing various subroutines, each of these being accessed by a jump address contained in a known offset from the start of the extra segment. If such jump offsets are held in a lookup table at the Stan of the extra segment, assembler functions the equivalent of BBC BASIC's 'ON x GOSUB' are simplicity itself. Since there are four segment registers, it is perfectly feasible to write quite large systems, without the need to frequently alter segment registers.

The RAM Memory Size

The Master 512 is provided with 512k of RAM as standard, or eight segments As can be seen from the way segment addresses are calculated, there would seem to be no problem in addressing a megabyte. In terms of the processor this is quite true, but other factors such as the operating system or the hardware

addressing chips may impose limits. In PCs this is precisely the situation. For reasons best known to themselves, the designers have, in most cases, imposed a limit of 640K on the directly addressable memory.

Expansion

In the 512 there is no such arbitrary hardware memory limit. The Solidisk PC Plus was supplied with a modified version of DOS Plus which permitted a full megabyte to be fitted. Since some packages require more memory than an unexpanded 512 can provide, a memory expansion project has been included later in the book and one is also available from Essential Software.

You should be aware that, while no 512 expansion board will give access to a full megabyte, it can provide more directly usable memory than in any PC- Because of this fact, there are few packages that will take advantage of extra memory over 640k. The main benefits of a larger memory are that a reasonable sized ramdisc or other memory resident utilities can be run, while still leaving the same net amount of free RAM as in MS-DOS systems.

3: Firmware

Tube Initialisation

This chapter concerns the actions performed during system initialisation. It is mainly concerned with booting the 512 system and loading DOS Plus, but parts of the operation are common to all co- processors and the initial stages are also performed when the host alone is activated.

It should be noted that the description of the steps performed in loading DOS may not apply precisely to all systems, particularly when exceptions to a normal load are encountered. Actions may depend on a range of variable factors, which together give quite a large number of possible combinations of hardware and software configuration. What is 'normal' for your system may differ in detail in the early stages of the load process, depending on which version of DOS is being loaded, whether the host is a model B, a B+ or a Master 128, whether the load takes place from floppy or hard disc and whether the ADFS in use is Acorn, STL or Watford Electronics. The following, therefore is a description of what theoretically happens.

On the pressing of the CTRL-BREAK keys together, the host performs a complete machine reset under hardware control, more commonly referred to as a hard break. There are some differences between a hard break and a power-up reset (i.e. when the micro is first switched on) but these differences are of no concern to, and have no effect on, the processes described here.

The hard break resets all vectors and the 6502's stack pointer, reinstates the default filing system, initialises service ROMs, checks the Tube and, if it is off, selects the default host language. This is followed by the return of either the familiar BASIC prompt or another language title screen if the default language is other than BASIC.

However, when the Tube is active, this process is interrupted. This takes place prior to language initialisation but after all other defaults have been set up or reset. On detection of a 'live' Tube, instead of initialising a language, the MOS downloads two sections of code, generally known as the Tube Host Code, from the filing system ROM into page zero and pages four to six. These areas are normally reserved for the current language ROM, but in this case, so far as the host processor is concerned, the Tube is effectively regarded as the 'current language'.

The BRK vector is redirected to point into the Tube host code in page zero at byte &16, both to handle unexpected error which might occur in any MOS or host software routine as well as to provide an automatic re-entry point to this code, while the program in pages four to six is entered to inform the co-processor of the hard reset.

The code stored in pages four to six is concerned with all standard inter-processor communication across the Tube and all the essential elements of co-processor to host MOS and filing system dialogue is provided by it. In the case of a 6502 co-processor and others, initialisation may subsequently follow a different path. For example, for a 6502 co-processor, if the host's current language ROM has a Second Processor relocation address, as does BASIC, a copy of it is written across the Tube and the language's prompt is displayed virtually immediately, with no necessity for further operations.

The Bootstrap Loader

In the case of the 512, when it is first notified of the hard reset it performs its own equivalent of the host's hard reset, again initially under hardware control. The 80186 executes a hard wired (permanently coded) jump into a tight program loop, which it continues to execute until a signal from the host causes a jump into the main bootstrap loader routine. When such a signal is received the 80186 executes a jump into the bootstrap loader, which first adds the familiar `Acorn TUBE 80186 512K` display to the host's screen display.

All the 512's hard-wired default code is located in the two EPROMs (as is the 80186 monitor, described later) which are permanently fitted to the 512 board. They contain the initial processor idle loop and sufficient code for the routines required to service the initial elementary inter-processor dialogue across the Tube for system start-up. On receipt of the Tube initialisation signal, the 80186 jumps out of its idle loop and enters the bootstrap loader.

Depending on whether the host's signal indicates a hard or a soft break, control passes to one of two routines. On a soft break the MOS remains in control displaying a * prompt and waiting for keyboard input while, in effect the 80186 awaits further developments within the 80186 monitor. At this stage users can demonstrate for themselves that the default 512 ROM code locates automatically in the 512's RAM by executing a soft break with the Tube enabled and entering:

```
GO FFFF:0000
```

As can be seen, this instruction causes execution to commence at the first byte in zero page in the 6502's memory, which at this stage causes the 80186 to jump to the start of the bootstrap code again and re display the Tube message. After this the system will re-enter the wait loop and do nothing further. In the case of a hard break (or the command `DOS` or `DOSBOOT`, after at least a soft break), however, the 80186 enters the loader routine proper.

Bearing in mind that, at this time, the only filing system control available to the system is that provided by the host in native mode, the reason is clearly seen why DOS must be booted from an ADFS format disc, and equally why the 800K DOS format cannot be used for this initial operation.

The bootstrap loader first sends a request to the host to close any open files, then it checks the current filing system type by issuing an OSARGS call. If it is not currently ADFS a filing system change is forced and confirmation is requested. Some versions of DOS on some machines stop here with an error if ADFS is not already selected, while others will perform the filing system change automatically as described.

On receipt of the confirmation of the correct filing system the bootstrap loader next checks for the presence of a Winchester. If a Winchester is available an attempt will be made to load from it by a request for the filing system to load the boot file from the ADFS file `DOSBOOT`. If `DOSBOOT` cannot be found on the Winchester, the load will usually automatically revert to floppy disc, but again this may not occur in some configurations.

It is helpful to have an appreciation of the full specification of file attributes to understand the next step. In the host's filing system the full specification of a file's load address is expressed in four bytes, though when the Tube is not active the first two can be, and usually are, omitted for convenience. When the Tube is active, however,

the full address must be specified, as all four bytes become relevant. This is because the coprocessor's RAM is memory mapped from `&00000000` to `&FFFEFFFF`, while `&FFFF0000` to `&FFFFFFF` is reserved for the host.

This complete addressing range allows 2^{32} bytes, which is equivalent to 4,294,967,296 bytes or 4 Gigabytes, of which the host therefore occupies only the top 64k.

As can be seen by a `* INFO DOSBOOT` in native mode, the file has a load address of `&04000000`. Since the load address of `DOSBOOT` clearly locates it in the co-processor, the file is read and sent across the Tube via the Tube host code located in pages four to six. The bootstrap loader receives the data on the 512 side of the Tube and locates it in the 512's memory, starting at the address indicated and incrementing the addresses as it does so. On completion of the `DOSBOOT` load the bootstrap loader's job is complete and control can be passed to the primary loader, which is now resident in the 512's RAM.

The primary loader is initially loaded into low memory at address 0400:0000, which eventually will be required by DOS Plus (which is not yet loaded) and so the primary loader first relocates itself further up memory at `3800:0000`. At this stage elementary DOS Pins filing system operations become possible and so the DOS file allocation table (FAT) and catalogue are read, and the (by now DOS, not ADFS) file `6502.SYS` is loaded into the 512 from disc, immediately followed by the Digital Research logo screen, if the file `LOGO.SYS` (optional) is present on the disc. (Hackers may wish to replace the `LOGO.SYS` file with an alternative graphic!)

Having been loaded into the 512, `6502.SYS` is at once downloaded back into the host's main RAM and located at &2800. This is followed by immediate initialisation of the new code in the host. The initialisation of `6502.SYS` re-directs interrupt request one vector (`IRQ1V`), the event vector and the user-vector to point into the code. Next, shadow is turned off by an OSBYTE 114, the Screen is cleared by the immediately following mode change to BBC screen mode 3. The `6502.SYS` code is now fully installed and active in the host. Next, the 512 sends the loader message and the Digital Research logo through `6502.SYS` to be displayed on the new screen, while CP/M and the DOS Plus envelope are loaded from disc, in spite of the fact that this screen calls itself the bootstrap loader. Ibis takes some time, so a dot is output to the screen for every 32 clusters (64k) of program loaded, to indicate to the user that the process is continuing satisfactorily.

The system is now almost live, so immediately prior to passing control to the user, or an `AUTOEXEC.BAT` file if present, DOS Plus must be downloaded to its permanent location and all OS variables and tables initialised. At the same time the `6502.SYS` code is again used to initialise the IBM screen display by directly programming the host's 6845 CRTC. Finally the DOS console command processor, `COMMAND.COM` is loaded from disc and a second copy is written to high memory, `AUTOEXEC.BAT` is executed if present, and control is passed to the user, giving the familiar DOS prompt, or to an application loaded by the batch file. The system is now ready for use.

The 80186 Monitor

As explained in the preceding section, after enabling the Tube by means of switching on (for external co-processors) or the appropriate CONFIGURE commands for a Master 128 using the internal Tube, the system will attempt to boot DOS Plus if a hard break has been performed. However, if a soft break is issued instead the

80186 monitor is entered (although on a Master 128 you may need to press ESCAPE).

In practice, the event which is most likely to cause you to do this is when an unexpected hang up or crash has occurred in DOS and you risk losing an appreciable quantity of work if you immediately re-boot the system. In DOS this type of occurrence tends to be more serious than in native BBC mode, primarily because DOS applications generally use much larger files and more dab can be lost. The very best insurance against this, of course, is to save working files frequently and back-up at least one, and better two or three, master copies after every work session. Remember also that the following techniques offer no salvation after a power cut. It is to be hoped, therefore, that most of the following information will prove interesting rather than necessary!

It is vital that you remember that, even when the 512 board remains powered up during a DOS re-boot, very large quantities of the 512's memory are overwritten several times and large portions are completely wiped when DOS initialises. If you must attempt to recover data directly from the 512's memory after a hang up or a crash, no other option than a soft break and the monitor exists.

By means of two of the commands, search and dump, areas of the 512's RAM can be searched for key data, the information can be checked visually, and finally by means of a `*SAVE` filename command (remembering to supply the full 4 byte address correctly) the located data can be secured to a (native format) disc. After this the normal DOS utility, `MOVE`, can easily transfer the data back to DOS Plus format for final file reconstruction.

Under no circumstances press CTRL-BREAK if data recovery is to be attempted. No matter how quickly you may realise that you have done it, you might be too late and part of the 512's RAM contents can be lost. Pressing only the BREAK key deposits you in the 80186 monitor, when several facilities will allow you to find and recover your data. Entering the monitor itself has no implications for the recovery of data from the 512's RAM. The monitor is contained in the two EPROMs mounted on the 512 board and is automatically mapped in low memory, in part of the RAM previously occupied by DOS, by a soft break, therefore no user data is overwritten when it is entered.

On pressing BREAK the top of the screen will include the normal display Seen when you are about to boot DOS, but nothing else will happen. Instead of a language or a DOS prompt you will find the command line prompt is a star (*)

The star prompt indicates that the 80186 monitor is waiting for manual user input. Commands entered at this point will be read by the MOS and are first passed to the monitor, which examines them. if they are not recognised by the monitor, control will be passed back to the host's MOS and the command is processed in the host in the usual way.

It will be found that not only monitor commands, but any command which will be understood by the host's MOS or its resident software can be issued (including filing system commands and service ROM calls) provided that such software is legally written. Note that ROMs employing dubious coding techniques are very likely to hang the system. However, even if this occurs there is no problem, as any number of repeated soft breaks have no effect on the contents of the 512's memory and should always return you to the monitor prompt.

As can be seen from the sequence of operations when the system is booted, you cannot use the monitor from within DOS by means of STAR command. Although automatically memory mapped in low 512 RAM on

pressing BREAK, the boot ROM code is only entered prior to system load. In other words the monitor is no longer memory resident after DOS has booted, since DOS itself uses this area of the 512's RAM. Entering STAR H. from the DOS prompt will clearly demonstrate that the monitor is not available in a live DOS system.

Commands that are not recognised by the monitor or host software will produce the host's normal error response of `Bad command, Bad Filename` etc. An attempt to call a 6502 relocatable language (such as BASIC) will produce the message not 80186 code. In the case of languages that can't relocate (even in a 6502 Second Processor) a suitable response should be provided by the language itself. The message given by such languages should be `Turn second processor off, Turn Tube off`, or something similar and equally to the point.

Monitor Commands

In addition to the standard MOS filing system and host resident software commands, the 80186 monitor provides a range of 512 specific commands. All of these relate solely to the 512's memory except one, which involves transfers between the memories of both machines. The monitor itself responds to the name MON (note that no full stop is required) and `*HELP MON` will produce the following help display, returning to the '*' prompt immediately afterwards:

```
*HELP MON

80186 TUBE 1.00

a (<segment>: (<start offset>) (<end offset>
dos
f (<segment>. <start offset> <end offset> <filll byte|word>
go (<segment>:) <offset>
mon
S (<segment>: <offset>
SR (<segment):) <start offset> <end offset> <string>
tfer <io addr.> (<segment>:) <offset> <length> <r>|<w>
```

The command prefix, entered immediately following the '*' prompt, identifies the function to be carried out using the parameters which follow. Note that the monitor command prefix is not treated as an abbreviation and in usual DOS manner no point (.) should be added to these either.

Items in angled brackets, e.g. `<segment>`, indicate required parameters, though in some cases these may be defaulted as explained below.

Items in round brackets e.g. (`<segment>`) indicate that the enclosed item is optional and may be omitted from the command. If, as in the case of (`<segment>`), although the enclosed item is a required parameter you may elect not specify a value, therefore an appropriate default will be supplied by the monitor.

Items separated by a vertical bar, e.g. `<r>|<w>`, are mutually exclusive alternative entries and one of the two

must be supplied. Monitor commands conform to normal DOS syntax rules and commands are not case sensitive. Generally either case, or a mixture of the two can be used, with the single exception of the `<"string">` argument in the 'SR' (search) command.

Where a segment address must be supplied, the monitor will permit leading zeros to be omitted and any address (stated in paragraphs) may be specified, but wherever a segment address is supplied it must be immediately succeeded by a colon, whether an offset follows or not. If no segment address is supplied the colon is also omitted.

For all commands, if the segment address is omitted the most recently specified segment is used as the default. If no segment has been specified in a previous command the value will default to zero. Extra leading asterisks and unnecessary leading or trailing spaces are ignored.

Where an offset is required it may similarly be entered without leading zeros. If more than four hex digits are entered the most significant digits are ignored.

While the above help display may remind you of the syntax required, it fails to explain exactly what each command does. The following list completes the information.

Name	**Function**
`P`	Dump 512 memory, displayed in hex and ASCII
`DOS`	Boot 005+ from hard disc or floppy
`F`	Fill 512 memory with a byte or word
`GO`	Go to (i.e. execute code from) a specified address
`MON`	Re-enter the monitor
`S`	Show (edit) 512 memory in hex and ASCII
`SR`	Search 512 memory for a specified character string
`TFER`	Transfer memory contents between 512 and host

Each of these commands is now examined in detail, with examples of use together with any additional comments that relate to a partition command.

Each of these commands is now examined in detail, with examples of use together with any additional comments which relate to a partition command.

D

Dump specified memory to screen

Syntax
`D (<segment:>) (<start offset>) (<end offset>)`

Function
This command produces a memory dump from the 512's RAM to the Screen between the specified addresses, with a display in hex and ASCII showing 16 bytes on each line. The start address of each line is shown in the normal segment:offset fornat. Characters outside the normal printable range of 20h to 7Eh (CHR$32 to CHR $127) am shown as a full stop in the ASCII portion of the display.

All of the parameters in this command are optional and may be defaulted. If the segment address is omitted the last used segment value (or zero if none) is used. If the start offset is omitted, the display begins from the address of the last byte displayed plus one, or zero if there is no previous address. If the end address is omitted, the start address plus 128 more bytes are displayed, therefore this requires nine lines of display- As displays are output in multiples of 16 bytes, in effect the default display length becomes 144 bytes. It will be appreciated that it is a practical as well as a logical impossibility to an arid offset without also supplying a start offset.

Note that a host *spool <filename> command can be issued prior to using this command (or in fact any MON command) to easily create a record of the displays, provided that a suitable disc is placed in the default drive (and *MOUNTed for ADFS). In fact, this technique was employed to capture the examples shown here. The operation may be carried out in any (practical) host filing system, most usually DES or ADFS. When the required data has been captured, * SPOOL should be used to close the file and correctly write its attributes.

Five examples of the D command are shown, since the method of supplying 512 addresses and the defaults which can apply for other commands are best illustrated by this one. Each of the following example commands were issued exactly in the sequence shown, so that the effects of the defaulted values can be clearly seen and followed from one display to the next.

This is the first MON command, issued immediately after a soft break only the segment address was specified. Note that the colon is required to indicate that the single parameter is a segment address Without the colon the value would be taken to be an offset. Since no previous monitor command has been issued the offset in this case defaults to zero and the end address defaults to the start address plus 80h bytes, that is the first byte of the last display line:

```
*D 1234:
1234:0000 BF 01 60 00 4F 3E 80 01 60 00 CC EE 80 00 97 03
1234:0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1234:0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1234:0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1234;0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1234:0050 00 00 00 00 86 00 8F 11 6A 00 99 11 E1 0E E1 0E
1234:0060 30 02 E2 06 D9 0F 00 00 85 0F 00 F2 A1 04 60 00
1234:0070 00 00 51 12 02 01 00 00 00 00 00 00 00 00 00 00
1234:0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The second command~the offset only is specifled. The segent has defaulted to the value used in the previous command. Note that the start address can be any value from 0000h to FFFFh, that is, it need not be an even or 'round' byte number. The absence of a trailing colon after the single address indicates that this is an offset address:

```
*D 8000
1234.8000 20 20 20 20 20 20 20 0D 0A 20 20 20 20 20 20 20
1234:8010 20 20 20 20 20 20 20 20 20 63 46 34 2E 55 6E 2D
1234:8020 64 65 6C 65 74 65 20 20 20 63 46 36 2E 54 6F 66
1234:8030 69 6C 65 2F 6D 61 72 6B 20 63 46 38 2E 52 69 67
1234:8040 68 74 2D 66 6C 75 73 68 20 63 46 31 30 2E 53 77
1234:8050 61 70 2D 66 69 6E 64 20 20 0D 0A 0D 0A 0D 0A 4D
1234:8060 45 4E 55 3A 20 20 68 65 6C 70 2C 20 65 78 69 74
1234:8070 2F 63 6F 6D 6D 61 6E 64 2C 20 73 61 76 65 2F 75
1234:8080 6E 73 61 76 65 3B 20 20 6E 61 6D 65 20 74 65 78
```

The third command - all three parameters, the segment, start offset and end offset are specified so as to start and end the output partway through the previous display, though the result would have been the same if the segment had been omitted:

```
*D 1234:8020 8030
1234:8020 64 65 6C 65 74 65 20 20 20 63 46 36 2E 54 6F 66
1234:8030 69 6C 65 2F 6D 61 72 6B 20 63 46 38 2E 52 69 67
```

The colon must be supplied after the segment address, but no space between the colon and the start offset is required (as is implied in the help display), though entering one does not cause an error- It should be noted that although the end offset is specified as 8030, the display does not necessarily terminate at the specified end byte. The complete display line of 16 bytes which includes the supplied end address is always output by this command. This is the reason that the default display length of start-byte + 128 bytes requires nine lines of display.

The fourth command~start and end offsets only were specified:

```
*D 8040 8060
1234:8040 68 74 2D 66 6C 75 73 68 20 63 46 31 30 2E 53 77
1234:8050 61 70 2D 66 69 6E 64 20 20 0D 0A 0D 0A 0D 0A 4D
1234:8060 45 4E 55 3A 20 20 68 65 6C 70 2C 20 65 78 69 74
```

Again note that the complete 16 byte line which contains the spedifed end offset is displayed.

Fifth command-all parameters defaulted:

```
*D
1234:8070 2F 63 6F 6D 6D 61 6E 64 2C 20 73 61 76 65 2F 75
1234:8080 6E 73 61 76 65 3B 20 20 6E 61 6D 65 20 74 65 78
```

```
1234:8090 74 2C 20 66 69 6C 65 20 6C 6F 61 64 2C 20 70 72
1234:80A0 69 6E 74 3B 20 20 64 69 72 65 63 74 6F 72 79 0D
1234:80B0 0A 52 55 4C 55 52 3A 20 20 73 70 6C 69 74 20 73
1234:80C0 63 72 65 65 6E 2C 20 65 64 69 74 20 6D 61 72 67
1234:80D0 69 6E 73 2C 20 6D 65 6E 75 3A 20 69 6E 73 65 72
1234:80E0 74 2C 20 64 65 66 61 75 6C 74 2C 20 74 6F 2F 66
1234:80F0 72 6F 6D 20 66 69 6C 65 2C 20 65 74 63 2E 0D 0A
```

In this case the segment and and start offset follow from the immediately preceding command and the default of nine lines of display is again applied. Note that if the start offset plus the display range should exceed the end address of the segment (FFFFh), wrap around occurs within the given segment. That is, the byte displayed after, for example, 1234:FFFFh is 1234:0000h.

DOS

Reboot DOS+

Syntax
`DOS`

Function
This command is intended to allow DOS to be booted without a CTRL BREAK, ie, from the monitorts star prompt after a soft break after leaving DOS, or powering up the coprocessor from native EEC mode followed by a soft break. The command will try to boot DOS, but,
depending on what you have been doing immediately previously, in some Systems it does not always function correctly. If this is the case the correct effect can sometmes be achieved by entering DOSBOOT from the MON star prompt, though it is equally simple and more reliable to perform a hard break, which will also produce the desired result.

F

Fill memory with a constant

Syntax
`F (<segment:>) <start offset> <end offset> <11 byte | word>`

Function
This command fills the 512's memory with the supplied fill value, begins at the the specified start offset and up to, but not including, the end offset. Only the segment may be defaulted, both offsets and the fill value must be supplied, there are no defaults. The nil value supplied may be specified either as a byte or as a word (two bytes). Fill bytes rnnst be given as hexadecimal values, strings are not permitted.

```
*F 1234:1000 2000 20
```

will fill memory locations l000h to IFFFh' inclusive, in segment 1234, with the hexadecimal byte value 20h (space).

```
*F 1000 1100 ABCD
```

In this case the segment is defaulted, the previous segment value (or zero if no previous value) being used as the default. The command will fill bytes I000h to l0FFh inclusive with the hexadecimal word value 'ABCD'. As in all internal word formats, the low byte precedes the high byte. i.e. byte 1000 will be CD, byte 1001 will be AB and so on up to byte l0FE which will be CD and byte l0FF which will be AB.

The end offset may be specified as 0 so as to fill from the start offset up to and including the last byte in the segment For example, to fill the entire segment l000h with null bytes (0h) the command would be:

```
*F 1000:0 0 0
```

GO

Execute code starting at the specified address

Syntax
`GO (<segment:>) <offset>`

Function
This command directs the 80186 to transfer control to the code at the specified address in the 512's RAM. Its original purpose was for use when debugging the operating system. As such it is of little practical use to the user, for whom far better debuggers are available and who, in any case, would normally want to execute applications code in a live DOS system.

MON

Enter the 80186 monitor

Syntax
`MON`

Function
This command is also provided for debugging the operating system, when memory addresses can be suitably manipulated to allow the monitor to be entered on demand from within various sections of code. As for 'Go', for end users the command serves virtually no purpose, since it cannot be called either from DOS or from the BBC Micro in native mode, but only from within the monitor itself or from host or 80186 machine code programs loaded and run ftom the monitor prompt.

S

Show memory contents (for editing)

Syntax
`S (<segment:>) <start offset>`

Function
This command displays the specified contents of the 512's RAM for examination and possible alteration. A single display line of 16 bytes of memory is presented in the same hex and ASCII formats as were seen in the dump command. The cursor is positioned lrdtialty under the least significant digit of the first byte specified. Cursor movement and the format of data entry is controlled by the following keys:

CURSOR LEFT	Move left, if at far left display previous 16 bytes
CURSOR Right	Move right, if at far right next 16 bytes
CURSOR UP	Display next 16 bytes
CURSOR DOWN	Display previous 16 bytes
SHIFT + LEFT ARROW	Move cursor to far left of current line
SHIFT + RIGHT ARROW	Move cursor to far right of current line
COPY	Toggle between hex and ASCII entry

The display, like the dump format, shows two 16 byte fields, with the hexadecimal representation on the left of the line and ASCII codes to the right. The COPY key will cause the cursor to jump between the two displays to allow data to be entered in either form.

For example:

```
*S 0040
```

followed by three cursor down key presses and two cursor up presses will give the following six display lines in sequence. Note the offsets.

```
0000:0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0030 D6 1F 00 F0 00 00 00 00 00 00 00 00 00 00 00 00
0000:0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0030 D6 1F 00 F0 00 00 00 00 00 00 00 00 00 00 00 00
```

During the display of each line, any of the 16 bytes shown can be amended by moving to it and entering a new value. When the cursor is in the hex field, data is always entered as two digits, each digit being shifted in from the right as it is entered, ie, to enter 2F into a previously null byte, you would move the cursor to the required hex byte position and first enter the 2, giving 02h then the F, causing the 2 to move left by one digit to give 2Fh. If you continue to enter more digits at the same byte position the cursor does not automatically advance. For example, using the current entry, if you next entered a 6 the 2 would be lost and the resulting value would be

F6h.

When the cursor is in the ASCII field, data is entered as ASCII bytes je as normal upper or lower case characters from the keyboard, or as control characters (CTRL+char). After the entry of every byte in ASCII format the cursor is automatically moved forwards to the next byte position. When a character is entered into the last byte on the line, the cursor is advanced to the first byte of the next 16 byte display line.
On completion of editing, ESCAPE returns control to the monitor's star prompt.

SR

Search memory for a text string

Syntax
`SR (<segment:>) <start offset> <end offset> <"string">`

Function
This command searches memory between the specified address limits attempting to match the supplied string, the search argument, which must be enclosed by double quotes. Each occurrence of the string found between the specified limits causes a displayed report of the start address for each match. The addresses are shown in the usual 'segment:offset' format. The search string may be up to a maximum of 72 characters in length and it should be noted that, unlike most entries in DOS, the search argument is case sensitive.

This command is most useful when readable data which can be easily or uniquely identified has to be located in the 512's RAM. Examples are, a key phrase or section title in a wordprocessor document, the first column header in a spreadsheet or the name in a database record and so on. Note that the search is always confined to the specified (or defaulted) segment.

After locating the data string the durnp command should be used to display the entire document or file, if possible. This is a wise precaution, used to verify that the correct part of the RAM has been located. Remember, without this check you may be just recovering a small part of your application's or DOS's workspace, perhaps just a couple of thousand or hundred bytes of data in a work buffer, or the holding area in a wordprocessor, rather than say, a complete 150k file.

To simplify the illustration of search, the same section of mernory as was displayed in the dump command has been used, in an attempt to match a space followed by a lower case 'c', and an uppercase 'F'.

The first example searches for the specified string, " cF", in segment 1234h between offsets 7500h and 8500h:

```
*SR 1234:7500 8500 " cF"
1234:8018
1234:8028
1234:8038
1234:8048
1234:84CD
1234:84E1
```

By visually comparing the search results to the output from the dump command it can be seen that the address given for each rnatch is the address of the first byte of the matched string. To search for this string right to the end of the segment, the end address may be specified as a single zero, as in the example below which, in this case, produces only one extra match at 8FE0:

```
*SR 8000 0 "cF"
1234:8018
1234:8028
1234:8038
1234:8048
1234:84CD
1234:84E1
1234:8FE0
```

If the search is restricted to a smaller area of memory, as you would expect only matches contained entirely within the search limits are induded in the results. Therefore:

```
*SR 8000 304A CE"
1234:8018
1234:8028
1234:8038
```

does not include the occurrence of string " cF" located at offset 8048, since part of this occurrence of the string (the last byte in this case) lies outside the specified search limits. It is perhaps easiest to think of the start and end addresses as the first byte to be searched and the first byte not to be included respectively. This is why an end address of zero permits a search to the end of the segment (offset zero minus one = FFFFh).

As already mentioned, search argument matching is case sensitive, therefore the search:

```
*SR 1234:8000 0 " cF"
```

produces no matched strings, nor does a search for either " CF or " cf" in this case.

Any 8 bit character can be specified in a search string and this can indude non-printable characters, but of course not all such characters can be entered directly from the keyboard by a single keypress. The method adopted uses the standard Acorn BBC Micro conventions which are.

The I(vertical bar) character prefix is entered to denote a control sequence, ie I@ is the single ASCII character 0 and IM is the single ASCII character 13, or a carriage return. CTRL+character may not be used in this case, because, for example CTRL-G (IC) only causes the micro to beep, while return (IM or CHR$13) will terminate the command input giving the error bad string, since there would be no terminating quotation mark. The reason for this is simply that the command line input is being read initially by the host's MOS, character by character, and normal BBC Micro MOS rules apply.

For characters greater than ASCII 127 (80h) the 'pling' operator (! or ASCII 21h, sometirnes referred to as

'shriek') is added to indicate that the top bit should be set. This method can also be used with control sequences, when '!' follows the vertical bar preceding the character.

For example "IA" would specify a character value of ASCII 1 (01h) while "I!I@" would specify ASCII Character 128 (or 80h) and "I!I would specify CHR$141 or 8Dh.

Two special cases exist, CHR$127 and CHR$255. since both of these would require the entry of a DELETE character, which is naturally impossible since the delete key performs its normal function during command entry. The question mark (ASCII 3Fh, CHR$63) is used in conjunction with the 'I' and '!' to indicate these two, so DELETE (7Fh) is entered as "I?", while cHR$255 (FFh) is entered as"I!?".

Obviously search string entries can become quite confusing when a number of non-printable characters must be included. It should be noted that invalid character modifier combinations do not necessarily cause an error. If a control sequence is not recognised, the modifiers are stripped one by one, reducing the entry until a valid character results. Only if no valid Character results from this reduction is an error reported.
For example "I1" would be invalid as it is meaningless, therefore it would be reduced by the monitor character interpreter to "1". Excess 'I!' modifiers are likewise stripped and ignored so "I! I! I @" is reduced to "I!I@'. Great care must be taken, therefore, in a search performed to recover important data which includes control sequences if the results are to be relied upon.

TFER

Transfer between the 512's RAM and the host's.

Syntax
```
TFER <I/O address> (<segment:>) <offset> <length> <R / W>
```

Function
This facility performs fast transfers of blocks of memory between the 512 co-processor's memory and the host's memory, in either direction.

The I/O address parameter, which must always be supplied, refers only to the start address of the memory block in the host's RAM and therefore does not require a full four byte specification (see examples). Segment and offset addresses refer the the 512's memory. with segment being optionally defaulted in a similar manner to previous commands. The offset address must always be supplied. The length must be supplied and refers to the length of the block of data to be transferred. It is specified as a two hex byte value, though leading zeros may be omitted, i.e. 00FF is taken to mean the same as FF.

The final parameter represents the direction of the transfer, which must always be specified. The two possibilities are R (read) or w (write). Conceptually it may help to remember that the monitor resides in the 512, therefore the command is executed by code within the 512, not the host. From the monitor's point of view therefore, 'R' means read into the 512, while 'W' means write from it.

For example:

```
*TFER 2000 1234:0 4000 w
```

would transfer the first 4000h bytes from segment 1234h in the 512 to the BBC host's RAM, storing the data starting at address &2000. Note that, in this case, unless a suitable screen mode is selected (eg, mode 7), part of the screen will be overwritten by the transferred data. Master 128 users can configure shadow to avoid this problem, using the 'dodge' of ensuring that no boot disc is available when the hard break is issued, followed by an immediate soft break or ESCAPE to return to the monitor.

Users of Model B or B+ hosts should note that, because theft shadow RAM is implemented in software rather than hardware, host direct memory accesses (DMAs) are unaffected by shadow settings and this option is of no use to them. In other words, even with shadow RAM selected the data will still be written to the screen RAM.

Transferring in the other direction:

```
*TFER C000 4000:0000 4000 R
```

would copy the 16K image of the host's MOS ROM to the 512's memory starting at byte zero in segment 4000h.

The transfer is implemented using OSWORD 0FAh and uses fast Tube transfer types 6 and 7 where possible. If the transfer length is not an integer multiple of 256 (FFh) bytes, remaining bytes (ie length MOD 256) are transferred using the slower type 0 and 1 transfers. The total amount of data transferred is, of course, in practice limited by the host's available RAM. Tube protocols and operational speeds are discussed in the next chapter, but from the subjective point of view of the operator all transfers appear to be virtually instantaneous.

TFER may be used manually or by machine code routines in either processor which have been loaded and run from the monitor prompt. As is usual with toolkits and memory editors, there is no check on the memory addresses to be overwritten by transfers in either direction. It is therefore incumbent on the user to ensure that no vital areas of memory in either the 512 or the host are destroyed by the use of this command.

Host Errors

When an error or escape condition is deteded by the host processor, the error number and string are passed across the Tube to the 512, generating an interrupt request. This is serviced by the Tube code in the 512's monitor.

The error number and string are placed in the error buffer of the 512 and a pointer is initialised to point to the error number. The error string is terminated by a null byte (00h). The 512 Tube code then jumps to the error handler which displays the error before returning control to the 80186 monitor.

The error handling provided by the monitor is not suitable for use in stand alone applications. Stand alone programs consist of code which runs in the 512 but not under DOS Plus, that is, programs which only use MOS functions and host filing systems.

Monitor error handling is not usable in this way because control is returned to the monitor by the default error handler, not to any other code that may have originally called it. Error handling is further discussed in Chapter

Eight DOS Plus interrupts.

3: The Tube

While the 80186 processor may be the heart of the 512 board, the Tube is the most critical part of the complete system. Every byte of data read from disc or keyboard or output to disc, screen or printer must pass through the Tube at least twice.

It is equally true that, even for experienced users, the Tube is also the most mysterious part of the machine. Its operations cannot directly be inferred from the activities which take place at either end of it and, particularly in the case of the 512, it is difficult to investigate, since two very different types of machine code are involved at the same time. The remainder of this chapter examines the general functions of the Tube and the standard inter-processor communication code which provides the coprocessor with MOS and filing system support in the host.

General Description

The Tube is an Acorn designed and registered proprietary data interface for connecting two microcomputers, which may be of similar or different types. The data transmission rate of the Tube is nominally 2 million bits (250,000 bytes) per second. With due allowance made for the overhead of mutually understood communication signals and latency between the two processors, effective data rates are less, though still more than fast enough to be small penalty even for rapid peripherals such as disc drives and powerful processors such as the 80186.

The Tube terminates in each machine in a dedicated chip called a Tube ULA (Uncommitted Logic Array), so as to provide a data bus 32 bits wide, giving four independent bi-directional communication paths. Each path consists of a one-byte wide control register, also referred to as a status register, and a one byte wide data transfer register.

While the two machines connected by Tube may appear to work in synchronisation to provide the facilities of the system as a whole, the Tube is seen by both processors more or less as an input-output peripheral. Each can make a request, or a response to a request, and then continue to perform other tasks, for example, processing interrupts or events.

In Acorn systems to date, the host, which provides all physical I/O services, has always been a variant of the 6502 based BBC microcomputer. Tube communication, therefore, is based around the facilities provided by the host's MOS and filing system and the parasite's anticipated requirements for them. In all versions of host, the range of MOS functions provided for parasite support is common, and this, therefore, dictates the protocol for requesting services or accepting a response. This constant base has, so far, ensured that all versions of the host (when fitted with the appropriate hardware) can support all versions of co-processor.

Since the host's software interface and its facilities are effectively invariant, it is the responsibility of the parasite to provide such machine code routines in its own language as are necessary to conform to Tube communications standards and to use the facilities as provided. If additional or non-standard facilities are required by the parasite over and above those provided by the Tube host code, they must be supplied by the parasite, but by means of extra code located within the host. This is the case for the 512, since it uses both non-standard screen output and non-standard disc formats, neither of which are supported by the host's MOS or filing systems.

The two processors on each side of the Tube are not in any way directly synchronised with each other. The Tube is regarded by both as an asynchronous data channel, which in the host must be checked for activity requests from time to time and in the parasite must likewise be checked, or it may demand attention by means of interrupt requests, or non-maskable interrupts for very fast transfers. This situation can best be envisaged by two simple illustrations. From the 512's viewpoint, keyboard or serial data input may be waiting at any time in the host, while from the host's point of view, the 512 might request any input or output services at any time.

To maximise overall system performance the Tube is therefore given a high priority by both processors. So as to avoid keeping the other processor waiting, each might delay certain other operations in order to service the Tube. This is sometimes visible in the 512 system when large quantities of screen data are output rapidly. Although data is being transferred and written directly to the host's screen memory map, the host may not be permitted sufficient time to update the visible image concurrently, which therefore is seen by the user to appear in 'chunks', or sometimes even a screenful at a time.

In the host the highest priority activity is usually the disc filing system (DFS), or the fileserver in networked systems. One of these two is normally the owner of the (sole) Non-Maskable Interrupt (NMI) in the host and NMI routines are permitted to cut across all other activity when the 6502 processor's services are required. In co-processors that exclusively use standard host facilities this is acceptable, since all I/O is via the Tube host code, using only MOS and filing system services. In the 512 system, however, since discs are also read and written by the 512's own 6502-resident code, this is not so.

The extra host code provided by the 512 is loaded from 6502.SYS and is subsequently downloaded into the 6502 host's RAM to supplement the Tube host code facilities- Amongst the extra routines provided are facilities for directly programming several of the host's hardware device controllers. Therefore this code introduces an extra NMI owner into the system. During such 512 operations, the host's MOS and filing system are by-passed and NMI ownership is claimed from the host (or from the network) and may be retained for the duration of the operation, which under no circumstances can then be interrupted. Since no NMI routine, once started, can be interrupted before completion, the 6502.SYS code has, by claiming the NMI, effectively implemented a method of masking both maskable and non maskable interrupts from any external source.

In the host to parasite direction, such critical operations as the transferring of disc data must not be delayed, since this would, in effect, hold up a host NMI, which itself holds up other host interrupt and

event activity. For this reason many Tube transfer requests generated in the host cause an interrupt request in the parasite, which therefore services the transfer rapidly. In practice, apart from the case of a machine connected to a network, there is nothing on the host side of the Tube which can generate unexpected NMIs. Even host disc activity is normally a direct result of a parasite request, which therefore will often be 'waiting' for the response.

However, there is one autonomous activity in the parasite that could interfere with such time critical functions. That is, processor clock interrupts, (an 86 clock interrupt is generally called a 'tick' for short, for the obvious reason). To avoid this potential conflict, very fast Tube data transfers can be set up so as to generate an NMI m the parasite, thus suppressing dock interrupts for the duration.

Tube Operations

The prime purpose of the Tube interface is to permit a parasite processor to gain access to host MOS and filing system functions which are concerned with host configuration and peripheral device support. The host system software calls, that form the basis of this support, are essentially of three types.

1. Calls which cause the host to carry out some independent action for which there is no returned information. For example, writing characters to the current output stream (OSWRCH) or changing the serial port speed (OSBYTE).
2. Calls which cause the host to perform an activity which will result in a return of low volumes of information in a non time critical (i.e. asynchronous) manner. For example, reading characters from the current input stream, usually the keyboard or the RS232 port.
3. Calls which cause the host to perform a function which will result in information being returned in volume and/or in a time critical manner. For example reading or writing disc information or communication through a network.

The four Tube registers are broadly assigned to these call categories as follows. Calls of type one from the parasite to the host are passed through Tube register one. Type two, the general MOS and filing system commands, are passed through register two.

Calls of both types (if appropriate) from the host to the parasite pass through register two. In both these types of call the relevant processor polls the appropriate status register until the status flags indicate that the data register is ready to accept or supply data. In the host code source listing, this loop of code is called `Tube_idle_loop`, which is executed permanently when the system is not involved in servicing existing Tube requests or its own interrupts. In the 512, the counterpart to these Tube registers are checked on each processor clock interrupt, and serviced if a Tube interrupt request (IRQ) is outstanding.

From parasite to host, register one is used for writing to the current output stream via OSWRCH, while register two is used to request various calls such as OSBYTE, OSWORD and filing system calls. However, a filing system call depending on the service requested, may or may not result in a transfer (in either direction) of type three.

If it does and the transfer is to the host, the host will read register three until data is received and process it according to the operation requested. When data is written to (or read from) register three by the host, if necessary an NMI can be generated in the parasite, which must therefore immediately suspend other activities So as to service Tube register three without delaying the host.

If the transfer is from the host, register one or four is first used by the host to generate an IRQ in the parasite to warn of the impending arrival of data. The host then waits for a response from the parasite.

The parasite immediately sets status register three to signal its readiness. At this stage, the host assumes all the data may be sent and proceeds to transfer the data through register three at a regular rate with no further status checks. It is therefore the parasite's responsibility to ensure that data is read from register three at a rate fast enough to keep pace with the host before the next interrupt is generated. It will be seen from the host source listing that, because this type of Tube transfer is time critical, the code includes three NOPs to introduce delays into the transfer loop to increase reliability, especially of the slower co-processors such as 6502 and Z80 based versions.

Note that not all transfers through register three generate NMIs in the parasite, only host NMI or large block transfer routines are so time critical as to require NMIs in the parasite. In addition, many type two calls return low volume information through register two.

Tube registers do not normally cause direct interrupt requests in the host, although the peripheral activity requested by the parasite may do so by means of the host's normal hardware and software functions. However, the parasite has no peripheral hardware except the Tube ULA, so the reading or writing of Tube registers by the host serves to set parasite IRQs or generate NMIs while the reading or writing of the registers by the parasite clears such requests. It should also be noted that at any time it is possible for more than one type of Tube transfer to be outstanding by the use of different Tube registers.

Tube Protocol

Since at any point when there is no current activity neither processor can know what the next Tube request is likely to be, or what the other processor is doing, both must be prepared to recognise and carry out any of the range of Tube functions at any time. A communication system must exist therefore, where each processor can either leave a request for action, or acknowledge a request left by the other, which will be understood and acted on accordingly. Many of these actions are followed by the transmission or receipt of data, which require a similar understanding. This communication system, together with the Tube register usage, is known as the Tube protocol.

Tube protocol is a standard method of signalling intentions, requests or results through Tube registers, which will be understood by both processors, even when they are of different types. The eight registers are generally used in pairs, the status register being used to signal a request or condition, the data register for the actual transfer or response, though in some cases the request is made through one register

and subsequent data transfer uses another.

The first necessity for any facility wishing to use the Tube is to successfully claim temporary ownership of it. This prevents any other facility attempting a simultaneous transfer in the same or the opposite direction. In this context a 'facility' can be the operating system in either processor, the host filing system or a network controller, each of which may wish to use the Tube host code for their own separate purposes. The entry point at &406 in the Tube host code is used for the purposes of claiming or releasing the Tube, as well as for transfers of data between the memories of the two processors.

To claim the Tube a call to this code should be made with a value in the accumulator of &C0 plus an identifier code that is unique to the caller. The identifier is a value between &1 and &F inclusive and is stored in the host's page zero on a successful Tube claim to record who currently has control. The claim at first may appear to have failed if the Tube is already in use, but the caller maybe the existing owner, therefore the caller's identity is compared with the current owner's. If it matches, the Tube claim is deemed to have succeeded. In other words, there is no 'you already have it' type of reply. There are only two possible results from a Tube claim, success or failure.

If the Tube claim is successful, the carry flag is set. The caller may then proceed with the next stage in the operation. If the claim returns with the carry flag dear the Tube claim was unsuccessful, which means it has already been claimed by another facility. If in this case the failed claimant is an operating system, it should immediately poll the Tube status registers on its own side of the Tube to ascertain what action follows. Filing systems and the additional Tube code in 6502.SYS simply keep trying the claim until successful. Since they have no Tube registers to monitor, such requests as are intended for them are notified by the operating system.

This aspect of Tube protocol is perhaps the most confusing. While transfer responses are made exclusively in the appropriate processor, all Tube claim or release attempts and data transfer requests (even the parasite's) take place in the host. It is perhaps easiest to think that, in effect, the Tube host code temporarily adopts the identity of the caller for the purposes of initiating these functions. A much-simplified example of this across Tube dialogue by the 512 may further clarify things, and is included at the end of this chapter after the Tube registers and calls have been described.

After a successful Tube claim and carrying out the subsequent transfer operation, the Tube owner should immediately either issue another Tube command, or release the Tube so as to make it available again to both processors for further operations in either direction. The release is called at the same entry point and with the same value in the accumulator as was used for the claim. The callers identity is then cleared and the Tube busy, flag is reset to 'available'.

The Tube Registers

All eight registers are read/write. In general, when intending to read or write any of the registers, both the host and the parasite first read the status, simply looping back to check again if the status shows 'not

ready'. The exceptions to this are the time critical block transfers through register three, when the status is checked only once prior to the start of the transfer, but not again during it.

The eight Tube ULA registers are memory mapped in both processors. This is always at fixed locations in SHIELA in the host, but varies in the parasite depending on the type and in the 512 depending on the version of DOS Plus. Each register is, however, assigned general purposes and status protocols as shown below, which do not vary with the parasite or OS type. Note that in all the Tube status registers bit six set is shown as 'not full'. It should be noted that, since some transfers are more than a single byte, 'not full' does not mean the same as empty. Frequently all the sender needs to know is that more data can be sent immediately, regardless of whether previous data has yet been received or not.

```
Register 1
Status  - bit 7 = data available + parasite IRQ pending
        - bit 6= not full
        - bit 5 = Set parasite reset active low (BREAK
pressed)
        - bit 4 = Enable 2 byte FIFO register three
transfers
        - bit 3 = Enable parasite NMI from register three
data
        - bit 2 = Enable parasite IRQ from register four
data
        - bit 1 = Enable parasite IRQ from register one data
        - bit 0 = Enable host IRQ from register four data

Data    - read or write.
        Host write generates parasite IRQ
        Parasite read clears IRQ
```

In the parasite to host direction, register one is used for OSWRCH. The data register acts a FIFO buffer big enough to permit the longest VDU command to be processed, thus increasing the chance that the two processors will achieve parallel execution.

In the host to parasite direction, the data register provides a one byte buffer which, when written to, generates an IRQ in the parasite. It is used to pass on host events, interrupts such as are caused by a keypress, and the ESCAPE flag. On initial system start-up, the value of bit 5 informs the parasite that a reset is in progress (i.e. the BREAK key has been pressed). During this start-up sequence, status register one may also be used to set up a returned IRQ to the host.

```
Register 2
Status  - bit 7 = data available
        - bit 6 = not full
```

```
        bits 5 to 0 not used
Data    - read or write.
```

This register is used to implement long (in terms of processor time) MOS calls, or those which cannot be permitted to interrupt the parasite to host OSWRCH serviced by register one. This group of calls, in fact, consists of all the remainder of the standard direct host MOS calls from the parasite except OSWRCH itself. The parasite passes a byte to indicate the call required and this causes a jump in the host code to the appropriate routine. The two machines then continue their dialogue through register two to pass any required parameters needed by the call.

The same register is used to initiate non time-critical data transfers through the Tube when the host code is entered at &406. The type of transfer is signalled by a reason code followed by an appropriate relocation address in the host or parasite (depending on the direction) sent in the usual 6502 four byte form. In the Tube host code these relocation addresses are indirected through X and Y in the normal 6502 manner. If an address in four byte format is not suitable for the parasite to use directly, as in the 512, it is the parasite's responsibility to translate the address to or from a suitable form (e.g. segment: offset), hence the host need not be aware of the processor type currently in use.

Register two is also used as the channel through which requests on the filing system are made, which may result in time critical filing system data transfers such as reading or writing a file. In this case, although the request is issued through register two, the transfer takes place through register three.

The reason codes passed through register two for all these data transfers and their meanings are:

0 - A single byte parasite to host transfer
1 - A single byte host to parasite transfer

These two calls transfer any number of single bytes and terminate in the release of the Tube by the sender, or recommending for a different action.

2 - A double byte parasite to host transfer
3 - A double byte host to parasite transfer

These calls transfer an even number of bytes, as each transfer is two bytes. This also results in a faster transfer than types zero or one, since less status polling is involved per byte transferred.

4 - Execute code

This call is used to instruct the parasite to commence code execution at a given address, as in the initial start-up signal prior to the co-processor entering the bootstrap loader. This call contains an implied Tube release and control is not returned to the caller.

5 - Not used for transfer, reserved for filing system release

6 - A 236-byte parasite to host transfer

7 - A 256-byte host to parasite transfer

Both of these calls transfer exactly and only 256 bytes, or by repeated calls using a counter, multiples thereof. They are the fastest form of data transfer since no status checking is carried out within the block and therefore no other interrupts are generated. Only after a complete block is transferred ray the Tube be released or re-commanded.

Bulk disc data transfers are, by definition, always a multiple of 256 bytes, but for other transfers two strategies are possible. If a transfer of, say, 300 bytes were required, in some cases two blocks totalling 512 bytes might be sent, the extra bytes being discarded. Alternatively, the first 256 bytes could be transferred by these calls and the remainder by means of calls zero to three.

```
Register 3

Status  - bit 7 = data available and parasite NMI generated
        - bit 6 = not full
        - bits 5 to 0 not used
Data    - Read or write.
Host to parasite:
        Host write can generate parasite NMI
        Parasite read clears NMI
Parasite to host:
        Host read can generate parasite NMI
        Parasite write clears NMI
```

Register three is used by transfers which require the attention of the host's NMI owner (i.e. the filing system or the network), hence any data transfer to or from the host may need to also generate an NMI in the parasite.

Register three data can be programmed as a one-byte channel or as a two-byte FIFO channel. The setting of status bits in register one determines whether NMIs should be generated in the parasite by these transfers. If set as a two-byte channel and NMIs are enabled, both bytes must be written to generate a parasite NMI. In the host to parasite direction, the writing of two bytes by the host causes the NMI, which must be cleared by the parasite reading both bytes. In the parasite to host direction, the action of the host reading both bytes causes a parasite NMI which is only cleared when the next two bytes are written by the parasite.

This technique ensures that any action on the part of the host can, if required, produce an immediate response from the parasite, hence the host (and it's own NMI functions) are not delayed. Since there are no physical external devices that can cause either IRQs or NMIs in the parasite (which would in a conventional processor be triggered by direct hardware activity), the Tube registers are used to fulfil this purpose. So as to avoid the need for buffering and relocating in the 512, the Tube register three NMI is processed by the direct memory access (DMA) routine in DOS, hence data is read and transferred immediately to its final memory destination, rather than being buffered and moved.

When register three is set as a single byte channel, slow or time independent transfers are carried out through it which do not generate NMIs in the parasite.

```
Register 4
Status  - bit 7 = data available + parasite IRQ pending
        - bit 6 = not full
        - bits 5 to 0 not used
Data    - Read or write.
Host write generates parasite IRQ
Parasite read clears IRQ
```

Register four is used as the control channel for the register three time critical block transfers. It is also used to transfer errors from the host to the parasite. In both cases the host can generate an interrupt in the parasite by setting the status and placing a byte in the data register.

In the case of fast-register three data transfers, the byte is an instruction byte describing the required action. The two processors then co-operate in passing data back and forth through register four until the parasite removes a synchronisation byte to indicate that it is ready for the register three transfer to begin. This interactive (or the nearest to it in Tube protocol) communication effectively causes the host to wait (for a short time) while the parasite prepares itself for the very fast register three transfer. This action will include disabling its own internal interrupts (i.e. the clock) and pushing all registers on the stack in preparation for the NMI, which will then follow as soon as the host is notified to proceed.

When used for error signalling, the host causes a parasite error interrupt request by writing an error code to register four. The two processors then co-operate in passing the error string through data register two. Typically these errors are as a result of the failure of a MOS or filing system function; (e.g. file not found, drive not ready, corrupted disc, RS232 parity error etc.). Some of these errors are reported verbatim by the 512, others are translated into DOS standard message, before presentation to the user.

It can be seen that, although the setting up of transfer requests and the possible delay before a response is received is asynchronous, if the timing of many transfers, once started, were not carefully synchronised, serious problems could arise. The code in both processors, therefore, has to be matched, not only for function, but in some cases is tuned for elapsed execution time for each operation, once begun.

The timings for the six categories of transfer shown above now follow. If in practice these times are exceeded, data loss or corruption may be incured. In Table 4.1, P refers to the parasite, H to the host.

Type	Direction	Start delay	Service time
0	P to H	24 µ secs	24 µ secs
1	H to P	0 µ secs	24 µ secs
2	P to H	26 µ secs	26 µ secs/pair
3	H to P	0 µ secs	24 µ secs/pair
4	-	-	-
5	-	-	-
6	P to H	19 µ secs	10 µ secs
7	H to P	0 µ secs	10 µ secs

Table 4.1 Timings of Transfers.

Note that not all 512 to host (or vice versa) activity is directed through the calls provided by the Tube host code. Special custornised functions are provided by the extra code in 6502.SYS, which is loaded from disc when DOS is booted. Functions which are not controlled through the host code are essentially all the IBM type operations, since none of these are supported by the host's Tube code, its MOS or its filing systems. These are:

1. 6845 CRTC direct programming for IBM to BBC screen mapping
2. The writing of all screen Characters
3. IBM graphics display bytes
4. Reading and writing of the user port
5. WD1770/1772 direct programming for all disc formats except 640k
6. NMI ownership swapping, necessitated by the above

These functions are discussed separately in the next chapter. The 6502.SYS source code is listed elsewhere with full commentary.

Host MOS Calls

The co-processor MOS calls supported by the 6502 are now detailed as seen from be host side of the Tube. That is, those for which facilities are provided in the standard Tube host code and which are available to all Acorn co-processors. The entry conditions for the 80186 registers for these calls within the 512 are detailed later.

In all cases below, it is implicit that the caller has already successfully claimed the Tube before issuing the request. During any call an error may result, in which case the call is abandoned by the host and the error information returned through a register four interrupt, as already described.

OSWRCH: Host Operating System Write Character

This call is directed through register one. In the 512 this is used for writing to the printer and the RS232 port. The parasite protocol is simply to wait until register one data 'not full' and write a character. Nothing is returned.

OSRDCH: Host Operating System Read Character

This call is directed through register two. The 512 requests a byte to be read from the current input stream. The parasite protocol is wait for register two not full and write &00 to it. A single byte is read by the host's MOS and transferred back, also in register two.

OSCLI: Host Operating System Command Line Interpreter

This call is directed through register two. The parasite protocol is wait for register two not full and write &02 to it. The host code then reads a slow block transfer through register two of up to 256 bytes or until CHR$13 is received. The command string is stored at &700 and immediately passed to the host's MOS. A completion byte (&7F) is then returned to the parasite to confirm completion of the command.

OSBYTE: Host Operating System Byte General Function Call

This call is directed through register two and is implemented in two forms. The first is known as a short OSBYTE, when only the 6502's accumulator and the X register require parameters. The second is the long OSBYTE, when the accumulator and both the X and Y registers parameters to the call must be supplied.

For a short OSBYTE the parasite waits for register two not full as normal then sends the short OSBYTE request, &04. It then transfers the x register value followed by the accumulator value defining the call. The MOS call is issued immediately and the 6502 polls register two status until not full, when it returns the x register value only.

For a long OSBYTE, the parasite must request the call by sending a value of &06. It must then send the X, Y and accumulator values in that order. The host code then calls the MOS routine, but before returning the call tests to see if the call was an OSBYTE 157, which is a 'fast Tube BPUT'. If it was there is no reply and the routine exits back to the idle loop. In this circumstance it is the parasite's responsibility to 'know that no parameters will be returned, so the Tube should be released or re-commanded immediately.

For all other long OSBYTE calls, all three of the 6502 register's contents are returned to the parasite. The sequence of return is accumulator, Y register and then X register.

OSWORD: Host Operating System Word General Function Call

This call is directed through register two, but again two separate versions of the call are implemented.

A general OSWORD call (i.e. any except zero) is indicated by the parasite writing a call value of &08. Next the OSWORD type is sent, followed by the number of parameters which will be required. Finally the parameters themselves are sent and stored in the host's RAM prior to the call.

The X and Y registers are pointed to the parameter block and the call is issued. If no parameters are to be returned the call ends. As for OSBYTE 157, the parasite should know from the type of call whether values will be returned and act accordingly. If returned values are to be transferred back to the parasite the data is sent through register two. The number of parameters sent and returned can be read from the following table.

OSWORD number	Params	
	Sent	Ret'd
1 (&01)	0	5
2 (&02)	5	0
3 (&03)	0	5
4 (&04)	5	0
5 (&05)	2	5
6 (&06)	5	0
7 (&07)	8	0
8 (&08)	14	0
9 (&09)	4	5
10 (&0A)	1	9
11 (&0B)	5	0
12 (&0C)	0	8
13 (&0D)	16	16
14 (&0E)	16	16
15 (&0F)	16	16

16 (&10)	16	13
17 (&11)	13	13
18 (&12)	0	128
19 (&13)	8	8
20 (&14)	128	128
21 (&15) to 127 (&7F)	16	16

Table 4.2 Numbers of OSWORD parameters transferred

For OSWORDs 128 (&80) to 255 (&FF) the number of parameters is embedded in the parameter block at X-Y offset zero for calls, and the number returned after the call is at (X-Y)+1.

OSWORD 0, read line from input, is indicated by writing register two with a call value of &0A. This call is irnplemented separately from the other OSWORD calls as it is a special case. This is because it may well be reading from user input at the keyboard, so firstly it may be extremely slow in processor terms. Secondly, although a maximum input length is specified in the call parameters, the user can terminate the call earlier by pressing one of several keys, therefore the length of returned information cannot be determined until the call is complete.

Finally, the user might press the ESCAPE key, which means the call will be abandoned entirely before completion, not necessarily with an error, and code must be provided to cater for this.

If the call is successfully completed, up to a maximum of 256 characters may be retumed through register two, or up to the first encounter of RETURN (CHR$13).

If the call is abandoned by the user pressing the ESCAPE key, the call is terminated, input is discarded and the escape is signalled by returning a single byte of value &FF.

OSARGS: Read/Write File Attributes or Read Filing System Type

This call is directed through register two and is requested by the parasite setting a call value of &0C. The parasite then transfers the file handle concerned, (or zero if none) followed by a varying number of actual parameters, though four bytes are always sent. This is followed by the OSARGS call type in the accumulator.
The call is immediately executed and four bytes (not all necessarily significant) are returned through register two.

OSBGET: Get (Read) a Byte from an Open File

This call is directed through register two, and is indicated by a call number of &0E. The file handle is passed, transferred to Y and the call is issued. If successful a single byte is returned, otherwise an error is generated and the call is abandoned.

OSBPUT: Put (Byte) a Byte to an Open File

This call is directed through register two, requested by a call number of &10. The file handle is first passed, followed by the byte to be written. The call is issued immediately and if successful, a single completion byte (&7F) is returned.

OSFIND: Open or Close a File for Byte Access

This call is directed through register two, by setting a call request number of &12. First the file handle is transferred, followed by the operation code which the host code stores on the stack, while the filename is transferred by the 'slow' block transfer using register two. The operation code is restored and the call is issued. If successful a single byte, the file handle, is returned, or zero for failure.

OSFILE: Read or Write a Whole File or its Attributes

This call is directed through register two by issuing a call request number of &12. This is followed by the sixteen-byte parameter block required by the call. The filing system call is then issued. The result is transferred back through register two, followed in all cases by the sixteen bytes contained in the parameter block.

OSGBPB: Multiple Byte Read or Write an Open File

This call is directed through register two by a call request of &16. The call parameters are then transferred followed by the call type. After the call the returned parameters are transferred through register two, followed always by the accumulator value, which contains the original call type.

A Short Conversation

As a brief (in real time, if not in words) illustration of how the host and parasite interact in their various activities, the following simplified scenario (missing out many of the actual steps) demonstrates how ownership of the Tube is swapped back and forth during even the simplest operations.

The user wishes to copy two files from drive A: to B: while a text file is printing in the background. The first copy command has been typed and RETURN pressed. Typing ahead of the next command continues.

DOS attempts to claim the Tube which is already in use. However, it was being used by the background 'PRINT' utility and the transfer was held up by the fact that the last character written to register one was not read by the host, which was processing and buffering a keyboard entry on an interrupt request. Since the Tube claim is successful DOS requests a filing system operation to open the input file and then must wait for a response. At this point two current 512 to host requests are outstanding and one host to parasite keypress is pending.

The host has now finished its keyboard interrupt processing and so it polls the Tube, finding a character to be transferred to the printer buffer from register one, which it proceeds to do. It next finds the outstanding register two filing-system request, the data for which it then transfers from the 512. The file is successfully opened and the file handle is returned to the 512, clearing the second request. By this time another host interrupt is outstanding (held up by the filing system NMI) and DOS is ready to request the output file to be opened. The Tube claim is successful (still owned) and the next request is issued by the 512, during which time the host scans the keyboard again and also writes another character to the printer. Immediately after its interrupt the host polls the Tube and finds the next file opening request, which it exits, again returning a file handle.

DOS now Issues a filing system request to load the 'from' file, then releases the Tube. The host passes the filing system request on, which generates a host NMI. The filing system claims the Tube (now free) and loads the file data straight to the 512 DMA routines via register three, releasing the Tube on completion. This generates an NMI in the 512, which therefore temporarily 'forgets' about printing and keyboard entries while the bulk data transfer is received. Next DOS claims the Tube (again free) and issues a request to write the data to the output file. On receipt of the reply the 512 waits for a farther signal from the host to tell it to proceed. The signal is produced when the host filing system is ready to write the data, which again generates an NMI in both the host and the 512 for the duration.

On completion of the read/write cycle (which may be repeated several times, depending on the length of the file) the Tube is released and normal interrupt activity resumes in both processors. The host will again scan the keyboard regularly and service its other normal interrupts, while the 512, under clock interrupts, will request keyboard entries and resume background printing.

During this type of operation, however lengthy printing will almost certainly visibly continue as normal because of the size of the printer's own integral buffer. After the operation is complete, the print routine and the host will easily 'catch up' and refill the printer's buffer, so the process remains 'non-stop'. However, the reason that keyboard characters may be 'lost' when typing ahead during disc activity can easily be seen. The cause is the fact that the interrupt routines of both the host and the 512 can be delayed for (relatively) quite lengthy periods during such operations. If this occurs during a host to parasite filing system transfer, one or two key depressions maybe lost by not being recorded by the host.

This point is mentioned because, as will be seen in the next chapter, the 512 requests key presses, even at its maximum rate, only half as often as the keyboard is scanned by the host, and the actual rate is frequently much less when other activity intervenes.

5: 6502.SYS

Functional Overview

In addition to the facilities provided by the Tube host code, the 512 co-processor boot process loads additional 6502 resident routines to support the non-standard facilities of the required IBM hardware emulation. Broadly, the control offered by these routines falls into three categories:

1. Console I/O - IBM screen display and extra keyboard functions
2. IBM and CP/M disc formats requiring direct FDC programming.
3. User port control (mouse or trackerball).

The extra code in `6502.SYS` is necessary for the 512 because, in native mode, the host MOS or filing system does not provide support for these facilities in the correct form, or at all.

In order to allow these facilities to be 'bolted-on' to the standard Tube host code and MOS facilities without disturbing the availability of existing functions, accepted 6502 coding practices are employed.

During the 512 system boot, the required code is first loaded into the 512 and then downloaded (both operations using the standard host code facilities) to reside at &2800 in the host's main RAM. The new code is then initialised, which results in the re-direction of three vectors. The user vector at &200 is re-directed to enter the program &2803, the host's main interrupt vector, IRQ1V at &204, is re-directed to point to an entry at &2834, and the event vector, at &220, is re-directed to enter the code at &2831.

It should be noted that the `6502.SYS` code claims and releases the Tube and also carries out some data transfers itself by means of the standard facilities in the Tube host code, which it enters at &406, but for reasons of efficiency and execution speed some MOS calls and Tube transfers are also carried out directly from within the code.

So as to achieve the desired speed of operation for such a rapid processor as the 80186, many of the functions of `6502.SYS` use techniques that are not strictly (or even vaguely) legal in other circumstances. Two obvious examples are directly poking the screen map for screen output, and bypassing the host's filing system by directly programming the FDC. For the latter operations `6502.SYS` also takes control of the NMI for certain commands, thus preventing any interference by the host's normal interrupt services, or indeed from external NMI sources such as a network fileserver.

This is why some 512 operations, such as reading or writing IBM format disc data can lead to lost keystrokes if characters are typed at the same time, or screen updates may be slightly delayed, though they are rapid when they do finally resume. For the duration of these NMI operations all normal host

functions are effectively 'turned off' and are only permitted to resume on completion. As a result, most of the functions in `6502.SYS` are time critical and must be completed in the shortest possible time, hence the coding techniques employed.

`6502.SYS` is entered on one of three conditions, two of which provide a degree of autonomous (from the 512) control and monitoring capability of both hardware and software functions in the 6302 host. When such monitoring produces an FDC result, the 512 is informed by means of an IRQ. The third entry point allows the 512 to directly control `6502.SYS` operations on demand.

Event Processing

The main 'automated' entry to `6502.SYS` is via the event vector, on event type four (screen vertical synchronisation, or vsync), which occurs fifty times per second in the host when the event is enabled. Adopting this technique has significant implications for the overall efficiency and performance of the system.

First, vsync uses existing, but otherwise unused, normal host event facilities. By examining the event code entry into `6502.SYS` at &2AFC, it can be seen that only an extremely small penalty is incurred by the host processor when the event type is other than four. When the event is type four, any possible screen display disruption is minimised, since much of `6502.SYS's` activity is performed during frame flyback, which occupies approximately 50% of the total elapsed time.

Secondly, when `6502.SYS` controlled functions are required, the 512 can send the information across the Tube very rapidly without having to wait while the function is performed, as this will be entirely automatic. After the information has been passed and buffered, the 512 can return to its own processes and leave `6502.SYS` to carry out the function on its own as an independent 6502 routine.

This is best illustrated by many of the initial commands involved in floppy disc control. While the resulting transfers may incur high data rates, initial command execution (opening a file, for example) requires small amounts of data but is relatively very slow. In 80816 terms it takes a great deal of time to get the drive up to speed and to move the read/write heads into position, during which time the 80186 could process, for example, a large amount of applications code, some background printing and perhaps several complete clock interrupt cycles

Finally, because `6502.SYS` contains very specific code for 512 operations, the 512 does not need to interrupt its own processing to claim the Tube to issue specific calls to monitor the progress or results of previous commands. Commands issued to the host via `6502.SYS` are automatically monitored by the 6502 code itself.

For some types of command, the 512 does not wait for results to be produced by the subsequent host processes because there aren't any. For example, after transferring the bytes required to perform a

particular action, the 512 may assume it will be carried out successfully. In practice, whether it is or not, for some host calls there will be no returned information, therefore `6502.SYS` can be left to perform the activity alone, regardless of the outcome.

For calls in which the 512 does have a further interest, for example the result of a previously issued disc command, the 512 can still return to its own activities and continue its own processing, knowing that `6502.SYS` will monitor the situation on every vsync and report only when results are finally available.

After the prime vsync operations, the 512 also has the option of performing any or all of several ad-hoc activities a number of times before releasing the Tube.

Unknown OSWORDs

The major secondary entry point is 'on demand', via the user vector which enters `6502.SYS` at &2803. This occurs on an unknown OSWORD call, which was issued originally by the 512 via the standard Tube host code. OSWORD calls which have a number greater than &E0 are not processed by the MOS and should not be used by other proprietary software. They are reserved for user code and so, when such a call is received, the MOS directs execution to the code pointed to by the user vector. If this vector has not been redirected, as in an idle host, the result is the production of the familiar `Bad command` error.

For the 512 OSWORD numbers between &FA and &FF are used, though within this range only three calls are functional in version 2.1 of `6502.SYS`. OSWORD &FA is implemented separately, while OSWORD &FD is not implemented at all and OSWORD &FE is harmless but non-functional, simply jumping to an RTS. On entry through the user vector the accumulator value is compared to each known `6502.SYS` OSWORD and the code is entered according to the action required. The unrecognised OSWORD calls used or reserved by `6502.SYS` in DOS Plus version 2.1 are:

```
Call    Used for
&FP     Fast graphics update
&FE     Hard disk read/write (never implemented)
&FD     Not used - was originally for text output
&FC     Cursor control, soft scrolling and mouse
&FB     FDC track/sector seek and read/write operations
&FA     Not in 6502.SYS - now implemented separately in
code downloaded from 512 ROM
        to reside at &2500 in host main memory.
```

Table 5.1 OSWORD calls used or reserved in DOS Plus, v2.1

It should be remembered that booting DOS and `6502.SYS` initialisation take place after normal host initialisation on a hard break, which means the user vector is always redirected to `6502.SYS`. Those

who might wish to interface alien devices by means of unknown OSWORDs should note that such extra code as is required, ideally and for reliability, should be implemented in sideways RAM using calls with a value of less than &E0. All such unknown OSWORDs are offered to the ROMs on a service call of type &08, prior to being passed to the user vector if unrecognised. Call numbers above &E0 are never passed to paged ROMs.

The only valid alternative is to modify the jump table in `6502.SYS` at &2813, adding a jump to your own routines, which must in turn jump to the original address in &2813 after execution with all registers preserved. In other words the technique is directly analogous to intercepting normal host vectors. This is vital, since yours isn't the only routine relying on `6502.SYS`. Even when no other third party software is involved, the OSWORD &FA code at &2500 must always be the last in the chain, as can be seen by reference to the source code in appendix F. Examination of the source listings will show that the final jump to the default handler is NOT vectored and cannot be intercepted.

Note also that if this method is chosen, none of the existing entry points or actresses in `6502.SYS` may be moved. You may 'plug in' only by an extra jump at &2813, as neither `6502.SYS` nor the OSWORD &FA code is relocatable. It should also be noted that very little of the host's RAM remains available when the tube is active, particularly when running a 512 - yet more reason to implement extra routines in paged ROM.

It is absolutely impossible to install an independent 6502 main RAM 'unknown OSWORD' routine. Any attempt to issue any unknown OSWORD to 6502 RAM resident routines when the 512 is active (except one of those listed, or one added by you by the above methods) would result in the equivalent of a 'bad command' error and would hang the machine, since the 512's legitimate OSWORDs would never be executed.

IRQ1V

The final entry point to `6502.SYS` is via IRQ1V, which is re-directed to enter the code at &2834. It is used for GEM mouse control only. Again, the entry uses existing host facilities and again, if the interrupt request is of no interest to the 512, the processing overhead in the 6502 host is very little.

OSWORD Functions

6845 Control

Entry to `6502.SYS` on an OSWORD &FC (issued through the Tube host) is used to directly program various hardware devices and to initialise `6502.SYS` and the host after a system boot.

The routine is initially entered at the routine which programs the host's 6845 Cathode Ray Tube

Controller (CRTC - the screen display controller) at &2A7F. If the byte received has the top bit set, it signifies one of the other functions or termination. Any byte with a value of or less indicates the 6845 is to be programmed.

Although using native screen modes as a basis, the IBM screen formats have differences and these are programmed directly into the 6845 within this routine. Two bytes are read from register two for every CRTC register change, the first being the control register number, the second is the new value to be written to it. On receipt of a (first byte) value of &So or greater the 6845 programming loop terminates.

After programming the CRTC, the OSWORD routine may continue or not, depending on the value of the terminating byte. If &FF was received the routine exits (e.g. after a PC screen mode change).

If a value of &FE is sent the mouse code is initialised, followed by the re-direction of IRQ1V and the setting of the user-port direction register to input. Following this, user-port interrupts are enabled and the peripheral control register is re-programmed to allow mouse input. The routine then jumps back to the original 6845 control entry to await further commands or to terminate (on &FF).

If a value of &FD is sent the event vector is redirected and the routine jumps back to the original 6845 control entry.

If &FC is received it is ignored and control returns to the original 6845 control entry. Any value below &FC causes an immediate exit from the OSWORD without return to the control entry. OSWORD &FC was originally used to write to the user-port, but this is no longer supported.

Screen Output

Screen output is carried out by entry to `6502.SYS` on an OSWORD &FF. As many of the displayed characters may be IBM special Characters, not supported by the host, all characters including normal text characters are sent as eight bytes of bit patterns.

For an explanation of how an 8x8 Screen character matrix is defined, examine the setting up of user-defined characters using VDU23 in native mode and screen character mapping in 80 and 40 column native screen modes.

Users will have realised from the screen display that the normal BBC Micro character set is used for alphanumeric display, but these definitions too are stored in the 512 and transferred in the same way.

This permits character highlighting or inverse video, while still using the standard Acorn screen character matrix which, it must be acknowledged, does not permit much variation. A further benefit of this approach is that the bit manipulation is carried out by the very much faster 80186, rather than the 6502. The steps in the routine following are numbered to aid references to previous steps, which may or may not be repeated depending on how many consecutive bytes or areas of screen are to be updated.

1. First the memory start address for the update is sent. This is actually the low byte high byte screen RAM address which is to be updated, but always sent high byte first. If instead of a high address value (which by definition cannot be zero) the first byte is &00, the OSWORD is complete and the routine exits.

   The screen address is stored at &70 and &71 in zero page, but the low byte of the base address held in &70 is always set to zero, with the real offset held in register Y. Because screen addresses always occupy a number of whole RAM pages, the base of the screen's current RAM block address is always the stored as the start address of the page. When this is combined with the index value in y, the result addresses RAM at the correct byte.

   This permits quite an elegant technique for writing a full screen or large contiguous sections of screen very quickly without the need to transfer updated addresses through the Tube. This is used to good effect in the fast output routines described below.
2. Next the program reads Tube register two status until a sync byte indicates that data or a command byte follows.

   If the byte following the sync byte is &FF, the currently addressed output is complete and the routine jumps back to the original entry point to read new memory co-ordinates at 1. This follows at least one and possibly numerous iterations of 3 and/or 4 below.
3. If the next register two data byte following the sync byte is &00, an irregular character definition pattern is to be sent, and these bytes are written directly to the screen RAM as they are read front register one, in the linear routine at &2A43, labelled `transfer_loop2`. After the character is written, the routine jumps back to wait for another sync byte at 2, which may result in immediate output of another character in the next screen position, or the sending of new co-ordinates.
4. If the next byte is any other value (than &00 or &FF), then part of the screen is to be cleared by writing spaces to it. If the character is a space all eight bytes of the character's fill pattern are of course the same, and so for this operation, only a single fill byte need be read. The program reads a single space fill byte from data register one and immediately writes the byte eight times in linear code without the need to read more data. After the character is written, the routine jumps back to wait for another sync byte at 2 as detailed in 3.

This technique results in very rapid screen updates because, not only are spaces by far the quickest characters to write, they are by far the most common. In graphics modes, of course, a space character is replaced by the pattern which represents the background colour, so the method still holds.

Note that after outputting every eight byte pattern the code jumps to read the next sync byte, at 2. If a contiguous area of the screen is to be updated, since both the space fill and the irregular pattern fill routines include code to implement both the index value in register Y and, on Y overflow, the high byte of the page address held in &71, no new addresses need be read through the Tube. The next screen fill bytes can be sent and output immediately, resulting in remarkably rapid bulk screen updates, so rapid, in fact, that the host's video hardware frequently visibly cannot keep up.

Block transfers

Direct access to the host's memory is achieved indirectly through `6502.SYS` by an OSWORD &FA. The code is documented in Appendix F and the entry parameters from the 512 are detailed in chapter 6.

This OSWORD is the only facility by which direct access may be gained to read or write data between the two processors completely independently of other functions. In addition it is the only method available for reading or writing either to shadow RAM (on a Master 128 or B+) regardless of the MOS shadow configuration, or directly accessing paged ROM or sideways RAM.

Apart from initial system load, it is not used by any standard DOS Plus facility.

WD1770/2 Control

Entry to `6502.SYS` via an OSWORD &FB is used for floppy disc control for all 512 DOS Plus disc formats except Acorn 640k, for which the code in the ADFS is used. This is the reason that all other DOS disc handling, even of the low density IBM 320k and 360k formats is very much faster than the native BBC 640k ADFS format, which must be used to boot the system (as explained in Chapter Three).

This series of operations is the most complex in `6502.SYS` and a full explanation would be extremely lengthy and somewhat pointless. Therefore the functions are described here at a high level. A detailed exposition on the programming of the floppy disc controller is not relevant in this context even for 80186 machine code programmers, unless they wish either to invent a new disc format or to hack protected IBM discs used by some software suppliers. For those with sufficient knowledge and inclination the source listing of `6502.SYS` is fully commented and all memory locations used and the FDC registers are identified.

The program is always entered at `OSWORD_fdc` (&28E0), via the unknown OSWORD passed through the Tube host code. The various instruction codes are passed directly through Tube register two for each operation to be performed. The first requirement prior to any FDC activity is that the NMI must be claimed from the current owner. This involves issuing a (direct) OSBYTE &8F, which is passed to the paged ROMs. The current NMI owner should respond to this by saving any of its important NMI data to its private workspace and releasing the NMI. This it does by returning its identity in the Y register. The previous NMI owner's ID is then stored by 6502.SYS, for return of NMI ownership on completion of the 512's FDC operation.

On successfully checking the NMI, the next step is for 6502.SYS to claim the Tube, which might have been released since the OSWORD was first issued. Next, four bytes are passed which are stored in the host's RAM and will be used to program the DMA transfer to or from the 512's RAM.

Following this, the identity of the host (recorded during system boot) is passed to 6502.SYS from the 512 and 32 bytes of the NMI data image, stored within the 6502.SYS code area, are copied down to page &D, the host's NMI work area. Next the appropriate host type's addresses are set up for the three memory mapped FDC control registers in SHEILA.

These locations differ between Master 128 hosts and Model B/B+ machines. The byte passed to these routines identifies the host as a Master (1) or as a B, B+ (2) so that the correct SHEILA indirection addresses are stored in &A0 and &A1 in zero page.

Having claimed the Tube, claimed the NMI and set up the FDC base for the particular host type, the FDC command can be executed. This requires parameters to be passed through register two for the intended operation. As various different formats of disc are processed by this code, the parameters must completely describe the operation to be performed in every detail (e.g. drive ID, start track, start sector, sector size, number of sectors, read or write, time allowed, etc). The command is executed and 6502. SYS enters a loop, monitoring the FDC status until an error, a timeout or a successful completion. Whatever the termination type, an FDC event is returned to the 512 together with the FDC status, possibly though not necessarily, within this call - see event functions overleaf.

If the operation is a read or write, data is transferred, a sector at a time, by means of DMA interrupts generated in the 512. Even when the read/write activity is continuous, the DMA routines in the 512 and the Tube transfer speed are more than fast enough to keep pace with disc transfer rates. On completion of all reads/writes and transfers, the Tube is released and the NMI is released back to its original owner. In practice, particularly for the slower disc formats, the 512 may issue the commands for a single logical disc operation as several separate OSWORD &FBs, causing 6502.SYS to release and reclaim the Tube several times. For example, the simple operation of writing a single file could well translate to mean separate calls to:

1. Read the catalogue
2. Read the file allocation table (i.e. identify the media format)
3. Create a new file entry
4. Open the new file for output
5. Write the data
6. Update the catalogue
7. Update the file allocation table
8. Close the file.

Some of these are slow operations and none except the data transfer (5) will require the 512's permanent attention. In these cases the 512 can continue to process its own code and 6502.SYS will monitor FDC status in the event four entry, described below, until the 512 is again required to intervene.

Hard Disc Control

The hard disc OSWORD &FE, is included for documentary purposes only. A 512 DOS Plus hard disc partition is allocated as a single large file on an ADFS disc and standard ADFS 'get byte' and 'put byte' routines are used. This is done for two reasons, the prime one being that the hard disc interface used in the DISC Micro host (SCSI with Acorn adaptor) is not standard for DOS systems and requires customised code to drive it. Duplication of this code is unnecessary, since it already exists in the ADFS.

Using the ADFS ROM code (via FS commands issued through the Tube host code) has two benefits. First it permits ADFS to maintain its view of the DOS partition as a normal file and therefore a mixed native ADFS/DOS Plus disc is possible. Secondly, from DOS's point of view (through the XIOS, which does the logical to physical translation work), the hard disc appears merely as a normal but large DOS drive, no different to any other.

Event Functions

6502.SYS is entered on vsync, (nominally) fifty times per second, at &2AFC, labelled 'eventcode'. Events other than four are ignored. This entry point is used for monitoring progress of any oustanding FDC functions, monitoring and reading the keyboard, keyboard LED control, screen re-mapping and reading the userport.

Outstanding FDC Operations

The first operation checks the FDC status to see if an FDC command (which was previously left to its own devices for the duration) has now completed. If so, the code diverts to signal an FDC event to the 512 by generating a host event ten, which is passed through Tube register one by the MOS, followed by the contents of 6502 registers A, X and Y to generate an IRQ in the co-processor. The 512 will then issue the appropriate OSWORD to execute the next step in the FDC command, or take suitable action if an error has occurred.

Keyboard Processing

If no FDC event is outstanding, the states of the SHIFT and CONTROL keys are tested. First a call to the MOS's keyboard vector is made. On return the two 6502 processor status flags N and V indicate the state of the SHIFT and CONTROL keys. If N, the negative flag (bit 7), is set the CONTROL key was pressed at the time. If V, the overflow flag (bit 6), is set the SHIFT key was pressed at the time.

The MOS's two key rollover locations, &EC and &ED, contain respectively the current key pressed and the last key pressed if two keys were depressed at the same time. The current keypress is transferred to register X with the top bit set on if CONTROL was pressed, off otherwise. The last keypress (if any, zero otherwise) is transferred to register Y, and the top bit is set on if SHIFT was pressed, off otherwise. As can be seen from this arrangement, it is impossible for DOS software to recognise more than two simultaneous ASCII key depressions, with optionally SHIFT and CONTROL, hence any PC software

requiring three simultaneous ASCII key presses cannot possibly be made to work in the 512.

Immediately after the keyboard scan, the event type (four) is reloaded into the accumulator and the event is passed on to the normal host routine by means of a subroutine call. All host events are passed to the co-processor by the MOS via a Tube register one parasite IRQ, along with the 6502's three processor registers, and this is the method used to pass host keypresses to the 512. However, as the host event routine was called by a JSR control returns to 6502.SYS and further operations may then follow as described below.

Asynchronous Operations

When the call returns, the 512 will have placed a command byte in register four as a result of the register one event IRQ. This point is labelled `async_command` in the source code, since any number of *ad-hoc* commands can be sent to 6502.SYS via this re-entry without recourse to further unknown OSWORD calls through the Tube host code, thus avoiding their attendant processing overheads and delays. Four possible command bytes are passed.

1. A value of zero indicates no further action. Exit 6502.SYS.
2. A value of one 'drops through' to the 6845 CRTC control routine, which executes as previously described under OSWORD &FC. This command is used for colour or PC screen mode changes.
3. A value of two is an instruction to read the user port data. This routine allows for a trackball or an AMX type mouse, with due allowance for the hardware differences. The user port (&FE60) is read directly for any button presses and these are transferred through Tube register one, causing an IRQ. The four RAM locations holding the two 'low byte-high byte' screen RAM Y-X co-ordinates of the mouse pointer are then read and are also transferred through Tube register one. The reading of the user port for mouse movement, and the translation and storing of those movements in RAM, is carried out in the host IRQIV intercept routine controlled by the host's interrupt timers (See below).
4. A value of three causes Tube register four to be read for the x parameter required for an OSBYTE &CA (*FX 202), which writes the keyboard status byte to agree with the 512's current recorded SHIFT/CONTROL settings. Following this, an OSBYTE &76 (*FX 118) is issued to set the keyboard SHIFT and CONTROL LEDS appropriately to reflect the actual situation.

Each of these operations terminates by returning to the entry point at `async_command` to read further possible commands, or until a zero is received to terminate asynchronous operations.

IRQ1V Functions

The User Port

The redirected IRQ1V enters 6502.SYS at &2B8C and the interrupt flag register is loaded from

SHEILA. The value read is masked to test for user port input. If there is no user port activity, the routine jumps to the original vector address for the host to service its own IRQ facilities.

The remainder of 6502.SYS, between &2B8C and &2C2F, is concerned with reading the user port hardware data lines to determine the direction and magnitude of the mouse or trackball movement.

This involves examining each of the four possible hardware direction signals for activity and translating them into screen mapped Y-X movements, allowing for pointer movement off the edge of the current screen limits. The resulting movement is finally stored in Y-X co-ordinate low-byte-high-byte form in four RAM locations, which are subsequently read by a call to `async_command` during a vsync event (see above).

6: DOS Plus MOS Calls

Having seen how the standard Tube host code and the extra facilities provided by 6502.SYS are implemented on the host's side of the Tube, this chapter details the MOS and filing system calls which can be made from within 512 code.

In 512 user programs, any language which includes the ability to load nominated 80186 registers with specified values, directly as in machine code or indirectly as in, for example, BASIC's CALL or USR functions, can access the host's MOS or filing system via these calls. The various facilities of 6502.SYS can likewise be accessed by means of issuing the appropriate unknown OSWORD call.

Note though, that in general applications programming it is neither necessary nor desirable to directly program the host's hardware device controllers or internal software flags, the FDC, perhaps, being the most obvious example. Most languages have specific filing system support commands for loading or saving data and these should be used whenever possible. Screen mapping too should remain transparent in DOS applications code.

If standard DOS Plus or CP/M facilities are used, not only will the resulting programs be easier to write and understand, they will also be more easily maintained or transferred to a different machine or a different language, should the need arise. 512 users, perhaps more than others, should appreciate the penalties of dubious programming techniques or 'illegal' direct hardware device control. Programs which use MOS Tube calls will no more work on a PC than PC programs that used direct PC hardware access do on the 512.

The following information is intended to allow programs to be written for the 512 which must, for particular reasons, gain direct access to the MOS or filing system calls of the host. Readers should note that, as we are now referring to 80186 registers and DOS functions, 86 series conventions apply and hexadecimal values are written according to the normal 86 standards. For example, a value of 127, which in native BBC mode would be written as &7Fh, is shown here as 7Fh.

Calling the MOS

All Operating System functions required by code running in the 512 under DOS Plus are made by calling a DOS interrupt. In fact, unless illegal coding techniques are employed, all program communication with all other software and the machine's hardware should be achieved through these calls. The bottom 1024 bytes of DOS memory is reserved for the interrupt vector table, each entry of which points to the DOS routines designed to handle the particular type of call Each interrupt vector entry is a four-byte segment:offset address, therefore the full range of interrupts number 256(1024/4).

The MOS Operating System calls of the Tube host code detailed in Chapter Four, and the unknown

OSWORD giving access to 6502.SYS explained in Chapter Five, can be called from the 512 by means of several (otherwise unused) DOS interrupt numbers appropriated for the purpose. Of the possible total of 256 hardware and software interrupts, the majority are unused in all DOS versions to date. Although more will inevitably be used by succeeding versions of MSDOS or PCDOS and later hardware, even as far as MSDOS version 3.3 only (some of) the interrupt numbers up to 2Fh, plus 33h, 44h and 67h had been allocated.

In the version of DOS Plus supplied for the 512, interrupt numbers between 040h and 04Ch are used for the eleven different MOS call types supported by the 6502's Tube host code, plus two MOS calls which are simulated. Some MOS calls, for example OSBYTE, normally require parameters to be placed in the 6502's registers. When called from DOS the values are placed in the 80186's AL, BH and BL registers. When the call is passed to the host for service through the Tube protocol, these values will be transferred to the 6502's A, X and Y registers respectively.

The following is an outline of the supported call types, but users wishing to employ them should refer to suitable MOS reference texts for the precise details of parameters and how the calls are implemented in the host.

MOS Call Types

The 13 MOS calls supported by the Tube host code are listed in Table 6.1 below in order of ascending interrupt call numbers. Interrupts 7h and 48h are included in the table only for completeness, though these are not supported by the Tube host code, as can be seen from Chapter Three and the host code Source listing.

Interrupts 47h and 48h are actually simulations provided in the XIOS to avoid the need for two more explicit calls in the Tube host code or 6502.SYS. A call to INT 47h, OSASCI, results in the current output character value being tested. If the character is Dh, (CHRS13) then INT 48h is called, if not then it is output directly through a call to INT 49h. A call to INT 48h, OSNEWL, simply loads Character Dh and calls INT 49h, then loads Ah and calls INT 49h.

```
Routine INT Function
OSFIND  40h Open or close a file for byte access
OSGBPB  41h Multiple byte read or write an open file
OSBPUT  42h Put (write) a byte to an open file
OSBGET  43h Get (read) a byte from an open file
OSARGS  44h Read/write file attributes or read filing
system type
OSFILE  45h Read or write a whole file or its attributes
OSRDCH  46h Host operating system read character
OSASCI  47h simulated, if Dh also send Ah through INT 49h
OSNEWL  48h simulated, output Ah and Dh through INT 49h
```

```
OSWRCH  49h Host operating system write character
OSWORD  4Ah Host operating system word general function call
OSBYTE  4Bh Host operating system byte general function call
OSCLI   Host operating system command line interpreter
```

Table 6.1 MOS calls supported by the Tube

The only 'standard' additions to this list are the functions provided by 6502.SYS, explained in the preceding chapter, or routines you may implement yourself by patching an additional 6502 routine into the jump table in 6502.SYS. Such a routine must, of course, be loaded after booting the system.

It should be understood that the following calls, when referencing a disc filing system operation, relate only to ADFS format (that is, bootable DOS 640K or true ADFS) or DFS format discs, not to any of the IBM DOS formats controlled through 6502.SYS. It should also be appreciated that, unless you are accessing a true MOS format disc for a special purpose, many of the host's filing system options are meaningless in DOS.

The same is true for the more general MOS calls which must be used with care. As a dramatic and salutary demonstration, you can prove this very simply by trying this example. While `*FX13, 4` in native mode produces no visible effects, (the MOS does not normally use this event itself) the same command issued as `STAR FX13, 4` from DOS will instantly and permanently paralyse the system by disabling all 6502.SYS operations.

As can be seen in the preceding chapter, event four is critical to the functioning of 6502.SYS, particularly for keyboard input. While both the MOS and DOS actually continue to function after this 'FX', you will have irreversibly severed their (and your) only means of communication through the Tube. It is not, in fact, just the keyboard you have disabled, all inter-processor communication is triggered by event four, which causes 50 IRQs per second in the 512 and which you have now permanently (until re-booting) disabled.

MOS Call Definitions

OSFIND - INT40h

Function:	Open or close a file for byte access
On entry:	AL specifies the operation DS contains the segment pointer to the filename BX contain the offset pointer to the filename

Operations:	AL = 0	The file is to be closed
	AL = 40h	The file to be opened for input
	AL = 80h	The file to be opened for output
	AL = C0h	The file to be opened for input/output

The filename must be terminated by Dh (CHR$13).

On exit: AL contains the returned file handle, or 0 if file has been closed or could not be opened.

80186 flags: Undefined

OSGBPB - INT 41h

Function: Read/write a block of bytes from / to a specified open file.

On entry: AL specifies the operation type
DS:BX Point to a file control block which may be in either processor,
where bytes contain the following

00	= File handle
01-04	= 4-byte DMA address pointer (low byte first)
05-08	= No of bytes to be transferred (low byte first)
09-0C	= Pointer value for transfer (low byte first)

Operations:	AL = 1 Put bytes using new sequential pointer
	AL = 2 Put bytes ignoring sequential pointer
	AL = 3 Get bytes using new sequential pointer
	AL = 4 Get bytes ignoring sequential pointer
	AL = 5 Get media title and boot option
	AL = 6 Read current directory and device
	AL = 7 Read current library and device
	AL = 8 Read filenames from current directory
On exit:	AL = 0 Operation was attempted AL = entry value: Call not supported by current host FS.
80186 flags:	CF unset = Call completed succesfully CF set = Call failed Others = Undefined

OSBPUT - INT 42h

Function:	Write a single byte to an open file using the file's sequential pointer
On entry:	AL contains the byte to be written BH contains the file handle (provided by OSFIND)
On exit:	Nothing returned
80186 flags:	Undefined

OSBGET - INT43h

Function:	Read a single byte from an open file using the file's sequential pointer.
On entry:	BH contains the file handle (provided by OSFIND)
On exit:	AL contains the byte read from the file
80186 flags:	CF is set if an attempt is made to read past the end of the file- others are undefined.

OSARGS - INT 44h

Function: Read/write an open file's arguments or read the current filing system type
On entry:

On entry: AL = operation type
AH = file handle (provided by OSFIND) or 0.
BX points to a 4-byte attribute block, which must have been previously set up in the host's RAM

Operations: If AH = 0 and
AL = 0 Return the current filing system type
AL = 1 Return the address of the rest of the command line in the zero page control block
AL =FFh Update all files to the media (i.e. ensure that all file's buffers are written)
If AH is greater than 0 and
AL = 0 Read sequential pointer of file
AL = 1 Write sequential pointer of file
AL = 2 Read length of file
AL = 3 Write the length of the file
AL = FFh Update the file to the media

On exit: If entered with AH=0 and AL=0, AL contains the filing system number as follows:

0 No filing system

1 1200 baud cassette

2 300 baud cassette

3 ROMFS

4 DFS

5 ANFS/NFS

6 TFS

7 IEEE FS

8 ADFS

If entered with AH = 0 and AL = 1, then the address of the remainder of the last command line is returned in a four byte zero page control block pointed to by BX.
This address is always in the I/O processor and its data should be read using OSWORD 5.
The remainder of the last command line is always terminated by 13h (CHR$13).
For all other calls no data is returned unless an error is reported via a normal register four IRQ.

80186 flags:

OSFILE - INT 45h

Function:	Read/write a complete file or catalogue information	
On entry:	AL contains the operation type DS:BX point to the file's control block	
Operations:	AL = 0	Save a block of memory as a file
	AL = 1	Write the information in the parameter block to the catalogue of an existing file
	AL = 2	Write the load address of an existing file
	AL = 3	Write the execution address of an existing file
	AL = 4	Write the attributes of an existing file
	AL = 5	Read a file's catalogue info with the file type returned in AL and the info returned in the parameter block
	AL = 6	Delete a named file
	AL = 7	Create a named file without transferring data
	AL = FFh	Load a named file
		The file control block is in the form:
	00-01 =	Address (lo-hi) of the filename
	02-05 =	Four byte load address (low byte first)
	06-09 =	Four byte execution address (low byte first)
	0A-0D =	Start address of data for save (low byte first) or file length otherwise
	0E-11 =	End address of data for save (low byte first) or file attributes

File attributes are stored in 4 bytes. The most significant 3 bytes are filing system specific. The least significant byte indicates the following when a bit is set on:

0 No read access to owner (i.e. filename /WR)

1 No write access to owner (i.e. filename LR/)

2 Not executable by owner (i.e. filename /)

3 Not deletable by owner (i.e. filename L)

4 No public read access

5 No public write access

6 Not executable with public access

7 Not deletable with public access

ADFS Note: The three most significant bytes are undefined, i.e. not used. In the least significant byte, bit 2 is not used and bits 4-7 are always the same as bits 0-3.
In write attribute operations all bits except zero, one and three are therefore ignored.
If the object is a directory, bits zero and one are also ignored.
The other bytes and bits are used by other filing systems, e.g. network.

On exit: AL contains the result code as follows:

0 = File not found

1 = File found

2 = Directory found

FFh= Protected file

80186 flags: Undefined

OSRDCH - INT 46h

Function: Read a character from currently selected input stream

On entry: No parameters

On exit: AL contains the character or an error code

80186 flags: CF unset = Valid character read
CF set = Error condition (code in AL)

OSASCI - INT 47h

Function:	Write character to the currently selected output stream but also output character Ah if current character is Dh
On entry:	AL contains the character to be written
Operations:	IF AL = Dh (CHR$13) THEN call INT 48h ELSE call INT 49h
On exit:	No returned information
80186 flags:	Undefined

OSNEWL - INT 48h

Function:	Write carriage return / linefeed to current output stream
On entry:	No parameters
Operations:	Loads AL with Dh (CHR$13) and calls INT 49h then loads AL with Ah (CHR$10) and calls INT 49h
On exit:	No returned information
80186 flags:	Undefined

OSWRCH - INT 49h

Function:	Write a character to the currently selected output stream
On entry:	AL contains the character to be written
On exit:	No returned information
80186 flags:	Undefined

OSWORD - INT 4Ah

Function:	Various functions, with parameters in a memory control block. (See also OSWORD FAh at the end of the chapter).
On entry:	AL contains the OSWORD function. DS:BX points to a control block, the size and organisation of which is call dependent. See 6502 MOS reference texts for full call specifications.

On exit:	Parameters returned are call dependent and if any, are placed in the control block
80186 flags:	Undefined

OSBYTE - INT 4Bh

Function:	Various MOS byte operations
On entry:	AL = OSBYTE type. See 6502 MOS texts for OSBYTE calls BL = 6502 X register parameter BH = 6502 Y register parameter (if required)
On exit:	BL contains the returned 6502 X register value BH contains any returned 6502 Y register value (see note) Note: For OSBYTE calls that do not require a Y parameter, the Tube host code implements a short OSBYTE routine which neither expects nor transfers the value in BH, which therefore may be safely ignored. However, for all OSBYTE calls, if the value in BH is important it should be preserved by the caller. When a short OSBYTE call is used, the Tube host code does not actually return this value and BH is therefore undefined. For other OSBYTE calls the Y register value is returned, but is not meaningful.
80186 flags:	CF value is call dependent but unless significant to the 6502, returned values should be regarded as undefined.

OSCLI - INT 4Ch

Function:	Sends a string to the host's MOS Command Line Interpreter. (* command)
On entry:	DS:BX point to the command string
On exit:	No returned values
80186 flags:	Undefined

Notes: The command string passed does not need a preceeding '*' and may be up to a maximum of 256 characters long.

The string should be terminated by Dh (CHR$13). E.g., setting DS:BX to point to the string `cat<CR>` would produce a catalogue of the currently selected filing system directory of the chosen drive.

Any variable parameters you wish to pass must be entirely included within the command string length. Unrecognised commands will produce an error which will be reported via the Tube register four error

IRQ, ultimately causing a DOS error unless this is trapped by your own code.

Coding of MOS Calls

As a simple example of how the MOS interrupts are called, the following extract of 80186 assembler code shows how OSCLI (interrupt 4Ch) would be called with a fixed string which is the equivalent to '*HELP ADFS' issued in native mode.

```
; some fxed constant declarations
oscli   equ     04Ch    ; OSCLI interrupt
cr      equ     0Dh     ; Carriage return

; the code that does the call
        mov     ds, cs  ; just in case it wasn't
        xor     al, al  ; clear AL to be neat
        mov bx, offset string_1 ; point BX to the string
        int     oscli   ; call INT 04Ch

string_1:
        ds      "help adfs"
        db      CR
```

Block Data Transfer

The OSWORD &FA which was removed from 6502.SYS is still available, but has been moved from the host code to the 80186 monitor, which already provides a slinilar facility for manual entry In TFER. It permits transfers to or from the 512.

The OSWORD interrupt code tests for a value of AL = 0FAh and diverts the call if it matches. This OSWORD call requires a control block to be set up in the 512, the location of which is pointed to by DS. BX. The format of the control block is:

```
0       Number of parameters sent to I/O processor (0Dh or 0Eh)
1       Number or parameters read from I/O processor (01h)
2       LSB of I/O processor address
3       ...
4       ...
5       MSB of I/O processor address
6       LSB of 512 offset address
7       MSB of 512 offset address
8       LSB of 512 segment address
```

```
9       MSB of 512 segment address
A       LSB of length of transfer
B       MSB of length of transfer
C       Operation type (See below)
D       6502 memory access control
```

The operation type specifies the type of transfer as follows:

```
0       Write to  6502 at 24 µs/byte
1       Read from 6502 at 24 µs/byte
2       Write to  6502 at 26 µs/pair of bytes
3       Read from 6502 at 26 µs/pair of bytes
6       Write to  6502 at 10 µs/byte using 256 byte blocks
7       Read from 6502 at 10 µs/byte using 256 byte blocks
```

The memory access control byte allows access to the paged RAMs, paged RAM and shadow RAM in the host machine and is laid out as follows:

7	6	5	4	3	2	1	0
-	sm	m/s	c	pr3	pr2	pr1	pr0

where the bits have the following functions.

```
Bit7    Unused
sm      IF  (I/O address between 3000h and 8000h) AND (sm=SET)
        THEN use screen memory regardless of state of *SHADOW -
overrides bit 5
m/s     0 = Use main memory if screen address specified
        1 = Use shadow memory if screen address specified
C       IF (I/O address between 3000h and 8000h) AND (C=SET)
        THEN use currently selected ROM
pr3-pr0 Paged ROM number binary value 0 to Fh)
```

The memory access byte is only used if the first byte of the control block is set to OEh, otherwise it is ignored. Use of the memory access byte allows paged ROM software to be copied and therefore should be restricted to system use. This would prevent access to the shadow RAM, which is not used by the system and could not be legally accessed by other means.

A small example of the call is now given. This assumes that the control block has been set tip correctly and is located in the first 64k segment. A contiguous 36 kbyte area of memory is used as a buffer for data written from 2000:1000 in the 512. The host buffer starts at 3000h and extends to BFFFh. 3000h to

7FFFh is specified as shadow screen memory and 8000h to BFFFh is specified as paged RAM in bank 5.

```
osword          equ     04Ah
transfer        equ     0FAh
                sub     ax,ax           ; nake ax zero
                mov     ds,ax           ; point DS at segment 0
                mov     bx,offset_transfer_block
                mov     al,transfe      ; set up OSWORD type
                int     osword          ; call interrupt 4Ah

transfer_block:
                db      0Eh             ; number of parameters sent
                db      01h             ; number of parameters read
                dw      3000h,0         ; base address in 6502
                dw      l000h,2000h     ; base address in 512
                dw      9000h           ; length = 36K
                db      6               ; fast 256 byte type 6
transfer
                db      025h            ; use shadow and paged RAM
```

7: DOS Plus

A Short History

Throughout its history the growth of DOS has been something of a chicken and egg event. With the development of 16-bit processors towards the end of the 1970s the DOS era began. DOS was developed directly from an Operating System called DOS-86, itself a direct development of CP/M. In fact so similar were they that programs, files and complete discs could be converted between one and the other by the simple mechanical process of running a conversion program.

The Microsoft Corporation acquired the rights to DOS-86 and further developed it, producing the first version of MSDOS (Microsoft DOS) version 1.0, in 1981. Digital Research were, at the same time, developing CP/M86 as a growth path for its existing CP/M user base and as a competitor to MSDOS, but they suffered delays in development. The intention of both was to offer their Operating System to IBM, who were then producing the first of their new range of PCs, and who intended to offer a choice of several Operating Systems with these machines. MSDOS was one choice and CP/M86 would have been another had it been available in time for the new machine launch.

Due to the hardware design policies practised by IBM, MSDOS actually appeared in a slightly modified form, using customised chips in the hardware to replace some of the software functions of MSDOS. This IBM PC version of DOS was called PCDOS. This was just before the liberated time of new anti-trust legislation in the USA. Previously IBM had largely confounded the competition, preventing them from gaining a foothold in IBM markets by using dedicated hardware to replace some software. This meant, it could be argued, that anyone who produced IBM compatible PCs was actually copying IBM hardware designs, not merely using a similar Operating System.

This theory was one of several eventually challenged in court and, while the conclusion was less than clear, in a practical sense the important point was that IBM did not win. As a result anyone could produce PCs, no matter how similar they might seem to IBM's machinery - so they did. The PC boom had begun, and that assured the future of the original MSDOS as the standard business PC Operating System, with IBM continuing to supply PCDOS. Since then the two have always been developed along the same lines while maintaining their fundamental differences.

At the present time there are literally millions of PCs which use one variant or another of DOS as their Operating System. Had Digital Research been quicker in developing CP/M86, it might very well now hold the position held by MSDOS. They were not and so found it necessary to add a 'front end' DOS emulator to their 86 series Operating Systems to provide compatiblilty with MS and PC DOS. This Operating System became known as DOS Plus. CP/M86 is still used as an Operating System in its own right, but not generally as standard in 86 series machines.

As both MSDOS and PCDOS owed so much of their internal structure and organisation to CP/M, these two versions of DOS and DOS Plus have also been able largely to evolve in parallel, even as far as using similar version numbering. Which is the 'better' Operating System between MSDOS and DOS Plus is, of course, a matter of opinion, but it must be acknowleged that, for any given version, Dos Plus can do all that the equivalent MS or PC DOS can do, while the reverse is not the case.

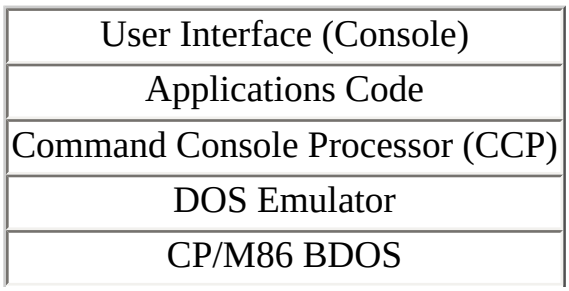
This is because the DOS functions provided by the emulator are similar to those of MS or PC DOS, but DOS Plus can offer all the direct facilities of CP/M86 in addition, and CP/M86 has capabilities that DOS does not. The most glaring example is that the latest versions of MSDOS (at the time of writing, up to about experimental version 4.3) may perhaps just be thinking about multi-user/multi-tasking, while CP/M86 has provided this capability since its inception.

Elements of DOS Plus

The basic design concepts of the original CP/M-80 and all succeeding versions of it (and DOS) are that the kernel of the Operating System is insulated from the outside world by a quite separate set of software interfaces. While the outward facing portions of these interfaces might alter for installation in different hardware, the inward connections to the DOS kernel, the BDOS (Basic Disc Operating System), and hence the applications software, do not.

The benefits of this approach are three-fold. First, within the limitations of memory size and range of peripherals, the way facilities are accessed by programs does not change, regardless of the type of machine. Secondly, from the point of view of a user, apart from physical differences, one DOS system should appear to be much like another. Finally, and equally important, the BDOS can be extended by adding to or altering its hardware interfaces, (for example for new storage media requiring new controls, like the read/write laser discs now beginning to appear commercially), without disturbing the kernel, the data it provides or uses, or the existing applications interfaces.

Although the main elements of DOS Plus in the 512 are similar to other systems, some differences are forced by the two-processor Acorn hardware configuration. The XIOS (eXtended Input Output System) is known in conventional systems as the BIOS (Basic Input Output System). The name change for the 512 is because the direct peripheral hardware interfaces are replaced by the indirect Tube protocols discussed previously. However, the concepts are the same and it is this very design philosophy which permits an eight bit BBC Micro host to run a 16-bit 80186-based DOS co-processor. A simplified schematic of this 'layered' approach is shown below in Figure 7.1.



XIOS
The Tube
6502 MOS and Filing System

Figure 7.1 The 512 DOS Plus Schematic

In practice the divisions between the various layers are not so dear cut. For example, an application may replace the original Console Command Processor entirely with routines of its own. Alternatively, by means of INT 224, Dos Plus applications can call the BDOS directly, bypassing the CCP and the DOS emulator. However, these operations are legal and cause no problems because the BDOS and the XIOS still stand between the application and the hardware.

Finally, and as is known only too well by 512 users, some applications entirely ignore all standards and short-circuit the system by attempting direct access to the hardware peripheral controllers. To work, this would need to be able to by-pass the DOS emulator, the BDOS, the XIOS, the Tube and the host's MOS or filing system. Obviously such programs have no chance of success in the 512.

The BDOS

The core of the system, both figuratively and literally, is the BDOS. This should be functionally invariant for all versions of DOS Plus. Its functional responsibilities are completely hardware independent, comprising memory management, logical character input/output, logical file and record management and timer interrupt control.

Programs can access BDOS functions directly via interrupt 224, reserved by Intel for Digital Research Operating Systems. The technique involves loading specified processor registers with parameters and calling the BDOS by an INT 224 instruction. System calls are documented in Appendices A-D.

The DOS Emulator

The DOS emulator in DOS Plus 2.1 imitates PCDOS 2.11. Specifically PC-DOS interrupts between 21h and 27h inclusive are emulated by being converted to CP/M functions, which are executed via calls to the BDOS.

It should be noted that PCDOS 2.11 includes numerous other DOS interrupt calls than this restricted range, but these are not 'officially' supported in DOS Plus (although many actually are implemented), and hence may perform different functions to the PC calls. Note however that IBM have subverted many of these calls for their own purposes, and frequently these PC interrupts are not compatible with MSDOS or the processor chip manufacturer's specifications.

In fact, IBM have used several interrupt numbers which were specifically reserved by Intel (the 86 series chip manufacturer) for future MSDOS expansion, and even some reserved by Intel for themselves for future processor development. For example, In the IBM PC/AT, interrupt 5, specified by Intel as an 86 series hardware interrupt bounds error), actually precipitates a screen print. This example (and there are others) perhaps adequately underlines how compatible some 'compatibles' and their Operating Systems really are and how imprecise this subject is.

The Console Command Processor

The CCP consists of two parts. The first, the command loader is transparent to the user and is loaded with the BDOS and DOS emulator when the system is booted. It is responsible for allocating sufficient memory for COMMAND.COM which is the second part of the CCP to be loaded, and which is located immediately above DOS Plus in low memory. The command loader maintains a second copy of COMMAND.COM in high memory and, as one of its secondary functions, if ensures that the user's copy of COMMAND.COM is intact on program termination. This it achieves by comparing the high memory and low memory copies. If they are not the same one or other has been corrupted.

If COMMAND.COM has been corrupted the command loader will always replace both copies by loading a new image from disc automatically, since it cannot be sure which of the two was damaged. The user may become aware of the command loader when this situation occurs and COMMAND.COM is not available on the current drive. It is the command loader that generates the error message cannot find COMMAND.COM, or can't load COMMAND.COM, depending on the version of DOS Plus (1.2 or 2.1) in use.

The command loader is also responsible for processing CTRL-C aborts during program execution (since COMMAND.COM may no longer be resident) and tidy program closedown during abort or normal termination.

COMMAND.COM is the transient portion of the CCP that provides the user's interface to DOS. A maximum command line length of 127 characters is permitted. It is responsible for reading user input and translating it into a meaningful action in one of two ways.

It can be either by means of passing the command direct to DOS if the command is recognised as a permanent function, when it will be executed by calls to BDOS functions, or by loading the appropriate transient utility and passing the parameters to it if there is no such permanent command. In this latter case the commands may be carried out by calls to DOS interrupts or BDOS functions, depending how the code was written. Such transient file names are recognised by the command name supplied in the instruction and one of the allowable executable filename extensions detailed below. In this case, if no such file can be found in any current path, COMMAND.COM responds by outputting the `Bad command or file name` message.

The transient portion of the CCP is also responsible for opening and reading the file and performing a

similar command translation function when a batch file is executed.

The XIOS

The XIOS (or BIOS in other systems) is responsible for communication between the BDOS and the computer hardware. It presents a fixed interface to the BDOS for the passing of peripheral commands as a result of program activity, but is customised by the computer manufacturer to provide the correct instructions to program the particular hardware device drivers and controllers being used in each machine.

In the 512 the task of the XIOS is precisely this, but instead of direct control of the hardware, the XIOS is responsible for translating BDOS commands into instructions which are passed through the Tube to the host, or receiving the replies, as documented in the preceding chapters on the Tube host code and 6502.SYS.

Because of the Tube interface, there is additional code in the XIOS that is directly responsible for initiating or receiving data transmission. While direct hardware triggered events and interrupts normally occur in a BIOS system, in the Tube these must be simulated by recognising the Tube protocols that generate IRQs and NMIs. However, in concept these are the equivalent of hardware interrupts and fulfil the same purpose.

The XIOS code itself is transparent, although special calls are provided so that applications can directly access the Tube software by means of the reserved function 50h within INT 224.

Executable Files

All versions of DOS recognise and reserve three file extensions for their own use, while DOS Plus includes two more. In all cases these extensions tell DOS that the file contains executable instructions in one form or another. 'COM', 'EXE' and 'BAT' extensions are common to all versions of MSDOS, PCDOS and DOS Plus. In addition, DOS Plus also recognises 'CMD' and 'RSX' as executable filename extensions.

Naturally, these filename extensions should not be used for any other type of file than those detailed below or the result of entering the filename as a command may, with the exception of 'EXE' files, be unpredictable and, in most cases, disastrous.

BAT Files

A file extension of .BAT (meaning a batch file) indicates that the file contains pure text which, line by line, can be processed by COMMAND.COM in a similar way to keyboard-entered instructions. That the instructions are automatically presented in a predefined sequence gives DOS a simple job control

language. Batch files originated in CP/M-80.

BAT files are opened for input only during normal execution i.e. you cannot update them or record data in them while they execute. Each text line up to the maximum command line length, or to the first occurrence of Dh (CHR$13), is treated as a single command, though each line may be conditioned and, by the use of labels, the file may effectively contain an alternative to the conditional command.

Unlike keyboard entry, where every command line is isolated, instructions in a BAT file have a logical position, or line number in the file. This permits a simple language structure by means of labels, the line number of which is remembered during execution. A GOTO command, therefore, instructs COMMAND.COM to reposition the file pointer to a given line number and resume execution from that point.

COM Files

A .COM file extension (in DOS, short for 'command') indicates that the file contains machine code (possibly with embedded data) which will occupy not more than 255 pages (i.e., one page less than a complete segment), and which should be loaded as a complete program in the first available free memory block large enough to contain it.

Note that there is no check implicit in the COM filename extension. That is to say, the file is not checked to ensure that it contains executable code, DOS simply loads the file and attempts to execute it immediately, which is why the extension should not be used for any other file type.

On initial load into RAM, a program segment is allocated to the file, the first page of which is automatically set up by DOS as the program segment prefix (PSP) at load time, that is, it is not stored as part of the COM file on disc. This area is initialised to contain certain system data in the first half, while the second hall is reserved to hold the original command line parameters (the command tail) if any were issued with the original command that called the program.

The purpose of this technique is that the complete uncorrupted command line (excluding the program's own name, i.e., parameters only) remains available to the code throughout execution, regardless of what might happen to the contents of RAM elsewhere in the machine.

Since they have no header information (e.g., containing an execution address), initial entry to COM programs is pre-defined at address 100h, the first byte after the PSP when loaded, the first actual byte of the file as stored on disc. COM files must therefore, always contain an executable instruction as their first byte. Commonly, immediate program jumps transfer execution to another address within the code segment if it is required to set up tables etc at 100h, which can be subsequently overwritten. The first 128 bytes of the PSP should never be corrupted, though what the program does with the command line parameters during execution is its own affair.

On initial entry to the code, all segment registers point to the code segment and the stack pointer is initialised to FFFEh (the segment size minus two), or to the address of the top of available memory minus two if less than a complete segment is available. This defines the first available stack address, which therefore will always be at the top of the program area unless the stack pointer is repositioned within the code. Immediately after loading the program the first two bytes of the default stack are already used by DOS, which stores a zero word there. It is, therefore, also implicit in COM files that at least two bytes of stack will be available in the code segment.

COM files are designed to be simple to produce with a minimum of 'red tape', permitting the programmer to produce a fast, efficient program. COM files are, therefore, commonly used for small utilities required to perform a limited range of functions.

A map of the segment and PSP for a loaded COM file is shown below.

Stack grows downwards	CS:FFFFh
Program code and data	CS:0100h
Original Command Line Tail	CS:00FFh CS:0081h
Command Line Length in bytes	CS:0080h
File control block areas	CS:007Fh CS:0055h
Unused	CS:0053-54h
Code for INT 21h, RETF	CS:0052h CS:0050h
DOS work area	CS:004Fh CS:002Eh
Environment string address	CS:002Dh CS:002Ch
DOS work area	CS:002Bh CS:0016h
Critical Error Handler Address	CS:0015h CS:0012h
The Previous CTRL-BREAK address	CS:0011h CS:000Eh
Copy of the terminate address (Usually in 'COMMAND.COM')	CS:000Dh CS:000Ah
Long Call to (INT 21h)	CS:0009h CS:0005h

Segment address of HIMEM	CS:0004h CS:0002h
INT 20h instruction (terminate)	CS:0001h CS:0000h

Figure 7.2 The COM File PSP

Spawning

The PSP terminate address in 000Ah to 000Dh usually points to the code in COMMAND.COM which handles tidy program closedown, but a program can change this if required.

The pointer in the PSP is set up before execution commences but may be temporarily saved (e.g. on the stack) then the PSP copy of it may be changed to point to a re-entry in its own code. This permits one program to call a second or subsequent program. As each subsequent program is loaded, its own terminate address is taken from the calling routine's PSP copy. If the caller has set up its own return address in its PSP terminate address, when the second (or n^{th}) program terminates it automatically returns control to the caller, which in turn can lump back to its own caller, and so on until finally the last in the reverse chain (i.e. the first loaded) restores the original termination address and exits by jumping back to COMMAND.COM.

In essence, this is the DOS technique known as spawning. It may help to remember that all COM programs are actually spawned by COMMAND.COM.

The technique permits one, or a number of programs to be called from within any other, while each calling program remains resident and in control for the duration of its execution. This is why the terminate address is not 'fixed' externally by DOS, but is made available to user code- The only practical limits on how many programs may be spawned is the maximum memory available and good sense.

It should be noted that, although the terminate-handler address has been used here for illustration, employing the technique of spawning necessitates that the both the CTRL-C and the critical error handlers' addresses be manipulated in a similar fashion. Failure to do so will result in (some) undefined vectors if a spawned program terminates unexpectedly (See Chapter Eight).

EXE Files

A DOS .EXE file (meaning simply, executable) also contains program code, but each EXE file has a header which identifies the file completely, prior to execution. EXE programs do not necessarily commence execution at 0100h, nor are they limited to a single segment in size and can use memory up to the total available if required. For this reason the header must define the file as an EXE (and nothing else) and include additional information about where in memory the program is to be executed from,

how much memory will be needed, if it is overlaid and so on.

For this reason an additional feature, a file checksum, is included. If the checksum is incorrect, DOS will report the fact and the program will not be executed, since the result could be catastrophic. Initially the file is brought into memory immediately above the PSP (which contains the same information as for COM files above), but the code may then be moved prior to execution.

The size and type of information contained in an EXE header depends on how big the program is and how much of it is to be automatically relocated at run tune, but the header size is always a multiple of 512 bytes. Another difference from COM files is that the processor registers have values pre-assigned to them, which will be set up before execution commences. This is because code, data and stack are presented to the loader as separate modules, and each might not be placed in the same segment. This is one reason why EXE ales tend to take longer to load, even when fairly small. The information in the header table is shown below, as it appears in an EXE load module (i.e. an .EXE file).

Byte Offset	Contains
0000h	First part of EXE file signature (4Dh)
0001h	Second part of EXE file signature (5Ah)
0002h	Length of file MOD 512
0004h	Size of file in 512 byte pages including header
0006h	Number of relocation table items
0008h	Size of header in paragraphs
000Ah	Minimum number of paragraphs needed above program
000Ch	Minimum number of paragraphs desired above program
000Eh	Segment displacement of stack module §
0010h	Contents of SP at entry
0012h	Word checksum
0014h	Contents of IP at entry
0016h	Segment displacement of code module §§
0018h	Offset of first relocation object in file
001Ah	Overlay number (Zero for resident module)
001Bh	Start of variable space for relocation table

§ The stack segment is variable if
§§ Program and data segments are variable

Figure 7.3 The EXE Load Module Layout

Before DOS transfers control to the loaded code, the header information is evaluated and, if the checksum is correct, the CS and IP register values are calculated from the entry point in the header (initial IP at 0014h) and the physical load address. The DS and ES registers are set to point to the PSP so that the program can readily access the environment pointer, the command line tail and so on. The SS and SP registers are likewise set to predefined values, indicated by a STACK declaration within the source code and decided by the programmer. The stack space can contain predefined data, or may be left indeterminate prior to execution. The program is then entered at the location pointed to by CS:IP.

The layout of an EXE file in memory is shown below in Figure 7.4, as it is just prior to execution. Note that the header has been discarded, as its purpose has been served. It does not form part of the memory resident program.

Stack segment stack grows downwards from top of stack segment	`SS:SP`
Data seqment	`SS:000h`
Code segment	`CS:0000h`
Program code (IP points somewhere in here)	
PSP	`DS:0000h ES:0000h`

Figure 7.4 An EXE Program in Memory

An EXE file is usually produced from a number of object modules, each of which has an .OBJ extension. OBJ files are essentially COM files with built in 'hooks' in a known format that allows them to be joined to other OBJ files. The normal production process involves writing and assembling or compiling several separate object modules, which are subsequently linked together (by a linker!), often with standard library routines added, to produce the final EXE file.

Many language compilers also produce OBJ modules that likewise require linking, though in some implementations this secondary process may be transparent. In practice, an EXE file may well be built from a mixture of assembled code, user library routines, compiled modules (perhaps from more than one language) and compiler supplied standard library routines. Library routines are ready made object modules that provide subroutines for common program functions, such as input/output or mathematical functions.

CMD Files

CMD files were the original CP/M predecessors to DOS's COM files, and likewise CMD is short for command. CP/M80 CMD files bequeathed most of their other characteristics to COM files too, including the one segment limit and the fact that they are not automatically relocatable. However, in CP/M86 extensions were made to the CP/M80 CMD definition, which allowed code and data to be separated so they could occupy different segments with automatic register initialisation, much like DOS EXE files.

The version of the CMD file produced depends on the way the code is written and the facilities required. In CP/M terminology generally, each separate area of the internals of a program is known as a 'group', thus the program code forms the code group, regardless of its size. The data for a program is the data group, the stack segment, if in a different segment is the stack group and so on.

In CP/M86 four categories of CMD file exist:

- The 8080 model
- The small memory model
- The compact model
- The large memory model

In the 8080 model (the original from CP/M80) the code and data groups are mixed, much as in a DOS COM file. However, segment registers can be manipulated within the code so as to access other segments, therefore the program is not, of necessity, limited to 64K overall, though there is no automatic distinction between code and data during initial program load.

The small model separates the code group and data group into different, though not necessarily single, segments, which can be placed separately in memory with automatic CS and DS initialisation on loading. Again, any additional segments in each group must be managed within the program code.

The compact model is used when an extra segment (ES) or a stack segment (SS) is also required. In this case CS, DS, ES and SS are automatically initialised. In effect, it provides the capability for a single code group but multiple data groups. As before, within each group, multiple segments can be managed by the program.

The large memory model permits programs with multiple groups for both code and data with, for all practical purposes, no limit on program size. All appropriate segment registers are initialised on program load. Memory permitting, each code group and each data group can be up to one megabyte in size.

The major difference between a large memory CMD file and an EXE file is that DOS assumes all code and all data segments occupy a single contiguous block of memory, while CP/M permits its various groups to be located anywhere in memory. This allows multi-user applications, when one or more groups of code can each have several, completely separate groups of data. In this case several users can each have several (usually up to four) of their own independent applications, each with separate data,

while CP/M86 requires only one copy of each code group to run them all. In other words the CP/M86 large memory model is, by design, a multi-user, and multi-tasking system.

For all types of CMD file, the system initialises a 96-byte stack, with the far return (termination jump) already placed on it. In addition, the base page (00h to FFh) of the loaded program is initialised in a way which depends on the memory model indicated in the file header.

The following list, Figure 7.5, shows how the base page is set up on program load.

Byte Offset	Name	Contains
0000- 0002h	CL	Code length (the size of the code group)
0003- 0004h	CB	The base paragraph address of the code group
0005	M80	A flag = 1 for the 8080 model, 0 otherwise
0006- 0008h	DL	Data length (the site of data group)
0009- 000Ah	DB	The base paragraph of the data group
000Bh	EL	Reserved
000C- 000Ah	EB	Extra data length (the size of the extra group)
000F- 0010h		The base paragraph of the extra group
0011h		Reserved
0012- 0014h	SL	Stack length (the size of the stack group)
0015- 0016h	SB	The base paragraph of the stack group
0017h		Reserved
0018- 0029h	AUX	The area containing the four optional independent groups for programs that execute using the compact memory model. Each is stored as a 3 byte length plus a two byte base paragraph address.
0030- 003Fh		Reserved
0050h	DR	The drive from while the program was loaded, 0 for the default drive, 1 - 16 for drives A through P
0051- 0052h	PWA1	The DS offset of the password field of the first command operand in the DMA buffer (see 0080h)
0053h	PWL1	The length of the first password field

0054- 0055h	PWA2	The DS offset of the password field of the second command operand in the DMA buffer (See 0080h)
0056h	PWL2	The length of the second password field
0057- 005Bh		Reserved
005C- 006Bh	FCB1	The default file control block (FCB1) for the first command operand (if there is one)
006C- 007Bh	FCB2	The default file control block (FCB2) for the second command operand (if there is one)
007Ch	CR	The current record position for FCB1
007Dh- 007Fh	RRN1	The optional random record position for FCB1
0080- 00FFh	DMA	The DMA buffer. Initially this holds the complete command tail

Fig. 7.5 Base Page Command Allocation for a Specimen CMD File.

RSX Files

Resident system-extension files have an .RSX filename extension. These are programs which, when executed, will automatically attach themselves to the Operating System, and can extend the function of the system by intercepting BDOS calls. The RSX can then either process the calls itself, thus modifying the BDOS function, or pass the call to the BDOS unchanged. More than one RSX can be in operation at any time, and a normal external program can also make direct calls to RSXs for special functions without issuing calls through the BDOS intercept.

RSX routines should not be confused with background programs, (e.g. PRINT.CMD and ALARM. CMD), nor with DOS Plus 'sticky memory' programs which are, or can be, the equivalent to MSDOS 'Terminate and Stay Resident' (TSR) programs, but which need not necessarily have anything to do with BDOS calls. RSX programs may be called by CMD programs and they may use sticky memory calls, but the programs issuing such calls are not RSXs, and programs using sticky memory may be, but are not necessarily RSXs.

Apart from the programming techniques employed, RSX programs are prepared in a different way to normal CMD files. The program is first assembled or compiled as a CMD file, then a special utility, GENRSX is required to correctly set up the RSX prefix. The RSX prefix, therefore, permanently precedes the program image in the resulting disc file. The data format of an RSX file prefix is shown below (Figure 7.6).

Byte Offset	Contains
0000h-0002h	Jump to the start of the RSX program code
0003h	The terminate flag - if not zero the RSX terminates when the calling program terminates. RSXs belonging to background programs are always removed when the background program terminates
0004h-0007h	The address of the next RSX in the chain
0008h-000Fh	The name of the RSX
0010h-0011h	The base page segment for the RSX
0012h-001Fh	Reserved
0020h on...	The RSX code

Figure 7.6 Data Format of an RSX Pile Prefix

Sticky Memory

Sticky memory describes the BDOS ability to allow a program to allocate memory permanently and exclusively for its own use. For the time it remains active only the program itself can free the sticky memory. By varying the memory allocation call type the program can define that the memory should be released on termination but, if it does not do so, the memory allocation remains unavailable to other programs when the sticky memory program terminates. When terminated, a program's sticky memory can be de-allocated by another program, but only if the precise start address and size of the sticky memory area is known.

The purpose of sticky memory is to allow a program to claim and reserve more memory than its immediate code and data needs justify. This is required when, for example, a program will dynamically build tables which may be small or non-existent initially, but for which expansion must be possible, as and when required. If memory could not be reserved in this way, loading a second program would allocate the area immediately following the top of the first program's initial memory allocation, which would then prevent such expansion. By allocating sticky memory, the reserved space cannot be used by a subsequently loaded program.

A common use of sticky memory is for interrupt driven routines which need to remain in memory (i.e. like MSDOS TSR pop-ups) and operate from time to time dependent on certain conditions. This facility is also used by the manual ADDMEM and COMSIZE commands.

Background Tasks

Some programs may need to carry out functions over a period of time, which would prevent any other

program running if they were to remain as a normal external foreground routine. Two examples are provided with the 512, ALARM and PRINT, both of which clearly would be pointless if they prevented other foreground programs from running simultaneously. DOS Plus permits up to three such programs to enter the background environment at anyone time.

Programs can enter the background by detaching from the console device by a call to INT 224 with a value of 147 in CL. After this call console I/O is impossible, but the program will continue to execute. Only CMD programs can enter the background and, once this is done, only direct BDOS calls are supported and DOS interrupts are no longer available.

If coded suitably, communication is possible with background programs from another program, so they can be terminated or otherwise commanded, but this is not possible directly from the console. Alternatively, like PRINT and ALARM, they can automatically terminate themselves when their function is complete, freeing their memory again for other programs.

Notes

Quite often, as in the PRINT utility, several of these techniques may be used in conjunction to provide a user service. There is (or was) a PRINT.RSX which is now transparent to the user, since it loads by default. In its idle state it requires no additional memory other than a small amount for its permanent code, which is effectively hidden within DOS Plus. When some print action is to take place, the user communicates with the RSX by indirect means, using the external program PRINT.CMD as an intermediary. In this case, PRINT.CMD passes on the user instructions to the RSX, then terminates. The print RSX then allocates the required sticky memory, opens the file and outputs it to the printer. On completion, the extra memory is freed and the system returns to its 'at rest' condition.

It is easy to demonstrate that the initial PRINT.CMD program actually does terminate, leaving the RSX to carry out the required functions, simply by issuing a command to query the progress of a user print instruction by means of the command PRINT. It will be observed that the immediate effect is that PRINT.CMD must be freshly loaded from disc to deal with the user's console communication. PRINT. CMD then interrogates the RSX, displays the state of the print queue (since RSXs cannot produce console output either), and once again terminates.

Except as mentioned above for RSXs and background tasks, all types of executable program except batch files, that is COM, CMD and EXE files, can issue all varieties of Operating System or hardware calls by means of DOS interrupts, BDOS calls or calls to the XIOS.

In general, direct XIOS calls are more efficient than BDOS calls, which in turn are more efficient than DOS interrupts since the overhead of the DOS emulator is avoided by both, while direct MOS calls also avoid the BDOS. On the other hand, many DOS interrupts are easier to code, since BDOS and XIOS calls, by definition, operate at succeedingly lower levels. In practical terms however, there will rarely be any discernible difference in execution time even for compiled code, much less for hand-coded

assembler, regardless of the method chosen for calling Operating System functions.

8: DOS Plus Interrupts

Introduction

Many users will find that an intimate knowledge of DOS interrupts is not needed if use is confined to ready written applications or many programming languages. For users who write their own programs in one of the many interpreted or compiled high-level languages, such common facilities as will be required for most program operations will be provided either in the form of single instructions, or library subroutine calls provided with the language package.

For example, the PRINT statement in BASIC is a single high level command, capable of processing several different types of data output in a convenient pre-packed instruction. In languages like C, the compiler authors will usually have provided similar facilities in ready-made libraries (Inc such as stdio, short for 'standard input/output').

Clearly, however, no matter how capable a language implementation is, there will always be a large number of system facilities which cannot be catered for in such a way. Many languages provide a means of adding to the routines held in libraries, or permit the embedding or calling of sections of machine code within programs to solve this type of need. Whatever the means of providing such facilities, the prime requisite for direct access to DOS interrupts involves the use of machine code at some point if legal coding techniques are to be employed.

The remainder of this chapter broadly outlines the philosophy of DOS system interrupts, which share many similarities with 6502 host MOS calls, but also include some considerable differences.

To those familiar with 6502 machine code, the concept of direct or indirect calls to Operating System or filing system functions will be well known. However, in the host there is a wide range of different call categories, the type and implementation of which is often governed by the operation required. In addition the filing system is a separate entity in the host, which also has numerous separate calls, again with their own rules.

For example, certain call types, like OSBYTE provide a large number of different general purpose facilities, but these are called in a different way to OSWORD calls, which are also, to a degree, general purpose. In addition there are numerous single function calls such as OSNEWL, OSRDCH and so on, each of which has a specific purpose and call all of which adds to the general complexity of 6502 assembler coding. Filing system operations involve yet more call types, again with several different calls and new sets of rules. The range of differences also extends to the method of interfacing to such routines by user code, since some calls are vectored and may be intercepted for modification, while others are not and may not.

In DOS and DOS Plus, in large part because so much more memory is available, the situation is much more straightforward and better organised. Although the information used or returned by individual Operating System calls naturally differs from one to another, there is only one type of call in the system. Happily in this case the Operating System also includes the filing system and the same type of call, therefore, also caters for all filing system requirements too.

As a final bonus, all the calls are vectored and the system provides ready-made facilities to intercept or redirect these vectors. Naturally, these operations themselves are also carried out by means of the same type of system call, the interrupt, each of which is simply identified by a number.

Confusion may arise for the 6502 programmer because the meaning of the term interrupt differs slightly in DOS compared with its meaning in the host. Whereas in the BBC host an interrupt is normally concerned with the MOS servicing of hardware devices, actions being triggered by a timer or hardware activity, in DOS the term is used for all types of direct or indirect requested OS activity, however caused, including explicit calls from within user code.

One further major difference from the host is that, in 86 series systems several simultaneous interrupts can be permitted and several types of NMI also exist, as opposed to the host's single NMI. However precise details of how these are handled is not required by the user, since the 'nesting' and prioritising of interrupts is performed transparently within DOS.

When an interrupt occurs the preparatory actions are much like those in the BBC host. Processor registers are saved and the return address is pushed on the stack. As each interrupt is allocated a priority category, all interrupts of an equal or lower category are masked out. However, higher priority interrupts are still permitted, therefore, if one of these should occur, the higher priority interrupt will interrupt an earlier one of lower priority.

As mentioned above, the very much larger memory available to DOS systems permits the luxury of, compared to 6502 systems, effectively unlimited stack space. This in turn permits a hierarchical interrupt philosophy. Suffice it to say that, if one interrupt should need to cut across the processing of another, there is sufficient space to permit all processor registers and data to be saved from the first interrupt in the usual way, so that its processing can resume normally after that of the higher priority interrupt is complete.

Essentially interrupts fall into one of three categories. Though the action taken within DOS follows a similar path regardless of the source and type of interrupt.

Internal Hardware Interrupts

These are generated by certain unexpected events encountered by the processor during program execution, such as attempting to divide by internal hardware interrupts are, perhaps, most easily related

to the BRK handling routines of the host, as both represent an irrecoverable error.

The assignment of such hardware events to these associated interrupt numbers is effectively 'hard-wired' into the processor chip and this association is not normally modifiable by any technique. So these interrupts cannot be prevented or avoided in a PC, and should not be suppressed or interfered with in the 512 by intercept code.

External Hardware Interrupts

These are interrupts generated by peripheral hardware activity, most easily thought of like the actions performed by the host processor during IRQ1V processing, for example, because of a keypress. In a true PC these interrupts are triggered by a physical device controller signalling the need for service, the availability of returned data or the completion of an action. In machines so equipped, the 8087 maths co-processor is another source of external interrupts.

External hardware interrupts can be tied to either the processor's non-maskable interrupt pin, or to the maskable interrupt pin, depending on the type and cause. The NMI line is usually reserved for catastrophic events that must be processed instantly, such as a RAM parity error. Maskable interrupts are, of course, similar in nature to those in the BBC host and are, by definition, of a lower priority. The assignment of the interrupt priority level is defined by the system designer and the processor chip manufacturer and cannot be altered subsequently.

In the 512 system all peripheral hardware activity takes place in the host, therefore all such external device interrupts are simulated in the XIOS as a result of Tube transfer activity. Whereas these interrupt connections are normally physically 'hard-wired' in PCs, in the 512 this is not so and certain facilities therefore cannot be implemented.

Perhaps the most frustrating example is that, in a PC, pressing CTRL-BREAK causes the keyboard controller to perform a hard wired jump to the hardware reset routine (an NMI). This often permits escape from a hung program while, in the 512, the only recourse is to perform a complete cold start reboot of the system from scratch, a much more long winded and annoying process.

Software Interrupts

These interrupts are generated by user code (or DOS itself) by issuing a direct interrupt call to request that an Operating System function be carried out. Direct user-called interrupts are generally of the lowest priority, though these interrupts may cause higher priority interrupts as a result of the action of the hardware function called.

Software interrupts are provided by a range of interrupt numbers, but it should be noted that these numbers are not significant and imply no priority. The interrupt numbers are simply a convention, which

are in no way hard-wired into the processor. All software interrupts are, therefore, always maskable.

The Vector Table

The bottom 1024 bytes of memory (0000:0000 to 0000:03FF) are reserved in DOS systems for the interrupt vector table. Each vector entry consists of four bytes, the first two for the segment address, the second two for the offset, each of which is stored in the familiar low-byte-high-byte format. The offset address of the vector entry for any particular interrupt can, therefore, easily be found by multiplying the interrupt number by four.

As 1024 bytes are available and each entry takes four bytes, there is provision for up to 256 vectors and each can, though does not always, represent an interrupt. In all versions of DOS to date, however, less than a quarter of these locations have been used.

In the early days of DOS design, groups or blocks of vectors were pre allocated, for various purposes, to permit individual developments in chip design, micro hardware and DOS facilities by the various parties while theoretically avoiding conflict between them. In practice the situation is not quite so clear since, as mentioned earlier, IBM, for example; have chosen to ignore these standards in several of their PCs, leading to some of the incompatibilities which appear between different types of hardware.

Officially the blocks of interrupt numbers are grouped as follows.

00h to 0Fh

These are the internal hardware interrupt numbers, effectively allocated to the processor chip. All of these interrupts are generated by hardware events and, although the vector entry can be physically modified to intercept the event, such hardware events should not be prevented from jumping to the correct address of their associated vector. For users who write general applications programs, therefore, these vectors should be left alone.

10h to 1Fh

These are the external hardware interrupts that are available for the computer hardware manufacturer. They direct execution to the BIOS (or XIOS) where the system supplier should implement the code, which varies according to the hardware devices and their controllers supplied with each different system.

20h Onwards

20h upwards are the software interrupts. 20h to 3Fh and E1h to FDh are officially reserved for current MSDOS or future developments by Microsoft Corporation. 60h to 67h are nominally reserved as user vectors and E0h is for Digital Research systems.

In summary interrupt numbers are officially allocated as shown in the table below.

Internal hardware interrupts

00h-04h Processor interrupts - all versions
05h-0Dh Processor interrupts - 80286 only
0Eh-0Fh Reserved by Intel for processor expansion

External hardware interrupts:

10h to 1Fh Peripheral and firmware functions

Software interrupts:

20h-3Fh Allocated to MSDOS
60h-67h Allocated as user vectors
E0h Allocated to Digital Research (CP/M)
E1h to FDh Reserved for Microsoft expansion.

Table 8.1 Allocation of Interrupt Numbers.

Officially all other interrupts should be unused though, as already mentioned, this cannot necessarily be relied upon, especially in IBM PC software. In some cases the contents of interrupt vectors do not point to executable code, but rather to data tables, often for graphics or keyboard Character definitions and the like. This particularly applies to software written for the PC Junior, IBM's (relatively) unsuccessful home computer, giving yet more compatibility problems.

Remember that in the 512, interrupts 40h to 4Ch are allocated to the host MOS calls, while FEh and FFh have also been pressed into service. The actions of these interrupts are naturally peculiar to the 512's Dos Plus implementation and no DOS software from any source (except programs specific to the 512) will be aware of them.

Note also that, although INTs 60h to 67h are allocated as user interrupts, in fact INT 67h is used in later PCs as the Extended Memory management System (EMS) interrupt, though this is not relevant to the 512.

512 DOS Emulation

Although officially DOS Plus supports only interrupts between 21h and 27h inclusive to the level of PC-DOS 2.11, in fact, numerous others have been implemented to varying degrees of completeness.

Some of these, as mentioned above, emulate in software the hardware interrupts of a PC, particularly those relating to internal interrupts and relevant external hardware devices. In most cases this technique is fairly successful but, where hardware interrupt emulation is less than complete, problems occur in applications software. Most frequently this relates to keyboard or disc functions Yet others are implemented in such a way that they simply jump to an immediate return with no action. This is done so that the interrupt may be called by an application without causing a program crash, which would be the result of a zero vector entry.

DOS interrupts are called by the assembly code instruction INT followed by the interrupt number, which causes the processor to prepare for an interrupt service request. This involves saving processor registers on the stack, setting up the return address and transferring execution to the code pointed to by the appropriate vector- For example, to call INT 21h, the most frequently used interrupt, the assembly code source statement would be literally:

```
int 21h
```

Clearly, as it stands this does not convey any data. In fact, it merely directs execution to the requisite section of DOS code via the vector table, which initially merely allocates the interrupt priority and masks other interrupts of equal or lower priority. It is therefore necessary for the caller to supply additional parameters to many interrupt calls, usually in the form of the contents of certain of the 80186's registers, but sometimes with additional data held in memory.

Much like host OSBYTE calls, some interrupt calls require only parameter values in the processor's registers and these are then used immediately. For other calls the registers are also used as pointers to memory blocks, which the programmer must set aside to hold call parameters, data or results, more like a host OSWORD call. In a few cases interrupts are highly specific, requiring no entry parameters, with some also returning no values. These highly specific interrupts are more like such host MOS functions as OSNEWL As can be seen, DOS interrupts naturally share many conceptual similarities with host MOS calls, but unlike host MOS calls, for all interrupts there is only one call mechanism.

The MS/PCDOS 2.11 compatible interrupt functions and codes available in the 512's DOS Plus 2.11 are listed in Appendix A. They are shown first in numeric sequence, the order usually required when disassembling or debugging code, followed by a list grouped by function type, the sequence generally required when writing new code. The two lists are followed by the detailed call list, showing the definition of each call's entry requirements and any returned values or conditions.

INT 21h - The Function Dispatcher

Interrupt 21h (also often referred to as the general function dispatcher) provides most of the facilities likely to be used in general applications code, and has a wide range of functions. To identify the particular operation required, therefore, in every call to INT 21h a function code must be supplied. This is always placed in the general-purpose byte register AH.

The function code is directly analogous to the call number placed in the 6502 accumulator prior to a host OSBYTE call. Also, like OSBYTE calls, certain function types require one or more additional processor registers to be set up to contain parameters to the call. However, unlike host OSBYTE calls, in addition a number of INT 21h functions, particularly those relating to disc and memory operations, require parameter or data blocks to be set up in RAM, most commonly with DS:DX holding the segment:offset pointers to the address of the block.

Unfortunately, as DOS has been enhanced in successive versions, new functions have been added in a manner that results in the INT 21h function codes appearing to have been allocated at random. The result of this is that functions are not logically grouped into operational categories and there is no easy way to deduce, from the function number, what the operation type might be - whole disc, directory, file and record handling calls are intermixed with memory/process management and the other general input/output functions in a more or less chaotic sequence.

INT 22h - The Terminate Handler

INT 22h has a single function, but this interrupt should never be called directly by user applications code. The vector entry at 0000:0088h to 0000:008Bh contains the address of the routine entered when a program terminates by means of a call to INT 21h (function 00h, 31h or 4Ch) or INT 27h.

The address of this vector is automatically copied into locations 0Ah to 0Dh in the PSP when a program is loaded, but before execution commences. The contents of this vector are, therefore, immediately available to each program, which may then modify the original vector for the purpose of spawning.

The original terminate address is already stored in the PSP, and a new address can be set up in the vector to point back to the calling program in question. The program can then call another program itself. On termination of the second or subsequent program the vector address is restored from its own PSP, itself taken from the modified vector entry. This technique ensures that any spawned programs always return to their original caller, which on final termination will have the original default termination handler vector replaced from its own PSP.

Note that, if program spawning is employed the technique of altering the terminate handler vector must be used, it is not optional.

If the terminate handler vector is not correctly modified to return control to the caller any (number of) calling programs would permanently remain memory resident, and unable to be removed. Since they would never be re-entered, they would never terminate, and control would return to COMMAND.COM with large portions of memory subsequently unavailable.

INT 23h - The CTRL-C Handler

INT 23h is the CTRL-C handler address, vectored through 0000:008Ch to 0000:008Fh. This routine should never be called directly in user programs, but is called when CTRL-C is detected during any character input/output function and most (but not all) other interrupt function calls. In PCs, other functions than character I/O can be instructed to ignore CTRL-C aborts by setting the BREAK flag to off. This flag is non-functional in the 512, although the command itself is implemented for applications compatibility.

The address of the vector for INT 23h is also available in the PSP of each program, at locations 0Eh to 11h. As for the INT 22h terminate handler routine, the original vector address may be modified using the same philosophy when program spawning is to be employed. Ibis ensures that unexpected user termination of a called routine also returns control to the calling program correctly. As before, the vector contents are copied back from the PSP on final termination of each called routine, the last.

Note that, if the technique of spawning is employed the caution given for INT 22h equally applies, the INT 23h vector *must* be altered accordingly.

INT 24h - The Critical Error Handler

The interrupt vector for INT 24h is located at 0000:0090h to 0000:0093h. It contains the address of the routine called when a critical error (usually a hardware error condition) is encountered. It should never be explicitly called by user programs. The vector is copied into the PSP of a loaded program at locations 12h to 15h. On termination the vector is restored from the PSP as for INT 22h and 23h.

The philosophy of substituting a calling program's re-entry address for the original vector contents is exactly the same for INT 24h as for the previous two interrupts. Like them, if spawning is employed this vector *must* be altered to allow for return from unexpected system or hardware events which might terminate the called routine.

INT 25h - Absolute Disc Read

This call is vectored through the address held at 0000:0094h to 0000.0097h. It provides a direct call to the XIOS to allow direct reading of one or more physical disc sectors from a nominated disc drive. The calling program must allocate a memory buffer of suitable size to hold the incoming data and set registers DS:BX to point to this. The disc drive to be used is indicated by 0 for drive A:, 1 for drive B: and so on. It should be noted that absolute sector numbers start from one on DOS discs, unlike native BBC formats which begin at zero.

It is important to remember that this call leaves the CPU status register contents on the stack after the call returns, which the calling program must explicitly clear itself before any other operations involving the stack are called for.

INT 26h - Absolute Disc Write

This call is vectored through 0000:009ah to 0000:009Bh. Its function is precisely the reverse of INT 25h. A block of memory pointed to by DS:BX is written to the specified drive starting at a specified absolute sector number and continuing for a specified number of sectors.

Note that since this call accesses discs at the hardware level no updating of directories or FATs takes place, therefore it should be used with extreme caution. As for INT 25h, CPU status flags remain on the stack after the call and must be cleared by the program itself.

INT 27h - Terminate and Stay Resident

Note that, information on this interrupt is supplied only for completeness. INT 27h is a relic from MSDOS version 1 and is only required for applications which must run under that version. This interrupt, vectored through 0000:009Ch to 0000.009Fh, terminates direct (i.e. console connected) execution of the calling program, while leaving a (variable) area of memory allocated so it may not be used by subsequently loaded code.

The amount of retained memory is indicated by a segment offset held in register DX, therefore the maximum amount of memory which can be retained by this call is 64kb. As machine's memory sizes and program complexity have grown this amount of memory is now considered insufficient for many purposes, so the call has been replaced with INT 21h function 31h (terminate and keep), which has no such memory limitation. Apart from this point the following description of the general concepts applies to both calls.

The normal termination routine is entered as usual, so as to flush any buffers, close any open files and correctly restore the vectors for interrupts 22h to 24h inclusive, but the memory allocation is not freed. After termination any program code or data in the reserved memory area remains unchanged.

Programs using this technique are usually re-entered subsequently on an interrupt, the vector of which has been directed to point into the program. For example, this technique is commonly employed to use a keyboard interrupt for activating TSR type 'pop-up' programs, although any interrupt may be employed as required and communication can be from another program, rather than direct from the user.

On a call to the chosen interrupt, the intercepted vector directs execution to the resident program(s) which can investigate the call parameters to decide if action is required by them. If it is, then almost any normal DOS activity can then be carried out. If not, then the call is passed to the original vector address for other processing.

There is no theoretical limit to the number of programs that can be chained together on any single vector, provided that the code is written legally. TSR type programs should not be confused with either

spawned programs, which are directly interrelated and generally terminate normally via INT 21h function 4Ch, or with CP/M background and RSX programs, neither of which exist in MS/PCDOS.

BDOS Calls - INT 224

Because the DOS emulator sits between applications code and the BDOS a special interrupt, E0h (224), has been reserved for Digital Research CP/M based systems by Intel, the 86 series chip manufacturer. Use of this call is valid for all DOS Plus programs except those which have entered the background, and BDOS calls may be mixed with DOS interrupt calls within a .COM, .CMD or EXE program as required and as is most convenient.

Interrupt 224 is vectored through 0000:0380h to 0000:0383h, which in DOS Plus 2.1 calls 6000:4401h and is a direct entry into the BDOS kernel, giving direct program access to CP/M functions. Call philosophy is generally much as described for DOS interrupts, except that the BDOS call number must always be placed in general purpose register CL, and the usage of the other registers, while consistent within the range of direct BDOS calls, is different to those used in DOS interrupts.

BDOS Call Notes

A few calls which are available in pure CP/M or CCP/M systems are not permitted in DOS Plus systems. In addition, BDOS calls must be used exclusively in that part of a program which is to enter the background, and in programs which ate destined to be converted to RSXs.

In a few cases the function of a particular BDOS call may be virtually identical to that of a DOS interrupt, and in such cases the programmer can simply choose the call which is the most convenient in the circumstances. However, in general, BDOS calls operate at a lower level than DOS interrupts, allowing access to program functions and system capabilities which are impossible in DOS, hence many DOS Plus utilities use BDOS calls to provide facilities which MSDOS and PCDOS cannot offer.

It should be remembered, when using BDOS calls, that facilities which relate to CP/M disc formats may very well not be appropriate for use with DOS-formatted discs. If such calls are likely to be used in programs which may be run with DOS formatted discs as well as CP/M media, it is the programmer's responsibility to test the media type and issue only suitable BDOS calls.

Host Application Errors

In the case of the 512, peripheral error conditions are detected by the host processor. These are passed across the Tube to the 512 generating an interrupt request. By default, the error number and string are placed in the error buffer of the 512 and a pointer to the error number is initialised. The 512 Tube code then jumps to the error handler, displaying the error details before returning control to the 80186 monitor.

As mentioned in Chapter 3, this facility is not usable by 512 stand-alone applications, because control does not return to the caller. If a program is to make use of the 512's error handling facilities it must intercept the error handler vector and direct control to its own error handling routine. This should be capable of identifying the error and dealing with it in an appropriate manner, either returning control to a suitable point within itself, or terminating tidily, in either case exiting by means of an interrupt return.

The locations of the 512 error handler vector and error pointer are given in the table below.

0000:05F4 Error pointer offset
0000:05F6 Error pointer segment
0000:05F8 Error handler vector offset
0000:05FA Error handler vector segment

Table 8.2 Locations of Error Pointers and Vectors.

Part of an assembler code example is now given, Listing 8.1, to illustrate a typical error handler. This assumes that the program is at 0000:8000.

```
                        cseg    0
                        org     08000h

osnewl                  equ     048h
oswrch                  equ     049h

error_pointer_offset    equ     .05f4h
error_pointer_segment   equ     .05f6h
error_handler_offset    equ     .05f8h
error_handler_segment   equ     .05fah

; initialise error handler to point to my error handler

                        sub     ax,ax
                        mov     ds,ax
                        mov     ax,offset my_error_handler
                        mov     error_handler_offset,ax
                        mov     ax,seg my_error handler
                        mov     error_handler_segment,ax

my_error_handler:
                        lds     si,dword ptr error_pointer
offset
```

```
                        int     osnewl          ; new line
                        inc     si              ; skip
error number
                        cld                     ; set fwd
direction

my error_loop
                        lodsb           ; get error str.chr.
from buffer
                        int     oswrch  ; and write it
                        test    al,al   ; end of string?
                        jnz     my_error_loop   ; no-get
next chr
                        jmp     my_command_loop ; yes -
jump to command loop
```

Listing 8.1 Assembler Code, Example of Error Handler.

80186 Error Messages

Application errors can also be generated within the 512, by means of interrupt 04Fh, supplying both an error number and the error string terminated with a null byte (00h). The error pointer will be initialised as for host detected errors and the error handler used will be as indicated by the contents of the error handler vector (see above).

An outline example is shown below, to illustrate a call to the 80186 error handler. The code tests for the presence of a file before attempting to load it and assumes that the file name is in the current data segment. Note that, if the error handler is called and no error handler has been implemented, the call does not return.

```
; fixed parameter equates
error           equ     04fh    ; tbe error interrupt number

osfind          equ     040h
open_for_input  equ     040h
not found error equ     06dh

cr              equ     13

cseg

look_for_file:  mov     al,open_for_input
```

```
                mov     bx,offset my_file_name
                int     osfind
                or      al,al
                jnz     load_the_file
                int     error
                db      not_found_error, 'Cannot find
file',0

; ******************************************************
; note no return after writing out error unless our own
; error handler is implemented!
; ******************************************************

; The file is loaded here it present (execution continues)
load_the_file:
; The rest of the code

dseg
my_file_name:   db      '$.myfile1',cr
; end
```

Listing 8.2 Assembler Code, Call to Error Handler.

Escape Processing

When an escape condition is detected by the 6502, the top bit of the escape flag at 0000:05F2h in the 512 is set and a Tube interrupt is generated. If programs are intended to process escapes, the condition should be tested for by checking this flag.

if an escape condition exists, the escape must be acknowledged in the host by issuing an OSBYTE with AL = 07Eh (Acknowledge detection of an ESCAPE condition). An application enor message may also be generated if required.

Note that, the escape flag should not be set or unset directly in the 512, as the change will not be reflected on the host side of the Tube, which will record an outstanding unprocessed escape. OSBYTE calls with AL = 07Ch (clear ESCAPE condition) or 07Dh (set ESCAPE condition) should be used to set or reset the escape condition.

9: DOS Plus Disc Structure

Introduction

In this chapter familiarity with the general operation of the disc filing system and its user facilities in DOS Plus. This chapter therefore details how the data is physically stored and managed on the media and the relevance of the fields required by DOS to maintain the system.

The standard DOS formats supponed by the 512, 360k and 720k, are also common to genuine PCs and clones, while the 640k ADFS based format and the 800k 512 format are peculiar to the 512. Given that the number of physical tracks and sectors may vary from one format to another the view of the disc as seen from a DOS application is constant because of the fundions provided for disc management in the Operating System.

Provided that disc access is carried out through standard interrupt facilities, most of the implications of differing disc formats can be ignored in user applications. The only obvious exception to this view is the matter of total storage capacity.

The information following explains the prindples employed by DOS in managing discs for those who may need to gain direct access to the storage medium at the physical level. The information is provided at a level which' should apply to all DOS disc formats. However, a note of caution is called for.

The specifications for the different DOS disc formats were loose enough to allow considerable variation in the detailed interpretation and implementation applied by different PC manufacturers. The result, incompatible discs of supposedly the 'same' format, is well known to most 512 users. A good knowledge of DOS interrupts, together with the precise details of each type of disc format, is needed if discs are to be updated at the physical rather than logical level. A detailed list of various DOS disc format spedfications is given in Appendix D.

Directories

The root directory is the highest logical level of organisation on a disc or volume, but regardless of the directory level examined, the type and format of the data stored is similar, with a limited number of minor exceptions in the case of the root directory. These exceptions are identified where appropriate, otherwise all information relates to any directory at any level.

Every complete single entry in a directory is 32 bytes long and usually represents one of only two items (or a single occurrence of a third in the root directory, the volume label). Apart from this, all entries represent either a single file or another directory. The total number of possible directory entries varies

with the disc format, for example, it is 192 for the root directory of an 800k 512 disc, 112 in the root directory for both Acorn 640k and IBM 360k. Note that directories can occupy more than one sector on the disc.

Whatever the disc size and format, each directory entry must sufficiently describe the object so that DOS knows all that is required about its characteristics and type. The layout of a single directory entry is shown below, with the position of the data shown as an offset from the start of the entry. Where a number is given in brackets next to an item, further information is given in the notes which follow.

Byte offset			
Dec	Hex	Len	Data type or use
0	0	8	Filename (1)
8	8	3	File extension
11	B	1	File attribute (2)
12	C	10	Reserved
22	16	2	Time created or last updated (3)
24	18	2	Date created or last updated (4)
26	1A	2	Starting cluster (5)
28	1C	4	Filesize(6)

Filename and extension (1)

The first byte of a filename field may contain one of several values, with meanings as follows:

00h means this entry has never been used. It also indicates the last used position in the directory, since all entries are initialised to this value during formatting. DOS always re-uses all previously deleted file entries before using a new one.

05h means the first character of the filename is actually E5h, which indicates that the file has been erased.

2Eh is the ASCII code for a full stop and indicates a directory name when in the first character of the filename. If the second character is a space the entry is an alias for the current directory. This is seen as a literal full stop (.) in the directory display.

If the second character is also a full stop then the entry is an alias for the parent directory, in which case the cluster field contains the cluster number of that parent directory, or zero when the parent directory is the root. ntis is seen as two literal full stops in a directory display.

Any other value indicates a valid filename. Both the filename and extension fields are padded with space (20h) Characters if less than the maximum number of characters (i.e. eight.three) is used.

File attribute (2)

The attribute byte of a directory entry can indicate the following meanings when the appropriate bit is set to one:

Bit	Indicates
0	Read only: the file may not be updated
1	Hidden file: excluded from normal searches and directory displays
2	System file: exduded from normal seardies and directory displays
3	The volume label - it can only exist in the root directory
4	Subdirectory: excluded from file based operations (e.g. search for file)
5	Archive bit: Set on whenever a file is modified, cancelled by archiving (backing up) with the 'BACKUP' utility.
6,7	Reserved

File attributes can be set when a file is first created, or subsequently can be modified by command line commands (e.g. FSET) or by system calls from within programs. A normal type file which has not been updated since it was archived has all attribute bits set to zero.

Bits three and four of the attribute byte cannot be modified.

The Time Field (3)

The time field is encoded as 16 bits made up of three binary numbers.
Bits are assigned as follows:

Bits	Contents
0 4	Binary number of two second increments (0 to 29)
5 - Ah	Binary number of minutes (0 59)
Bh - Fh	Binary number of hours (0 to 23)

The Date Field (4)

The date field is encoded as 16 bits made up of three binary numbers.

Bits are assigned as follows.

Bits	Contents
0-4	Day of month (1 to 31)
5 - 8	Month of year (1 to 12)
9h - Fh	Year relative to base 1980

The Cluster Field(s)

The cluster field contents are:

a. The cluster number for the start of the file for a file entry.
b. The cluster number of the directory for a subdirectory or a parent dfrectory.
c. Zero when the parent directory is the root directory.

The value held in this field is not only an absolute cluster number, bnt also a pointer into the cluster entry in the FAT for the flrst or only cluster entry. (See clusters and the file allocation table later in this chapter)

The File Size Field (6)

The file size field is only of relevance to the entries. It contains a four byte integer, with the two Iow-order bytes stored first. It records the actual number of bytes used within the file allocation, not the number of disc bytes allocated to the file (See clusters later in this chapter).

Disc Organisation

Unlike the host's DFS or ADFS filing Systems, DOS does not require that each file occupies a contiguous area of disc. This gives major advantages in convenience, since it provides a self-maintaining system which never requires such operations as compacting, rendering errors like `can't extend` impossible. The only error in this category which is never seen in DOS is `Disc full`. A subsequent `DIR` will show that either this is literally true, or that the file to be written to the disc will not fit into the remaining unallocated spaces

The converse of this convenience is that, over time, as files are extended or contracted and new files are added or old ones deleted, individual files become very fragmented. This has two possible implications for the user.

The first, which is unavoidable, is that pieces of file may become literally scattered around the disc at virtually random locations. This eventually can make itself felt when file access times become extended. It is of little consequence for small files, but can have a very noticeable performance implication for

regularly used large files.

The solution is to format a new disc and copy the data to it file by file. The copying *must* be carried out at the file level because copying the whole disc track by track, (i.e. using the `DISK` program) will also take the existing fragmented structure with it, defeating the objective.

Copying the files individually reorganises them, by joining the fragments together on the new disc. The result is that the newly copied files are again stored in contiguous areas and disc performance is restored. Assuming two floppy drives this can be simply achieved by the command.

```
copy A*.*.* B:
```

Using this technique will require that copying is carried out on a directory by directory basis, and that the appropriate directories have already been created on the target drive. Although the effects of file fragmenting may be less noticeable for longer periods with a hard disc, this problem and its solution (unfortunately very much more tedious in this case) apply equally.

The second implication of fragmented files is that, in the event of a physical disc failure or the accidental deletion of a file, retrieving and assembling the fragments of a file in the correct order by recovering hardware sectors from disc can be an extremely lengthy and laborious process.

As usual, but even more especially important for DOS disc formats than for BBC native mode discs, the very best advice is to attempt to totally avoid this possibility by ensuring adequate backups are taken on a separate disc at frequent intervals during lengthy file updating sessions.

Clusters

So as to keep track of where on a disc the files and directories are physically stored, DOS uses a disc space allocation unit, known alternatively as simply an allocation unit, or more usually as a cluster. A cluster is the smallest amount of disc space that DOS will allocate to a file, no matter how small the amount of data to be stored. The minimum allocation of one cluster per file accounts for the seeming contradiction between the amount of space remaining free on a disc and the fact that an attempt to copy another file which ought to fit' can sometimes produce The `Disc full` message.

A cluster is made up of a number of physical disc sectors and is always a binary number of sectors, 1, 2, 4 or 8. The number of sectors per cluster varies with both the disc type and the sector size. Ufe would be easier if the total size of a cluster in bytes were fixed, but unfortunately there is no hard and fast rule.

For example, in the 512's 800k format there is one 1024 byte sector per cluster (1k), in IBM 360k discs there are two 512 byte sectors per cluster (also 1k), while Acorn's 640k disc has eight 256 byte sectors per cluster (2k) and ICL 720k CP/M discs have four 512 byte sectors per cluster (aIso 2k).

In practice, the size of a cluster is a balancing act between a managable number of clusters and wasted disc space caused by small files occupying a whole cluster. Different manufacturers appear to have differing opinions about the optimum mix. To easily establish the number of sectors per cluster, the show command can be used with the [DRIVE]option. This is slightly awkward, as the SHOW command is particularly fussy about how its parameters are entered. The three outputs shown below from this command were captured by redirecting the output from the command to disc. 'The disc in drive A: contained the SHOW program, while drive B: contained the disc to be investigated. The format of the command given (from drive A:) was therefore:

```
A>SHOW B:[DRIVE] > SHOW.OPn
```

where n was a number from one to three, one for each disc examined. The point to watch carefully is that there must be *no space* between the target drive identifier and the (DRIVE) option. if the command were issued as:

```
A>SHOW B: [DRIVE] > SHOW.opn
```

with a space included, only the capacity of drive B: would be given and the drive Characteristics would be those from drive A:. Clearly this is a bug in the command line interpretation, but it's far too easy to miss, resulting in the wrong information being produced, so take care.

Issued correctly (with no space) the characteristics of three disc formats were obtained for illustration. They are, in order, IBM 360k (formatted on an Olivetti M24) Acorn 64k and DOS Plus 800k. The disc capacity is identified on the third line of each display.

```
    A: Drive characteristics
2,880: 128 Byte Record capacity
  360: Kilobyte Drive capacity
  112: 32 Byte Directory Entries
  112: checked Directory Entries
  128: Records / Directory Entry
    8: Records / Block
    9: Sectors / Track
    0: Reserved Tracks
  512: Bytes / Physical Record

    B: Drive characteristics
5,120: 128 Byte Record capacity
  640: Kilobyte Drive capacity
  112: 32 Byte Directory Entries
  112: Checked Directory Entries
```

```
  126: Records / Directory Entry
   16: Records / Block
   16: sectors / Track
    0: Reserved Tracks
  256: Bytes / Physical Record

    B: Drive Characteristics
6,400: 128 Byte Record capacity
  800: Rilobyte Drive capacity
  192: 32 Byte Directory Entries
  192: checked Directory Entries
  128: Records / Directory Entry
    8: Records / Block
    5: sectors / Track
    0: Reserved Tracks
1,024: Bytes / Physical Record
```

As can be seen in each display, the last line shows the number of bytes per physical record. A physical record is CP/M terminology for a sector, so this line means the number of bytes per sector. This is 512 for the 360k IBM disc, 256 for the 640k Acorn disc and 1024 for the 800k DOS disc.

The seventh line shows a number of records per block. This is 'CP/Mese' again. Internally, CP/M operates on the basis that all disc formats are logically regarded as being made up of 128 byte records (this is nothing to do with records in files). In fact it's merely a historical hangover, but has been maintained for compatibility with earlier CP/M software. The block referred to in this case is the allocation unit, or the same unit as a DOS cluster. In the 800k format disc, for example, there are eight logical 128 byte records (=1024 bytes) per block (i.e. per cluster).

From these values we can easily see that there are 1024 bytes per cluster and, at 1024 bytes per sector, an 800k 512 DOS Plus disc has one sector per cluster.

As a second example, consider the 640k ADFS disc format. The last line shows '`256 bytes / physical record' (= bytes per sectors)` and line seven shows `16 records / block )= 16 * 128 byte logical records per cluster)`. In this case a cluster is 2048 bytes, or 2k, and at 2k bytes per sector there are therefore eight sectors per cluster.

Repeating the exercise for the 360k IBM disc gives the result of two 512-byte sectors per 1k cluster. By using `SHOW [DRIVE]` and this simple calculation the physical organisation of any DOS disc can be easily established.

Fortunately the information which identifies the disc organisation is equally easy to derive in programs. It is stored on the disc at the beginning of the first sector of the first track.

In PC disc formats, this identification information forms part of the first sector of the boot record, which itself always starts at the first sector of track zero. PCs use this fixed location and the data contained to decide if they can recognise the disc, and during their start-up procedures when booting. Although DOS 800k discs are not bootable and sector zero contains the first FAT sector (see below) this information is still present in the same location in the first sector.

The layout of the relevant identification bytes in the first sector of a DOS disc are shown below in Table 9.1.

Byte Offset		**Len**	**Contents**
Dec	Hex		E9 XX XX or EB XX XX (disctype)
0	00	3	OEM name and version
11	0B	2	Bytes per sector
13	0D	1	Sectors per allocation unit (cluster)
14	0E	2	Reserved sectors starting at zero
16	10	1	Number of FATS (usually two)
17	11	2	Number of root directory entries
19	14	1	Total number of sectors in volume
20	15	1	Media descriptor byte (see below)
21	16	2	Number of sectors per FAT
23	18	2	Sectors per track
25	1A	2	Number of heads
27	1C	2	Number of hidden sectors

Table 9.1 Allocation of Bytes in a DOS Disc, Sector 1.

In a bootable IBM disc this would be followed by the bootstrap routine from byte 1E. The disc type byte at offset zero is, in IBM discs, a direct jump to the bootstrap loader. To start the boot process, therefore, a PC merely identifies the disc type and jumps to the address indicated.

The media byte is an IBM disc type identifier. Those IBM formats which are relevant to DOS Plus 2.1 are identified as:

0FCh 5-25" single sided, nine sectors per tracK
0FDh 5.25" double sided, rune sectors per track
0FEh 5.25" single sided, eight sectors per track
0FFh 5.25" double sided, eight sectors per track

There are other identifiers, but they apply only to DOS version 3 or to eight inch floppy discs.

512 800k discs have no boot sector. They are identified as the second type shown, with 0FDh stored in the first byte of the first sector, which is the first FAT sector. This is explained below in the FAT schematic.

The File Allocation Table

Obviously, as files become fragmented, so do both the used and unused clusters, therefore a complete record of the used and unused areas on a disc must be maintained. This is the purpose of the file allocation table, usually referred to as the FAT for short.

When a new file is created the minimum allocation unit, one cluster, is initially made available for the file, and the address of that cluster is stored in the cluster field in the directory entry at offset 1Ah, as shown earlier.

If the size of a single cluster is insufficient for the quantity of data in the file, a second cluster is allocated and the address of that second cluster is stored in the FAT, within the entry for the first cluster. This process of extending the file cluster by cluster is repeated until the required file size is achieved. The last FAT entry for a file, which of course does not point to another FAT entry, contains an end of file marker to indicate that no more clusters follow.

Since DOS has a record of the start cluster for each file or directory on a disc in the directory entry, and within a file clusters are 'chained' together internally, the only additional other information required to manage the whole disc is the address of any free clusters.

The method used is the simplest. The FAT is large enough to hold an entry for every cluster on the disc. Each logical cluster entry, therefore, always resides in a known physical offset in the FAT, regardless of the disc capacity. Further, the directory entry cluster field not only provides a simply calculated index into the FAT, each FAT entry points to the next. In addition the address of each FAT entry can be used to directly calculate the physical location on the disc of the first sector of the corresponding cluster.

Since each FAT entry is, in effect, self indexing, the actual contents of the entry are not needed to point to physical disc locations (i.e. FAT entry one always refers to cluster one). Because of this the contents of a FAT entry can be used to indicate the status of the cluster instead of its location.

Each FAT entry in 512 readable discs occupies three nibbles (12 bits). The possible meanings of the cluster status as recorded in the FAT are:

Value	Meaning
000h	Cluster is available (free for use)

001h	Cluster is in use (this is not a chain number since the minimum chain number is two, i.e., the second cluster)
002h-FEFh	Cluster is part of a file (the contents point to the next cluster in the chain)
FF0h-FF6h	Reserved cluster (cluster is not free for use and is not used)
FF7h	Cluster is bad (contains bad sectors)
FF8h-FFFh	This cluster is the last in a file chain (i.e., an end of file cluster)

In this way, accessing the contents of any cluster entry in the FAT gives all the information required for logical to physical disc mapping as well as the status of that area. In interpreting entries in the FAT the relevant facts are:

1. FAT entry number = cluster number (entry zero is reserved, see below).
2. Cluster number multiplied by sectors per cluster gives the first physical disc sector of the cluster.
3. FAT entries are stored in pairs, in three bytes.
4. To index into the FAT:

    a. multiply the cluster number by three
    b. integer divide the product by two This gives the offset into the FAT.
    c. Move the word at that FAT offset into a register. Note: Remember that FAT entries can span physical sectors as they are a multiple of three bytes.
    d. IF (a) MOD 2=0 THEN result = (c) AND 0FFFh ELSE result=(c) SHR 4 bits
    e. The resulting value 'result' corresponds to one of the FAT entry values given in the preceding table. if it does not either you have made a mistake, or the disc's FAT is corrupt. In the latter case you can attempt to retry using the second FAT.

If you get an error at e) which indicates the two FATS do not agree, your program should report the fact and include an option to stop the current operation immediately. For security you should either include a suitable routine in your program (or use the `DISK` utility) to produce an exact physical duplicate of the disc before proceeding.

Schematically The FAT appears as follows:

Offset	**Content**
0	disc type
1-2	FFFFh (always)

3-5	Group 1 = FAT entries 1+2
6-8	Group 2 = FAT entries 3+4
9-Ah	Group 3 = FAT entries 5+6

and so on, up to the number of FAT entries given by the number of sectors per FAT (offset 16h in sector zero) multiplied by the bytes per sector (offset Bh) divided by three. Within each three-byte FAT group the significant nibbles for each FAT entry are organised as shown in the following diagram, Figure 9.1.

Three byte group:						
Byte no.	1		2		3	
Nibble no	1	2	3	4	5	6
Hex digits for low entry	2	3	-	1	-	-
Hex digits for high entry	-	-	3	-	1	2

Figure 9.1 Organisation of a FAT Entry

Each hex digit shown above is numbered, one to three, where one is the most significant value, three the least. In other words the nibble values that result in the chosen 16-bit register after decoding should be in the order 0123.

As described, each 32 byte directory entry for a file or a directory contains the number of the first cluster entry in the FAT. If the contents of the cluster entry in the FAT are zero that cluster is free for use. Any value between 001h and FF6h indicates that the contents are the next cluster number in the current chain, while a value of 0FF8h or greater is used for the last cluster in any chain and therefore represents an end of file marker.

There is one other cluster status that is important. That is the value of 0FF7h, which indicates a bad cluster that must not be used. This occurs when DOS attempts to write a cluster but fails after a specified number of attempts (usually defaulted to three tries). On failure, the cluster is marked as bad and will not be used again for any purpose This is acceptable to a limited degree on a Winchester, but if it occurs on a floppy, the data should be retrieved and the disc scrapped as soon as possible.

The ability to mark clusters as 'bad', yet to continue to be able to use the disc, is one of the mechanisms used by DOS to increase the reliability of the disc filing system when a disc eventually begins to show signs of failure.

To further improve matters, DOS keeps two copies of the FAT on each disc in most manufacturers' formats (although this is not true for all formats particularly the smaller capacity 'older' formats). This technique not only increases security further, it also provides a means for dealing with certain

exceptional events which might occur during normal operations.

For example, if a file were being copied and part-way through the operation DOS found there was insufficient disc space, the copy would have to be abandoned. If any portion of the file had already been written, some of the unused clusters on the target disc would already have been allocated to the file (which will not now be copied) and the FAT would be incorrect.

To avoid this type of problem, DOS dynamically updates only the first FAT during the disc operation, updating the second FAT only on successful completion. If then the operation must be abandoned part-way through for controllable reasons (e.g. disc full), copying the contents of the second FAT back to the first restores the status quo immediately.

Unfortunately, operations sometimes fail for uncontrolled reasons, such as power failure, a machine hang-up or simply a faulty disc area within the FAT itself. In such cases, the first copy of the FAT may unavoidably be left in an inconsistent or even unreadable state and a part copied or part-updated file might or might not be accessible subsequently.

In the event of a failure which leaves a FAT corrupted, the CHKDSK utility can be used in 'repair' mode, (/F, /L or /R options) when an attempt can be made to rejoin and reallocate appropriate clusters with the intention of making the disc usable again. 'CHKDSK reads and compares the two FATs, attempting to make sense of the two sets of information in the FATS and the contents of the disc. (If one FAT is entirely unreadable the other one alone is used.) This is by no means a guaranteed solution, but when it does work it can save a great deal of effort and time.

If such a failure occurs a warning is given and your first action must be to salvage any available data to a newly formatted disc.

12 and 16 bit FATs

As can be seen above, each FAT entry is three nibbles, or one and a half bytes. This means that the maximum possible number of clusters is 16^3, or 4096 clusters. In IBM hardware this number is inadequate for anything other than floppy discs or small Winchesters (e.g. 10 Mb), so a second type of FAT exists, consisting of 16 bits per entry. This is used for larger disc formats as it permits addressing of up to 65535 clusters.

The 16 bit FAT was only introduced with DOS version 3.0+ and as it applies only to larger Winchesters it is of no relevance to the 512. The size of the FAT entry is decided automatically by DOS during formatting and it should be impossible to have a 16 bit FAT on any floppy disc, therefore all FAT entries in the 512 are of the 12 bit type, regardless of the disc format.

10: The MOVE Utility

Introduction

The MOVE command is an EXE file provided on Issue Disc One of DOS Plus (version 2.1 only). Although they are also still included in the version 2.1 issue, both GETFILE and PUTFILE are superseded by MOVE, which combines the functions of both, plus additional facilities, into one comprehensive utility. Also, MOVE is more robust than the two separate utilities, which are unreliable unless they are the Issue 2.1 version used from hard disc. MOVE's predecessors also had difficulty copying non-ASCII data files. It is recommended that they should not be used if you have MOVE.

MOVE can be used to copy files between any of several BBC filing Systems. As it is a DOS program, MOVE is provided for the obvious purpose of transferring files between the DOS disc and native BBC filing systems. It is also capable of copying data between any two files, either on the same or different discs, in the same or different filing systems. Neither the source nor the destination file need to be in DOS format. It is, therefore, perfectly possible to copy a file from one DES disc to another using MOVE, without leaving DOS.

The motivation for a more comprehensive file copying utility was the need to transfer data between DOS Plus and network systems, since Econet support is now available for DOS Plus. Fortunately Acorn have also included support for both DFS and ADFS. As the majority of use is expected to involve copies to or from DOS, greater attention is given to this type of operation, but examples of other copying operations are also included.

Operation

In the explanations DOS Plus will be referred to as DOS, and DOS disc files will be referred to as DOS files. Native disc filing systems and their files are referred to or prefixed by their usual synonyms, i.e. DFS, ADFS or NFS, or referred to collectively as native BBC or MOS formats. The abbreviation FS means any type of filing system.

When either the source or destination file is to be DOS, the precise format specification (e.g. 360k, 640k, 720k or 800k) is irrelevant. If the 512 can normally read and/or write the disc in question, MOVE will process the files as required without special instructions.

MOVE enables DOS PLUS files to be transferred from DOS to BBC filing systems and subsequently copied back to DOS with the original filename, filetype and attributes preserved. Original BBC FF files can be copied to DOS, but the filename may be truncated and BBC FS file attributes will not be saved.

Note that because the MOVE utility is a one-shot transient (i.e. it cannot be semi-permanently invoked with a command line, like STAR or PIP), unless retrieved via the current PATH setting or by supplying an explicit pathname when calling the command, it must be present in the current directory on the current drive. The corollary to this is that, for two disc systems, the copy must always be to (and) or from DOS.

For two disc Systems, where the copy does not involve DOS either as source or destination filing system, the solution is to set up a memory disc, copy MOVE .EXE to it and then select the M: as the current drive. Other BBC FS copies can then be performed between the two floppy drives. You should also copy COMMAND.COM to the memory disc to minimise the risk of being forced to re-boot in the event of an unexpected error corrupting memory.

In its simplest form the MOVE command may be issued without parameters, when a brief help prompt will be displayed as below:

```
MOVE version 0.66
syntax: move <source filespec> <-fs> [lopt] <dest filespec
> <-fs> [(opt]
```

where:

`<source filespec>` is the name of the Source file or directory
`<dest dirspec>` is the name of the destination file or directory
`<-fs>` is the type of filing system applicable to each disc
`[/opt]` is an option parameter farther qualifying the operation

Command Parameters and Syntax

Source File Specification

The `<source filespec>` indicates that a source file specification should be supplied. Either a single file, or multiple files using the wild card characters * and ? for DOS and * and # for BBC native files, may be used when files are not being renamed during the copy. The source file specification can include any valid DOS or RISC FS pathname, though for DOS files (Source or destination) the current drive and directory will be used by default as normal if no explicit path is specified.

Wherever a drive is specified the normal conventions apply. That is, DOS drive A: equates to ADFS drive 0 or DFS drives 0 and 2, DOS drive B: equates to ADFS drive 1 or DFS drives 1 and 3. MOVE does not recognise passwords in DOS Plus or CP/M file specifications. If files are password protected a default password must be set.

Destination Directory Specification

The <dest dirspec> is used to identify the destination. If multiple files or a single file is being copied without a name change during the copy, this will be a directory name alone, which can include any valid pathname. If a single file is to be renamed during the copy the new filename should be supplied as part of the specification. (See examples using the /r option).

Even if a file is not to be renamed, it is a more reliable technique to use the /r option to qualify the destination specification. This instructs the (single) source file to be copied to a fully named destination, rather than copying to a directory and allowing MOVE to set the filename by default with, occasionally unanticipated character substitutions.

As usual with both DOS and BBC FS copy operations, if the destination file name does not exist it will be created by the copy. If the file does already exist it will be overwritten, provided that it is not locked in ADFS and DOS, or set to read-only in DOS or NFS. If an attempt is made to overwrite a locked BBC FS file MOVE displays an error in the form:

```
copying <source name> as <destination name>
MOS error: File locked
```

The equivalent DOS message for a read only file is Access denied

For all MOVE operations, if a destination directory is specified (which effectively applies to DOS, ADFS or ANFS only) that directory must already exist on the destination disc. Just as in DOS and ADFS when you can only copy to existing directories, so it is with MOVE.

Note that MOVE permits the special native BBC FS character '\' to be used to mean 'the currently selected destination directory', but accepts it for either DOS or native BBC FS transfer operations. You should remember that @ is a valid character in DOS filenames, therefore its use should be avoided as far as possible. (See 'filename Syntax' page 23 in the Dabs Press Master 512 User Guide). In the context of MOVE, using @ as a DOS destination directory name causes a logical impasse. Should MOVE use the current directory, or one literally called '@'?

<-fs>

The <-fs> field is a mandatory parameter, which is used to specify both the source and destination filing systems. The possible value for each entry is as follows:

Filing system	<-fs> entry	Meaning
DOS Plus	-dos	Any DOS Plus disc format

ADFS	-adfs	Acorn Advanced Disc ~
ANFS	-net	Acorn Network rs
DFS	-disc	Acorn Disc Filing System

Note that some DFS discs may have been formatted by an 8271 FDC. The use of such discs is limited to those of a standard format which can normally be read and written correctly by the WD1770/1772 FDC when used in DFS.

[/option]

The `[/option]` field is not mandatory, but it can be used to modify the copy operation. When used it can specify one of the following modifier actions for source or destination files. In the following explanations directory means any directory except the root.

In the explanations the meaning of the option is also supplied, but this is simply as an aid to memory. When entered as part of a command the option is always entered after the filing system and in the exact form shown, including the '/'.

Opt	Meaning
`/C`	Copy directory This is only valid for native BBC FS source specifications. It specifies that directories are to be copied from source to destination. The default for BBC to BBC copy operations without this option is to ignore the directories, copying only the files.
`/L`	Ignore directory This is valid only for native BEC FS source specifications. It specifies that directories should be ignored. It is only required when native BBC files are copied to DOS, as it is the default for BBC to BBC copies. It allows DOS files with no extension to be copied from BBC FS discs when individual filenames are not supplied.
`/S`	SYS: This valid only for use with a DOS Source file specification. It is used to copy DOS system files, that is, those with the /s attribute set which do not appear on directory listings unless requested, (eg. 6502.SYS).
`/R`	Rename : This applies only to destination files and is used when a single file is to be copied, which must be the case if renaming is to take place. It specifies that a destination file has been specified, rather than a directory. Note that the destination specification in this case can use the same filename as the source, or it may be renamed as part of the copying operation The destination filename can also include any legal pathname, therefore a destination directory may still be specified.

Single File Moves

The command

```
MOVE WORDPROC.TXT -DOS :1.$.DOSTEXT -ADFS
```

copies the DOS file called `WORDPROC.TXT` from the current directory in the current drive (A:, C: or the memory disc) to the directory DOSTEXT on ADFS drive 1, using the original filename for the destination file.

```
MOVE B:*.TXT -DOS :0.$.DOSTEXT -ADFS
```

copies all the DOS files with a .TXT extension from the current directory of drive B: to the directory DOSTEXT on ADFS drive 0, using the original filenames for the destination files. When the command is used in this type of operation note that it is essential to ensure that there is sufficient space on the destination disc for all the files, and enough unused filename entries in the chosen directory.

```
MOVE WORDPROC.TXT -DOS :0.$.WRDPROC -Disc /r
```

copies the DOS file called WORDPROC.TXT from the current directory in the current drive (B:, C: or the memory disc) to the root directory on DFS drive 0, using the new filename WRDPROC for the destination file.

The command

```
MOVE :1.$.DOSTEXT.WORDPROC -ADFS :0.D.DOCUMNT -Disc /R
```

copies the file called WORDPROC from the directory DOSTEXT in the ADFS disc in drive 1 to the file called D DOCUMNT on the DFS disc in drive 0. Note that, in this example, the current DOS drive can be only a Winchester or the Memory Disc, and MOVE must be present on that drive, as both floppies are being used for native BBC FS formats.

```
MOVE 0:$.TEXT.* -ADFS :1.$.TEXTBAK -ADFS
```

copies all the files from the directory TEXT on the ADFS disc in drive 0 to the directory called TEXTBAK on the ADFS disc in drive 1, using the source filenames for each of the destination files. When the command is used in this type of operation it is essential to ensure that there is sufficient room on the destination disc, and enough unused filename entries in the chosen directory.

```
MOVE :1.WORDPROC -DISC :1.DOCUMNT -Disc /R
```

copies the file called WORDPROC, in the root directory on the DFS disc in drive 1 to the file called DOCUMNT, also in the root directory of the same DFS disc in drive 1.

Multiple Copies between DOS and MOS

This section explains how to copy a group of files from DOS to a MOS filing system (ANFS in this example), and then how to copy them back to DOS. It is assumed that the user has already logged on to the network by some means (e.g. the STAR utility), has created a directory called DOSPLUS in the User's Root Directory (URD), and that it is the Currently Selected Directory (CSD). (See ANFS documentation for explanation of URD, CSD).

It is not absolutely necessary to create a separate directory for DOS files but it is strongly recommended, as mixing DOS and MOS files in the same directory probably will at least, cause confusion and may also cause MOS files to be copied unnecessarily during MOS to DOS transfers.

The following command will copy all the files with the BAT extension from DOS drive B: to the CSD on ANFS (i.e. DOSPLUS):

```
MOVE B:*.BAT -DOS @ -NET
```

To copy the files back from ANFS to the CSD on DOS type:

```
MOVE BAT.* -NET @ -DOS
```

Note that the source specification for the ANFS files reverses the filetype and filename of the DOS file specification. This is necessary because MOS filenames cannot have a filename extension (see later).

To copy all the files from subdirectory GEM in DOS to the same directory in ANFS use:

```
MOVE \GEM\*.* -DOS @ -NET
```

To copy all the files back from ANFS to DOS directory GEMBAK. In drive B: the command would be:

```
MOVE -NET B: \GEMBAK -DOS
```

Note that, for the ANFS source specification, '*', is equivalent to the DOS '*.*' global wildcard specification and copies all files from the source directory.

Copying Whole Directories

The example above shows how to copy files from DOS to a single MOS directory as all the files were being copied to the same directory. This method is adequate if the source DOS disc does not contain any DOS sub-directories (DSDs), or if only a small number of files are to be copied. However, if the DOS source disc contains several subdirectories (which may themselves contain further subdirectories), then copying all the files to the same MOS directory would result in loss of the DOS directory structure.

To allow DSDs to be created in a MOS directory, the special character \ is used as the first character of the DSD name. This is necessary because of the way that DOS files are stored in MOS filing systems. DOS files that have no filetype extension e.g. `TEXTFILE`, are stored in the specified destination directory. Files that have a filetype extension e.g. `LETTER.BAK` are stored in a directory called `BAK` with the filename `LETTER`. Similarly, the file `VDU.EXE` would be stored in an FSD called EXE using the filename VDU.

An appropriate filetype subdirectory (FSD) is automatically created if it does not already exist.

As DOS filetype extensions are a maximum of three characters, any subdirectories with a name longer than three characters is ignored in a MOS to DOS file copy. This provides an ideal technique for distinguishing between DSDs and FSDS in the same MOS directory. If a DSD name is longer than three characters, it will not be mistaken for a filetype subdirectory.

The following diagram illustrates the mapping between a DOS directory structure and an equivalent MOS directory structure when whole directory copying has been employed and all DOS directories were copied to a directory called DOSPLUS in the User's Root Directory (URD). ANFS is used for the following examples.

DOS To MOS Directory Mapping

;7\j;kaa1a} (LOUcafl~?;l;£;P;us~ck~la
docns-- invoice
/r.tYter.tak
ens--invoice
(vdu.exe} h (orde
exe r} (letter)
ba\
(vdu)
(nns.lib} (stdio.lib)
'lb
I\.
mes staic)

<table>
<tr><td colspan="4">MOS</td><td colspan="4">DOS</td></tr>
<tr><td colspan="4">toucan.exe) --a:\ ---(koala)</td><td colspan="4">$.eric</td></tr>
<tr><td>lc</td><td colspan="3">docs -- (invoice)</td><td>exe {toucan}</td><td>dosplus</td><td>koala</td><td></td></tr>
<tr><td>{vdu.exe}</td><td colspan="2">h (order.txt)</td><td>{letter.bak}</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>(MOS.LIB)</td><td>(STDIO.LIB)</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Where objects enclosed in () are files, objects not enclosed in () are directories.

The above example shows how files would be copied from DOS drive A: to an ANFS directory `$.eric.dosplus`. That is, to directory DOSPLUS in user ERIC on the network.

It is important to note that the DOS Root Directory (DRD), DOSPLUS and the DOS sub-directories \tc, \DOCNS etc are not automatically created by the MOVE utility, i.e., MOVE does not copy a whole directory tree. It is the user's responsibility to create the URDs and DSDs when they are required. This is directly analogous to using the DOS Plus copy command, which copies files between directories but requires the destination directory to exist already.

The character '\' is used above to indicate DOS subdirectories in a MOS directory, rather than using names of four or more characters. Either method can be used at your discretion. Whichever is chosen, it is good practice to decide on a method and to remain consistent thereafter.

The commands required to perform the above MOVE operation are now shown. It is assumed that the following MOS * commands are issued to prepare for the copy and that ERIC is already an authorised user of the network. The preparatory MOS commands are:

```
*net
+I am eric              Log on to net
*cdir dosplus           (create DOS Root Directory)
*cdir dosplus.\lc       (create DOS Subdirectories)
*cdir dosplus.\le.\h    "   "     "   "  "     "
*cdir dosplus.\docns    "   "     "   "  "     "
```

The four required MOVE commands are:

1. Copy all files in the root directory in DOS drive A: to directory DOSPLUS in ANFS.

```
MOVE A:\*.* -DOS DOSPLUS -NET
```

2. Copy all files in directory A.\DOCNS to directory `DOSPLUS - \DOCNS` in ANFS.

```
MOVE A:\DOCNS\*.* -DOS DOSPLUS.\DOCNS -NET
```

3. Copy all files in directory A:\LC to directory DOSPLUS \LC in ANFS

```
MlOVE A:\LC\*.* -DOS DOSPLUS.\LC -NET
```

4. Copy all files with type LIB in directory A:\LC\H to directory DOSPLUS.\Lc.\H in ANFS.

```
MOVE A:\LC\H\*.LIB DOS DOS?LUS.\LC.\n -NET
```

The commands required for the reverse operation, copying from the NET to a DOS disc this time in drive B:, are now given. It is assumed that the following DOS PLUS commands have been given to prepare the directories on the destination disc for the copy.

```
MD B:\LC
MD B:\LC\H
MD B:\DOCNS
```

1. Copy all files from ANFS directory DOSPLUS to the DOS Root directory in drive B:.

```
MOVE &.DOSPLUS.* -NET B:\ -DOS
```

2 Copy all files from ANFS DOS sub-directory DOSPLUS\DOCNS to DOS directory \DOCNS in drive B:

```
MOVE &.DOSPLUS.\DOCNS.* -NET B:\DOCMS -Dos
```

3. Copy all files from ANFS DOS subdirectory DOSPLUS\LC to DOS directory \LC in drive B:.

```
MOVE &.DOSPLUS.\LC.* -NET B:\LC -DOS
```

4. Copy all files with type LIB from ANFS DOS subdirectory DOSPLUS\H to DOS directory \LC\H.

```
MOVE &.DOSPLUS.\LC.\H.LIB.* -NET B:\LC\H -DOS
```

Note that when copying from ANFS, the source subdirectory must be referenced back to the URD using the '&' character, because the currently selected directory in ANFS cannot be restored after the MOVE operation.

Copying Files Between MOS and MOS

In addition to copying files between DOS and MOS, files can also be copied directly from MOS FS to

MOS FS, using either two different filing systems or the same filing system as source and destination.

When copying files between MOS filing systems, MOVE will only copy flies from the specified directory. It will not search any filetype subdirectories unless the /C option is supplied. The default operation provides the same type of copy as performed by the AGES and DFS *COPY command or MOS *MOVE command and is suitable for copying MOS files (e.g. BBC BASIC programs or VIEW text files).

It is also possible to copy a MOS directory that contains DOS files and DOS filetype sub-directories by specifying the /C option in the source file specification. This type of operation is most suitable for copying DOS files that have been saved to a MOS FS disc by a previous MOVE command.

As an illustration, the DOS files copied to the NET directory DOSPLUS in the previous examples could be copied to an ADFS directory `$.DOSPLUS` on drive 1. The ADFS disc must first be prepared for the copy by issuing the following * commands to create the appropriate directories, if they do not already exist.

```
*adfs
*mount 1
*cdir dosplus
*cdir dosplus.\lc
*cdir dosplus.\lc.\h
*cdir dosplus.\docns
```

The following MOVE commands will then perform the copy:

```
MOVE &.DOSPLUS.* -NET /C :1.$.DOSPLUS -ADFS
MOVE &.DOSPLUS.\DOCNS.* -NET /C :1.$.DOSPLUS.\DOCNS -ADFS
M0VE &.DOSPLUS.\LC.* -NET /C :1.$.DOSPLUS.\LC -ADFS
MOVE &.DOSPLUS.\LC.\H.LIB.* -NET /C :1.$.DOSPLUS.\LC.\H -
ADFS
```

Moving MOS Directories

As an example of a similar MOVE listing only MOS FS files, if NET user ERIC also has directories $.ERIC.BASIC and $.ERIC.VIEW,containing, say, BBC BASIC programs and VIEW text files, these could be copied to the same ADFS disc (assuming directories :1.$.BASIC and :1.$.VIEW exist) using the following two MOVE commands:

```
MOVE &.BASIC.* -NET :1.$.BASIC -ADFS
MOVE &.VIEW.* -NET :1.$.VIEW -ADFS
```

Wild Card File Specifications

In the examples above, a group of files has been specified by the '*' wildcard character. This, of course, will match all files in a source directory, for both DOS and MOS files.

It is also possible to specify partial wildcard matching. This means that files must always match a limited range of actual characters, with other individual specified characters remaining wild, for them to qualify for copying. This is achieved by using combinations of the wildcard characters *, ? and #. These wildcard characters have different meanings, which depend on the source filing system. These dependancies are summarised in the following table:

Filing System	Wildcard	No. Characters Matched
DOS	*	0-8 or 0-3(extension)
DOS	?	0-1
MOS	*	0-10
MOS	#	1

Table 10.1 Wildcard Facilities in DOS and MOS

There are certain differences between the use of wildcards for DOS and MOS filing systems which should be noted. To illustrate, consider the following (improbably named) DOS files after they have been copied to a MOS filing system (i.e. ADFS or NFS).

```
A.BAT
AA.BAT
AB.BAT
ABC.BAT
TEXT1B.DOC
TEXT2B.DOC
TESTB.DOC
```

These would be copied to the MOS FS using the following names

```
BAT.A
BAT.AA
BAT.AB
BAT. ABC
DOC.TEXT1B
DOC.TEXT2B
DOC.TESTB
```

The following table shows which files would be matched by the various DOS and MOS wildcard file specifications.

FS	Wildcard	Files Matched
DOS	*.*	All 7 DOS files
DOS	A*.BAT	A.BAT AA.BAT AB.BAT ABC.BAT
DOS	A?.BAT	A.BAT AA.BAT AB.BAT
DOS	TE?T?B.DOC	TEXT1B.DOC -TEXT2B.DOC TESTB.DOC
MOS	*	All 7 MOS files
MOS	BAT.A*	BAT.A BArAA BAT.AB BAT.ABC
MOS	BAT.A#	BAT.AA BAT.AB
MOS	DOC.TE#T#B	DOC.TEXT1B DOC.TEXT2B
MOS	BAT.*A	BAT.A BAT.AA
MOS	DOC.T*B	DOC.TEXTIB DOC.TEKT2B DOC.TESTB
MOS	DOC.T*X*B	DOC.TEXT1B DOC.TEXT2B

Table 10.2 Examples of file Matching using Wildcards.

It can be seen that the DOS single character wildcard, 'r and the MOS single character wildcard, '#' behave differently. In DOS .7. will match any single character or a space Cie, no character) while in MOS, '#' matches only a non-space character.

The global wildcard '*' functions identically for DOS and MOS if it is used as the last character of the filename or filetype.

When issuing MOS wildcard specifications, the '*' character need not be the last character of the filename or type. It is, therefore, possible to use patterns such as *A to mean 'all files ending in A', or run to mean 'all files starting with T and ending with B'. However, uch patterns are illegal in DOS and may not be used. This means that care must be taken when copying from MOS to DOS when destination flames are not explicit.

You must ensure that the wildcard specification will copy all the required files. That is to say, ensure that all the default destination names will be valid. if you are uncertain about whether all the ales will be copied correctly you should use a more explicit pattern. If this causes more files to be copied than required, simply delete any destination files which are not required.

Occasionally using a wildcard specification may result in two destination names being the same when 'translated' to the other filing system. In such cases, the solution is to supply the full source file name and possibly a destination name too.

Wildcard file specifications are permitted only in host filing systems that support the OSGBPB function with A=8 (read names from current directory). This includes DPS, ADFS, NFS and ANFS but excludes

cps (cassette) or RES (ROM filing system).

For CFS and RFs all file specifications must exclusively identify a single file and therefore wildcards are prohibited.

Filename Character Translation

When copying from DOS to MOS the destination filename is formed from the filename and nietype of the source DOS file. In a few special cases this can produce a destination filename which contains invalid characters and which cannot be saved to the MOS filing system.

To overcome this problem, the invalid characters are translated to valid MOS characters to allow the file to be saved. When the file is subsequently copied back from MOS to DOS the invalid characters are translated back again to restore the file's original name. The following table shows the character translation used.

DOS character	MOS character
#	?
$	£
&	{
-	+
@	=

Table 10.3 Character Translation for Invalid characters

The corresponding MOS characters have been chosen to ensure that no conflict occurs with other non-alphanumeric DOS characters that do not require translation.

When copying MOS flies from MOS to DOS (e.g. text files), the following additional non-alphanumeric characters are invalid in DOS and are translated to the underscore character.

The following non-alphanumeric characters are valid under both the MOS and DOS and are not translated:

The use of meaningful filenames, avoiding the inclusion of meaningless chandem in file or directory specifications is by far the best method of preventing compatibility problems between different filing system character conventions.

11: EDBIN - The binary editor

Introduction

EDBIN is the name of the transient utility which provides the facility for EDiting of BINary files in DOS. Binary files are defined as any file which does not contain exclusively displayable alphanumeric characters in the range 20h to 7Eh.

By using EDBIN, any file can be loaded into memory and examined, amended and resaved as necessary. Commands are provided for loading a file into memory, editing memory in hexadecimal and ASCII, for initialising areas of memory to a specified byte or word value, for copying the contents of a block of memory to a different location, for searching for a specified string and comparing two areas of memory.

After editing, data can be written back to disc, either to an original filename or as a different name if required. All EDBIN commands can be terminated, at any time, by using CTRL-C, or suspended using CTRL-S. If console output is suspended by using CTRL-S, it can be restarted usmg any key except CTRL-C, CTRL-S or BREAK. The EDBIN program is included on the DOS Plus Boot Disc (issue Disc 1).

Invoking EDBIN

Two methods are permitted when first calling EDBIN. It may be called either with a filename as a command tail, or with no command tail. This permits a filename to be supplied immediately if, perhaps only one file is to be edited, or for several files to be loaded and saved from within EDBIN. Filenames may include a drive and path specification.

To load EDBIN with a filename supplied, at the DOS prompt enter:

```
EDBIN <filename>
```

where `<filename>` is the file that you wish to edit. The file will be loaded into memory by EDBIN automatically, which will then output a '-' prompt to indicate that it is ready to accept further commands.

Alternatively, if you wish to change discs after loading EDBIN because the file is on a different disc, EDBIN can be started by entering only it's name. Again, EDBIN is loaded and responds with the prompt, but no file is loaded for editing. In this mode, memory can be examined and modified without loading a file, or files may be loaded, edited and resaved.

EDBIN Command Syntax

All EDBIN commands are entered as a single letter, followed by between zero and five parameters terminated by a RETURN. The simplest command is the help command, which requires only an H with no parameters. The redirected console output, produced by entering EDBIN, issuing the help command and quitting, is shown below for illustration:

```
not specified
-h
EDBIN Version 0.52
Avai1able commands are:
c(ompare) (<segment>:) <source start> <source end>
(<segment>:) <dest start>
d(ump)    (<segment>:) {<start offset>} {<end offset>}
e(dit)    (<segment>:) <offset>
f(ill)    (<segment>:) {<start offset>} {<end offset>} <end
offset> <fill byte|word>
h(elp)
m(ove)    (<segment>:) <source start> <source end>
(<segment>:) <dest start>
q(uit)
r(ead)    <filename>
s(earch)  (<segment>:) {<start offset>} {<end offset>}
<"string">
w(rite)   (<filename>)
-q
```

The command prefix is entered immediately following the prompt and identifies the function to be carried out using, where appropriate, the parameters which follow.

Items in angled brackets, e.g. `<segment>`, indicate required parameters though, in some cases, these may be defaulted. Items in round brackets e.g. (`<segment>`), indicate that the enclosed item is optional and maybe omitted from the command. If it is, as in the case of (`<segment>`), and as the enclosed item is a required parameter, a suitable default value will be supplied by EDBIN.

Items separated by a vertical bar, e.g. `<fill byte | word>`, are mutually exclusive alternative entries. One and only one of the two must be supplied. EDBIN commands conform to normal DOS syntax and commands are not case sensitive, with the exception of the `<"string">` argument in the (s) earch command.

Where a segment address must be supplied leading zeros can be omitted and any address (stated in paragraphs) may be specified, but wherever a segment address is supplied, it must be immediately

succeeded by a colon. If no segment address is supplied the colon is also omitted.

For all commands, if the segment address is omitted, the most recently specified (i.e. current) segment is used by default. If the segment is omitted and no segment has been specified in a previous command, the value will default to the address of the current file buffer. Unnecessary leading or trailing spaces are ignored.

Where an offset is required, it may similarly be entered as one to four digits without leading zeros, for example, 50h can be entered as 50, 050 or 0050. If more than 4 hex digits are entered, the most significant digits are ignored, for example, 12345 would be treated as 2345 where `<end offset>` is required, it refers to the byte after the last one required. For example:

```
D 100 180
```

will dump the bytes between 100h and 17Fh inclusive. To specify the end address of a 64k segment in 16 bits, an end offset of 0 is used (i.e. 0FFFFh + 1), so that:

```
F C000 0 E5
```

would fill the last 4k bytes of the current segment with the byte value E5h.

The following table further expands on the meaning of the commands given in the help display above.

Command	Function
C (ompare)	memory, byte by byte, in the specified segment from the specified start offset up to the specified end offset, with the target segment:offset, reporting differences.
D(ump)	memory contents to the console output device in hex and ASCII formats, starting in the specified segment at the start offset up to the end offset.
E(dit)	memory from the specified segment:offset in hex and/or ASCII values.
F(ill)	memory starting from the specified segment:offset with the supplied constant byte or word value
H(elp)	gives a summary of commands and syntax as shown above.
M(ove)	a block of memory starting at the specified segment:offset, ending at the specified end ofiset, to the memory beginning at the specified destination segment:offset.
Q(uit)	EDBIN and return to the calling environment
R(ead)	a file into memory for editing.
S(earch)	memory. starting at the specified segment:offset, continuing to the specified end offset, for matches of the specified string.

W(rite) memory to the specified disc filename.

Table 11.1 EDBIN Command Functions.

The operation of EDBIN is very similar to that of the 80186 monitor, therefore, where commands are similar this is pointed out. For such commands more comprehensive illustrations are available in the monitor examples (see Chapter Three).

A more detailed explanation of EDBIN commands follows below.

EDBIN Commands

C(ompare) two areas of memory

Syntax: `C (<seg>:) <src start> <src end> (<seg>:) <dest start>`

Function

Compares the block of memory, byte by byte, between the offsets source start and source end in the start segment, with the same sized block of memory beginning at `<dest start>` in the second segment specified.

C attempts to match the two specified memory areas and, if differences are found, reports them in the form:

```
<source address> <source byte> <dest address> <dest byte>
```

The addresses are displayed in standard segment:offset format and the byte content at that address is displayed in both hexadecimal and ASCII.

If there are no differences between the two blocks of memory EDBIN displays nothing, returning only the '-' prompt to indicate that the blocks are identical.

Compare can be used on single memory blocks of up to a segment in length by specifying an end offset of 0 for the source end address. If a comparison of greater than 64kbytes is required, it must be divided into blocks of 64 kilobytes and several commands must be used.

Examples

```
C 4000:0 1000 5000:0
```

compares the first 1000h bytes of segment 4000h with the same area in segment 5000h.

```
C 300 500 1000
```

compares the 200h bytes starting at 300h and extending to 4FFh in the current (defaulted) segment, with 200h bytes starting at 1000h, also in the current segment.

```
C 0 0 2000:0
```

compares the whole 64 kilobytes of the current segment with another 64k block starting at segment 2000h.

D(ump) specified memory to screen

Syntax: `D (<segment>:) (<start offset>) (<end offset>)`

Function

This command is identical to the monitor's dump command and produces a memory dump of the 512's RAM to the console output device between the specified addresses. The display is in hex and ASCII, showing 16 bytes on each line. The start address of each line is shown in the normal segment:offset format. Characters outside the standard ASCII range of 20h to 7Eh (CHR$32 to CHR$127) are shown as a full stop in the ASCII portion of the display.

All parameters in this command are optional and may be defaulted. If the segment address is omitted, the last used segment value (or the default buffer address) is used. If the start offset is omitted the display begins from the address of the last byte displayed plus one, or zero if there is no previous address. If the end address is omitted, the start address pIus 128 bytes are displayed. As displays are in multiples of 16 bytes, the default display length is 144 bytes. It impossible to specify an end offset without also supplying a start offset.

The memory contents are displayed as hexadecimal values and ASCII characters. Each line of the memory dump shows the memory address in segment:offset form, followed by 16 bytes of memory contents in hexadecimal values and ASCII characters. Characters outside the ASCII range 20h - 7Eh are shown as a full stop. The minimum number of bytes displayed is 16, regardless of the end address requested.

Examples

```
-D 1234:8000
```

```
1234.8000 20 20 20 20 20 20 20 0D 0A 20 20 20 20 20 20
20         ..
1234.8010 20 20 20 20 20 20 20 20 20 63 46 34 2E 55 6E
2D           cF4.Un-
1234:8020 64 65 6C 65 74 65 20 20 20 63 46 36 2E 54 6F 66
delete   cF6.Tof
1234:8030 69 6C 65 2F 6D 61 72 6B 20 63 46 38 2E 52 69 67
ile/mark cF8.Rig
1234:8040 68 74 2D 66 6C 75 73 68 20 63 46 31 30 2E 53 77
ht-flush cF10.Sw
1234:8050 61 70 2D 66 69 6E 54 20 20 0D 0A OD 0A 0D 0A 4D
ap-find  ......M
1234:8060 45 4E 55 3A 20 20 68 65 6C 70 2C 20 65 78 69 74
ENU:  help, exit
1234:8070 2F 63 6F 6D 6D 61 6E 64 2C 20 73 61 76 65 2F 75 /
command, save/u
1234:8080 6E 73 61 76 65 3B 20 20 6E 61 6D 65 20 74 65 78
nsave;  name tex
```

For brevity only the output for this first command is shown

```
D 2000:100 105
```

Displays 100h to 10Fh (the minimum 16 bytes) in segment 2000h.

```
-D 1000 1580
```

Displays memory from 1000h to 157Fh inclusive in the current segment. If the dump command is subsequently used without parameters i.e.,

```
-D
```

The next 128 bytes of memory from 1580h to 15FFh will be displayed. Further `D` commands entered without parameters will continue to display succeeding memory in blocks of 128 bytes.

E(dit) an area of memory

Syntax: `E (<segment>:) <start offset>`

Function

This command is similar to the 'S(how)' command in the monitor. It displays memory at the specified location, allowing the contents to be examined or amended in hexadecimal or ASCII. A single display line of 16 bytes of memory is displayed in hex and ASCII formats, with the cursor initially under the first digit of the first byte specified.

Cursor movement and data entry is controlled as follows:

<--	moves cursor left. If it is already at the far left of the display. the preceding 16 bytes are displayed.
-->	moves cursor right. If it is already at the far right of the display, the next 16 bytes are displayed.
^ |	displays the next 16 bytes.
<--	displays the preceding 16 bytes.
Shift - <--	moves the cursor immediately to the far left of the current display line
Shift - -->	moves the cursor immediately to the far right of the current display line
ALT-DELETE	toggles between the hex and ASCII areas of the displayed data

The display consists of two 16-byte fields, a hexadecimal display and an ASCII display. ALT-DELETE keys toggle between the two fields (i.e., to switch between the two data formats ALT (i.e., the copy key) should be pressed down and held while DELETE is pressed). While the cursor is in the hex field, data is entered in hex digits, each digit being shifted in from the right.

To advance to the next field, the normal cursor keys are used. SHIFTed cursor keys are used to move to the far left or right of the current field. If the cursor is in the ASCII field, data is entered as ASCII bytes. The cursor is automatically advanced to the next field to allow text to be typed in directly. When text is entered at the far right of the field, the next 16 bytes are automatically displayed to allow typing to continue into the next 16 bytes.

If data is being entered in the ASCII field, control codes in the range 0 to 31 can be entered by pressing the appropriate key in conjunction with the CTRL key, i.e. CTRL-Z will enter the end of text marker 1Ah. The only exception is CTRL-C which terminates the command.

II the user wishes to enter CTRL-C or any other control greater than 7Fh, the entry should be made through the hexadecimal data field as pure hex digits.

Example:

```
E 3000:560
```

displays:

```
3000:560 41 20 62 6F 74 20 6F 66 20 74 65 78 74 1A 07 0O A
bot of text...
```

The spelling mistake in the text shown in the ASCII field can be corrected by either of the following two methods:

1. Move the cursor to the fourth byte in the hex field and enter the hex digits 6 and 9.
2. Use COPY-DEL to move to the ASCII field of the display, move to the fourth byte in this field and enter the character 'i'
   CTRL-C terminates the command for either method.

The results of editing can conveniently be checked by using the dump command, when a large area of memory has been edited, as dump is capable of displaying all of the required portion of memory. Remember that CTRL-S can be used to halt the output, and any key will then resume it when more than a single screenfulll is involved. For our simple one line example above, entering:

```
D 560 570
```

will display the required portion of memory for examination.

```
3000:560 41 20 62 69 74 20 6F 66 20 74 65 78 74 1A 07 0O A
bit of text...
```

F(ill) memory with a constant value

Syntax: `F(<seg>.) <start offset> <end offset> <fill byte|word>`

Function

This command is identical to the monitor's fill command. It fills the 512's memory with the supplied fill value, beginning at the the spedfied start offset and up to but not including the end offset.

Only the segment may be defaulted, both offsets and the fill value must be supplied, but an end offset of 0 can be used to specify a fill operation to the last byte in the specified segment. The fill value supplied may be specified either as a byte or as a word (two bytes). Fill bytes must be given as hexadecimal values, strings are not permitted. If a word value is specified the least significant byte is written first.

Examples:

```
F 3000:1000 1010 55
```

will fill bytes 1000h - 1001h indusive in segment 3000: with the byte value 55h.

```
F 0 1010 1234
```

will all bytes 1000h - 100Fh inclusive in the current (default) segment with the word value 1234h, with the least significant byte written first, that is, the contents of memory will be:

```
3000:0000 34 12 34 12 34 12 34 12 34 12 34 12 34 12 34 12
```

and

```
F 4030:1300 0 20
```

will fill the remainder of segment 4000:, from offset 1000h, with spaces (20h)

H(elp) displays EDBIN help summary

Syntax: H

Function

Displays the help list of EDBIN commands. The H command displays the EDBIN version number and a list of available commands with their required syntax. 'mere are no parameters. See page 168 for sample output.

M(ove) a block of memory to a specified location

Syntax: `M (<seq>:) <src start> <src end> (<seq>:) <dest start>`

Function

This EDBIN command has no monitor equivalent. It moves (i.e., copies) the block of memory specifed to a specifed destination address.

The block of memory at the source address is not generally affected by the move, it is simply copied to the given address. If an overlapping move is specified (a move where part of the destination block overlaps the somce block), the data that will be overwritten by the operation is always moved first to ensure that the destination data is correct. In such moves it is inevitable that the source block must be

corrupted in the overlapping area.

```
M 2000:100 740 1200:10
```

moves 640h bytes from 100h to 73Fh in segment 2000h to the memory block beginning at address 10h in segment 1200h.

```
M 100 740 1200:10
```

Exactly as in the previous command, except that the move in this case is from the current (default) segment. As before 640h bytes are moved from 100h to 73Fh to address 10h in segment 1200h.

Q(uit) EDBIN

Syntax: `Q`

Function

Exits EDBIN and returns to the DOS Plus command line prompt or the calling routine. The Q command has no parameters. It closes the currently active file (if one exists), restores the calling DOS Plus environment and returns to the caller (usually the DOS Plus command line).

R(ead) a File into Memory

Syntax: `R <filename>`

Function

This command has no monitor equivalent, as you can use `*LOAD` at the monitor level. It reads a named fie into memory for examination and possible amendment. The specified file is read into memory at the start of the file buffer. This can always be located by a `D` command with no parameters, when the segment address, used by default, is the buffer's address. The default segment is always reset when a new file is loaded.

The size of the file and the number of bytes actually read are displayed as the file is being loaded into memory. These two numbers should normally be the same hut, in the event of a disc error, the number of bytes read will indicate how much of the file has been read.

When a file is loaded by using the Read command (or was loaded by adding the filename to the initial EDBIN command) the specified `<filename>` becomes the currently active file (CAF). This will

remain the active file until a subsequent Read operation closes the CAF, when the newly loaded file becomes the new CAF. If EDBIN was initially executed with a filename added to the command line for immediate load, that automatically becomes the Currently Active File. (For further information on the the Currently Active File see the Write command).

```
R PROFILE.EXE
```

loads the file `PROFILE.EXE` into memory at the start of the file buffer and resets the current command segment/offset variables, so that file editing default addresses operate on the file buffer.

EDBIN always loads all file images into memory at the start of its file buffer, regardless of the file type, neither CMD or EXE files have the file header stripped.

The entire file is loaded into memory exactly as it is stored on disc. The user should remember that in the case of CMD and EXE files, the start of the code segment is offset by the length of the file header, and the file header should not normally be amended for executable files.

(See Chapter 7 for executable file types and header information)

S(earch) for a String

Syntax: `S (<segment>:) <start offset> <end offset> <"string">`

Function

This command operates exactly like the monitor's SR command. It searches memory between the specified address limits attempting to match the supplied string, the search argument which must be enclosed by double quotes against memory contents. The end offset specified is the end address+1 of the search area, so to allow the search to continue to the end of a segment an end address of 0 can be specified.

Each complete occurence of the string found between the specified limits causes a displayed report of the start address for each match. The addresses of matched strings are shown in normal `segment:offset` format. The search string maybe up to a maximum of 72 characters in length and it must be remembered that the search argument is case sensitive.

This command is useful for locating known data which can be easily or uniquely identified, for example, a key phrase or literal in a program file. Note that each iteration of the search is confined to the specified (or defaulted) segment. A multipIe-segment search must therefore be carried out as a number of separate searches. After locating the string, the Dump command can be used to display a larger section of memory to ensure the correct location has been found.

```
S 3000:400C 0 "myprog.com"
```

searches from 4000h to 0FFFFh inclusive in segment 3000h, attempting to match the text string 'myprog-com'. The condition for a string to be matched is that it must be completely contained within the search area, i.e. if string 'myprog.com' begins only at at 03FFAh then:

```
S 3000:0 4000 "myprog.com"
```

will not locate it since it is not entirely contained within the specified search limits. If, however, the string began at an offset address of 03FF6h or lower, the above search would locate it. Any string consisting of 8-bit characters can be searched for, using escape sequences to allow indusion of control codes and characters above 07Fh (these are compatible with the MOS escape sequences). The I character is used to denote an escape sequence.

The following table summarises the range of control code specifications between 00h and 7Fh:

String entry	Hex value
"|@"	0
"|a" or "||A"	1
to	to
"||z" or '||Z"	1A
"|{"	1C
to	to
"|_"	1F
" "	20
to	to
"~"	7E

Table 11.2 Control Code Specifications (00h to 7Fh).

Exceptions are the following three special cases:

""""	22 (Quote)
"|||"	7C (vertical bar)
"|?"	7F (delete)

Characters of 80h and above are entered by preceding the seven bit value with the |! operator, i.e.,

"|!<char>"	80 - FF

where <char> is any character between 00h and 7Fh, thus 0FFh would be entered as |!|?

Note that all characters in the search are case dependent, including escape sequences. Unrecognised escape arguments are reduced to the argument alone and any surplus or redundant | operators are ignored. For example,

|1 is reduced to 1

|!|!|@ is reduced to |!|@

Any string not terminated by a double quote character, or which contains an odd number of double quote characters, will be reported as a bad string, e.g.,

```
String          Reason for error
"abc            No terminating quote
"ab"            Single quote character in string
"ab"""C"        Odd number of quotes
```

A `Bad string` error is also generated if no argument is supplied for the escape character |, or if a null string or the equivalent is specified, e.g.,

```
String  Reason for error
"a|"    No escape argument
""      Null string
"|!"    Reduces to null string, hence as above
```

The search operation can be stopped at any time by pressing CTRL-C.

Example

```
S 100 7B00 "a text string"
```

will search from 100h to 7AFFh for the string "a text string". Any occurrences of the string in the specified section of memory will be reported in the form:

```
01F3:0151
01F3:0279
01F3:7A01
```

(W)rite a File to Disc

Syntax: `W (<filename>)`

Function

Writes the specified file or currently active file to a disc file.

If the w command is used with no filename, the modified data in memory is written back to the currently active file, the file that was specified in the last a command. If a filename is specified the data is written instead to this specified file, but note that, using the w command with a supplied (different) filename does not affect the currently active filename.

This means that subsequent w commands with no filename will write the data to the original filename specified in the last R command. This permits several saves of a file during editing, for security, and until completion the file can be saved to a work file if required, rather than the original file. Note that normal rules apply. If you write to a new filename it will be created, but if you write to an existing filename, if the file has write permission it will be overwritten.

When writing the file, the number of bytes written is displayed on the screen. If EDBIN cannot write all the bytes to the file an error message is displayed to indicate the problem.

Example

```
W
```

writes data from the file buffer to the currently active filename. The number of bytes written is equal to the length of the file loaded by the n command.

```
W workfile.tmp
```

writes the currently active file to a file called `WORKFILE.TMP`. if this file does not exist it will be created. If it does exist it will be overwritten, except as shown below.

EDBIN Error Messages

File not found

The `R` command has been used to try and read a file that does not exist on the disc, or the full path name for the file has not been specified.

Path not found

The `R <filename>` or `W <filename>` command has been used to read or write a file in a directory

that does not exist.

Too many open files

The R command or W<filename> command has been used to try to open a file when there are already 20 open files. This error should not normally occur, and it will only happen if some previous application has not closed its files on exit. EDBIN requires only two files open at any time.

Access denied

The R command has been used on a Read Only file. The a command attempts to open a file for R/W access, on the assumption that the user will probably want to write the modified data back to the file. To prevent this error, the file should be set to R/W using FSET before entering EDBIN.

This error may also occur if W <filename> has been used to write to a file that has the same name as an existing file that is marked R/O, or if the specified directory is full.

Illegal file handle

The W command has been used when there is no Currently Active File to write the data to. The CAF must be adivated by an R command, or by specifying a filename in the command line when EDBIN is first called.

A: Interrupt Summaries

The Interrupt Vectors

It should be noted that these vectors are as set in an unexpanded 512 running an unmodified DOS Plus 2.1 with no software loaded except `COMMAND.COM`. Users are warned that loading almost any application program may damage some of the vectors. These vector contents should never be accessed directly.

Access should be either by an implicit program jump as a result of a call to the appropriate interrupt, or, when intercepting vectors, by the legal means of INT 21h function 25h or 35h.

In the table below the interrupts are grouped. In all cases the table shows the hex and decimal interrupt number, the offset of the vector in segment zero, the segment and offset of the vector contents and the function.

Internal Hardware Interrupts

Interrupt Hex	Dec	Seg0. Offs	Seg:Offset	Vectored to Function
00	0	0000	06BB:47B5	Divide by zero overflow
0l	1	0004	06BB:47CC	Single step error
02	2	0008	06BB:119F	Non Maskable Interrupt
03	3	000C	06BB:47D6	Breakpoint
04	4	0010	06BB:47DB	Overflow error
05	5	0014	06BB:47E0	Bounds (range) error
06	6	0018	06BB:47E5	Invalid Opcode
07	7	001C	06BB:47EA	'ESC' opcode exception
08	8	0020	06BB:1658	Hardware timer tick ±
09	9	0024	0000:0000	Not supported
0A	10	0028	0000:0000	Not supported
0B	11	002C	0000:0000	Not supported
0C	12	0030	06BB:0713	Tube register 1 or 4 interrupt
0D	13	0034	0000:0000	Not supported

External Hardware Interrupts (IBM ROS Emulation)

0F	14	0038	0000:0000	Not supported (PC FDC)
0F	15	003C	0000:0000	Not supported (PC FDC)

10	16	0040	06BB:21F1	Display services for PC monitors ±
11	17	0044	06BB:2729	Return equipment flags for PC ±
12	18	0048	06B3:2734	Return memory size for PC ±
13	19	004C	0B4E:06EE	if AH = 8 Return no. of floppy drives ± else error = 'Unsupported IBM XT ROS call'
14	20	0050	06BB:2740	Async services int for PC Comm ports ±
15	21	0054	068B:2801	Cassette services interrupt - Not supported
16	22	0058	06BB:2802	Keyboard Driver (Dummy - returns) ±
17	23	005C	06BB.287A	Printer driver
18	24	0060	063B:2935	(ROM BASIC) ±
19	25	0064	06BB:28EA	(Bootstrap loader) ±
1A	26	0068	06BB:1661	IBM time of day
1B	27	006C	0B4E:10DC	IBM ROS kbd. break (Dummy-returns)$
1C	28	0070	06BB:4794	IBM ROS tick (Dummy - returns)
1D	29	0074	068B:2199	Video parameter address pointer §
1E	30	0078	06BB:218E	Disc parameter table pointer §
1F	31	007C	0000:0000	Not supported (Graphics char table in PC)

MSDOS Interrupts

20	32	0080	0B4E:0756	Terminate process (obsolete)
21	33	0084	0B4E:07B1	General function despatcher
22	34	0088	0B4E:069F	Terminate handler address §
23	35	008C	0B4E:08E1	CTRL-C handler address §
24	36	0090	0B4E:0A84	Critical error handler address §
25	37	0094	0B4E:0B43	Absolute disc read
26	38	0098	0B4E:0B99	Absolute disc write
27	39	009C	0B4E:0BBD	Terminate and stay resident (obsolete)

MSDOS Interrupts - Reserved for Future Use

28	40	00A0	06BB:4794	Wait for keyboard input (Dummy-returns)
29	41	00A4	0000:0000	INT 29h through to 2Eh are
2E	46	00B8	O000:0000	not supported - vectors zero
2F	47	00BC	06BB:4794	MSDOS print spooler (Dummy - returns)
30	48	00C0	0000:0000	INT 48h 43h are
3F	63	00FC	0000:0000	not supported - vectors zero
40	64	0100	F000:lDAC	Host OSFIND (f/disc driver on PC/XT)
41	65	0104	F000:1DCF	Host OSGBPB (h/disc param. table PC/XT)
42	66	0108	F00D:1F1F	Host OSBPUT
43	67	Ol0C	F000:1F3B	Host OSBGET
44	68	0110	F000:1F4F	Host OSARGS (Graphics chr. table PC Jnr)
45	69	0114	F000:1F93	Host OSFILE
46	70	0118	F000:19B0	Host OSRDCH
47	71	011C	F000:19D8	Host OSASCI
48	72	0120	F000:19DD	Host OSNEWL
49	73	0124	FODO:19E5	Host OSWRCH
4A	74	0128	FO00:1E6D	Host OSWORD
4B	75	012E	F000:1E10	Host OSBYW
4C	76	0130	F000:lA11	Host OSCLI
4D	77	0134	0000:0000	Not supported
4E	78	0138	0000:0000	Not supported
4F	79	013C	F000:2048	80186 error handler
50	80	0140	0000:0000	INTs 50h to DFh are
DF	223	037C	0000:0000	not supported - vectors zero

In the above range 60h to 67h are the user vectors, while 67h is used for EMS in later PCs.

CP/M (86)/BDOS calls (Available to DOS Plus)

E0	224	0380	0060:0144	Direct call to BDOS (See below)

```
                                        Unused - Reserved for Future
MS/PCDOS Expansion
El      225     0384    0000:0000       INTs Elh to FDh are
F0      253     03F4    0000:0000       not supported - vectors zero
```

Acorn 512 'Specials'

```
FE      254     03F8    06BB:4795       SYSDAT vector for GEM (Kills
INT 224)
FF      255     03FC    06BB:47F4       80186 register dump on
critical error
```

§ These are pointers to code or data tables and must not be called directly.
± These are incomplete implementations or supply suitable simulated return information.

INT 21h Summary by Function Number

In the following list all interrupt functions are shown for completeness. The function number is always in register AH.

A '§' indicates that the function is available in MSDOS v2.0 onwards, but is not officially recognised or supported by the Microsoft Corporation.

A ± against the function indicates that the call is incompatible with DOS Plus 2.1 for the reason shown.

Hex	Dec	Function type
00	00	Terminate process
01	01	Character input with echo
02	02	Character output
03	03	Auxiliary input
04	04	Auxiliary output
05	05	Printer output
06	06	Direct console input/output
07	07	Unaltered character input no echo
08	08	Character input without echo
09	09	Display string
0A	10	Buffered keyboard input
0B	11	Check input status
0C	12	Flush input buffer then input
0D	13	Drive reset

0E	14	Select drive
0F	15	Open file
10	16	Close file
11	17	Find first file
12	18	Find next file
13	19	Delete file
14	20	Sequential read
15	21	Sequential write
16	22	Create file
17	23	Rename file
18	24	MSDOS Reserved function ±
19	25	Get current drive
1A	26	Set disc transfer area (DTA) address
1B	27	Get default drive data
1C	28	Get drive data
1D	29	MSDOS Reserved function ±
1E	30	MSDOS Reserved function ±
1F	31	MSDOS Reserved function ±
20	32	MSDOS Reserved function ±
21	33	Random read
22	34	Random write
23	35	Get file size
24	36	Set relative record number
25	37	Set interrupt vector
26	38	Create new PSP
27	39	Random block read
28	40	Random block write
29	41	Parse filename
2A	42	Get date
2B	43	Set date
2C	44	Get time
2D	45	Set time
2E	46	Set verify flag (non-functional in DOS Plus)
2F	47	Get DTA address
30	48	Get DOS version number
31	49	Terminate and stay resident
32	50	Get disc information (undocumented call) §±
33	51	Get or set break flag (non-functional in DOS Plus)
34	52	Find active byte (undocumented call) §±
35	53	Get interrupt vector
36	54	Get drive allocation data
37	55	Set or get DOS switch character
38	56	Get or set country information

39	57	Create directory
3A	58	Delete directory
3B	59	Set current directory
3C	60	Create file
3D	61	Open file
3E	62	Close file
3F	63	Read file or device
40	64	Write file or device
41	65	Delete file
42	66	Set file pointer
43	67	Get or set file attributes
44	68	I/O control
45	69	Duplicate handle
46	70	Redirect handle
47	71	Get current directory
48	72	Allocate memory block
49	73	Release memory block
4A	74	Resize memory block
45	75	Execute program
46	76	Terminate process with return code
4D	77	Get return code
4E	78	Find first file
4F	79	Find next file
50	80	Set address of PSP (undocumented call) §±
51	81	MSDOS Reserved function ±
52	82	MSDOS Reserved function ±
53	83	MSDOS Reserved function ±
54	84	Get verify flag (non-functional in DOS Plus) ±
55	85	MSDOS Reserved function ±
56	86	Rename file (not implemented) ±
57	87	Get or set file date and time stamps (not implemented) ±
58	88	Get or Set allocation strategy (DOS Version 3.0+) ±
59	89	Get extended error information (DOS Version 3.0+) ±
5A	90	Create temporary file (DOS Version 3.0+) ±
5B	91	Create new file (DOS Version 3-0+) ±
5C	92	Lock or unlock file region (DOS Version 3.0+) ±
5D	93	MSDOS Reserved function ±
5E	94	Get machine name/Get or set printer setup (DOS Version 3.1+) ±
5F	95	Device redirection (DOS Version 3.1+) ±
60	96	MSDOS Reserved function ±
61	97	MSDOS Reserved function ±
62	98	Get PSP address (DOS Version 3.0+) ±
63	99	Get lead byte table (DOS Version 2.25 only) ±

```
64  100 MSDOS Reserved function ±
65  101 Get extended country information (DOS Version 3.3+) ±
66  102 Get or set code page (DOS Version 3.3+) ±
67  103 Set handle count (DOS Version 3.3+) ±
68  104 Commit file (DOS Version 3.3+) ±
E0  224 Call BDOS (See below)
```

INT 21h Function Summary by Operation Type

In the following list only interrupts compatible with DOS version 2 and which should be available in DOS Plus 2.1 are shown. where a function has (v.1) shown, it is the original implementation from DOS version 1 and has been superseded or has improved facilities in the later call referred to, which should generally be used in preference. Calls marked ± are not usable in the 512 even though they are DOS 2.1 calls.

Hex Dec Function type

Character Input/output

```
0l  1 Character input with echo
02  2 character output
03  3 Auxiliary input
04  4 Auxiliary output
05  5 Printer output
06  6 Direct console input/output
07  7 Unfiltered character input without echo
08  8 Character input without echo
09  9 Display string
0A 10 Buffered keyboard input
0B 11 Check input status
0C 12 Flush input buffer then input
```

Disc Operations

```
0D 13 Drive reset
0E 14 Select drive
19 25 Get current drive
18 27 Get default drive data
1C 28 Get drive data
   46 Set verify flag
32 50 Get disc information (undocumented call) §
36 54 Get drive allocation data
```

54 84 Get verify flag

Directory Functions

39 57 Create directory
3A 58 Delete directory
3B 59 Set current directory
47 71 Get current directory

File operations

0F 15 Open file (V.1 use 3D)
10 16 Close file (V.1 use 3E)
11 17 Find first file(V.1 use 4E)
12 18 Find next file (V.1 use 4F)
13 19 Delete file (V.1 use 41)
16 22 Create file (V.1 use 3C)
17 23 Rename file
23 35 Get file size
29 41 Parse file name
3C 60 Create file
3D 61 Open file
3E 62 Close file
41 65 Delete file
43 67 Get or set file attributes
45 69 Duplicate handle
46 70 Redirect handle
4E 78 Find first file
4F 79 Find next file
56 86 Rename file
57 87 Get or set file date and time

Record Operations

14 20 Sequential read
15 21 Sequential write
lA 26 Set disk transfer area (DTA) address
21 33 Random read (See 27)
22 34 Random write (See 28)
24 36 Set relative record number
27 39 Random block read
28 40 Random block write

2F 47 Get DTA address
3F 63 Read file or device
40 64 Write file or device
42 66 set file pointer

Process Management

00 00 Terminate process (V.1 use 4C)
26 38 Create new PSP (V.1 use 4B)
31 49 Terminate and stay resident
34 52 Find active byle (undocumented call) §±
4B 75 Execute program
4C 76 Terminate process with return code
4D 77 Get return code
50 80 Set address of PSP (undocumented call) §±

Memory Management

48 72 Allocate memory block
49 73 Release memory block
4A 74 Resize memory block

Time/Date Functions

2A 42 Get date
2B 43 Set date
2C 44 Get time
2D 45 Set time

Miscellaneous System Functions

25 37 Set interrupt vector
30 48 Get MSDOS version number
33 51 Get or set break flag: (not implemented) ±
35 53 Get interrupt vector
37 55 Set or get DOS switch character
38 56 Get or set country information
44 68 I/0 control (limited implementation)
E0 224 Call BDOS (see below)

INT E0h (BDOS) Summary by Function Number

In external (i.e. DOS resident console-connected) programs, all functions are initiated by a call to INT 21h with a fundion code of 0E0h in AH and the BDOS function number in CL. There are gaps in the sequence, either because the call is not implemented up to 2.1, or because it is a pure CP/M function not available in DOS Plus.

```
Hex Dec Function type
00  0 Terminate program (exactly as 8Fh)
01  1 Read a character
02  2 Write a character
03  3 Auxiliary input
04  4 Auxiliary output
05  5 Write a character to the list device
06  6 Direct console I/O
07  7 Auxiliary input status
08  8 Auxiliary output status
09  9 String output
0A 10 Read an edited line
0B 11 Get console status
0C 12 Get BDOS version number
0D 13 Reset all drives
0E 14 Set default drive
0F 15 Open a file for record access
10 16 Close a file
11 17 Search for first matching file
12 18 Search for next matching file
13 19 Delete a file
14 20 Read records sequentially
15 21 Write records sequentially
16 22 Create a file entry
17 23 Rename a file
18 24 Return 'Login Vector' for all logged in drives
19 25 Return the default drive ID
1A 26 Set the DMA buffer offset address
18 27 Get address of disc allocation vectors
1C 28 Set the default drive to read-only
1D 29 Get read-only drive vectors
lE 30 Set file attributes
1F 31 Get calling process's default disc DPB address
20 32 Get/set current user number
21 33 Read random records
22 34 Write random records
23 35 Return the size of a file
24 36 Set the random record number in the FCB
```

25 37 Reset the specified drives
26 38 Unallocated
27 39 Free specified drives
28 40 Write random records and zero-fill unallocated blocks
29 41 Unallocated
2A 42 Unallocated
2B 43 Unallocated
2C 44 Set the system multi-sector count
2D 45 Set filing system error mode
2E 46 Return unallocated space on the specified drive
2F 47 Load and execute program specified in DMA buffer
30 48 Write pending internal disc buffers to media
31 49 Get/set system variable
32 50 Call BIOS (XIOS) directly (See below)
33 51 Set the DMA buffer segment address
34 52 Return segment:offset of DMA buffer
35 53 Allocate maximum memory available in system
36 54 Allocate maximum memory available at specified address
37 55 Allocate a segment as specifed in MCB
38 56 Allocate specified memory at specified address
39 57 Free memory starting at a specified address
3A 58 Unallocated
3B 59 Load CMD file into memory
3C 60 Call RSX program

3D 61 Calls 3Dh to 62h are
62 98 unallocated

63 99 Truncate file to the specified random record number
64 100 Create or update a directory label
65 101 Get directory label data byte for specified drive
66 102 Return a file's date/time stamps and password mode
67 103 Create or update a file's extended FCB
68 104 Set internal date and time
69 105 Get internal date and time
6A 106 Set the default password
6B 107 Get the DOS Plus serial number (Not version number)
6C 108 Get/set program return cole
6D 109 Get/set console mode
6E 110 Get/set output delimiter
6F 111 Write a specified block of characters
70 112 Write a specified number of characters to the list device

```
71 113 Calls 71h to 8Ch are
8C 140 unallocated

8D 141 Delay a specified number of ticks (wait)
8E 142 Relinquish processor to other programs (multitasking)
8F 143 Terminate program (exactly as 0)

90 144 Calls 90h to 92h are
92 146 unallocated

93 147 Detach program from console (enter background)

94 148 Calls 94h to 97h are
97 151 Unallocated

98 152 Parse ASCII string and initialise FCB
99 153 Unallocated
9A 154 Get System data area (SYSDAT) address

95 155 Calls 95h to ADh are
AB 171 Unallocated

AC 172 Read characters from AUXIN
AD 173 Send characters to AUXOUT

AE 174 Calls AEh to FF are
FF 255 unallocated
```

INT E0h Function Summary by Operation Type

The BDOS call summary shown below groups functions by operation type. The list is essentially in numerical order within operations, but where two functions complement each other they have been placed together if this does not occur naturally.

It must be remembered that, in CP/M disc formats a directory sub- structure is not supported, nor is file date/time stamping the norm (although this can be initiated by the DOS Plus INIT command). Many of the disc and file handling calls may, therefore, not be appropriate for use with MSDOS or PCDOS disc formats.

In CP/M, files are protected or separated from each other by being assigned to a group under different user numbers (0 to 15) rather than by being placed in different directories. In CP/M, media attributes can

be set at disc and/or file level which can, even in direct BDOS calls, prevent or restrict unanthorised access to files in several different ways.

Hex	Dec	Function type
		Auxiliary character I/O
03	3	Auxiliary input
04	4	Auxiliary output
07	7	Auxiliary input status
08	8	Auxiliary output status
AC	172	Read characters from AUXIN
AD	173	Send characters to AUXOUT

Console Input/Output

01	1	Read a character
02	2	Write a character
06	6	Direct console I/O
09	9	String output
0A	10	Read an edited line
0B	11	Get console status
6D	109	Get/set console mode
6E	110	Get/set output delimiter
6F	111	Write a specified block of characters
93	147	Detach program from console (enter background)

Disc Drive open functions

0D	13	Reset all drives
0E	14	Set default drive
18	24	Return 'Login Vector' for all logged in drives
19	25	Return the default drive ID
1B	27	Get address of disc allocation vectors
1C	28	Set the default drive to read-only
10	29	Get read-only drive vectors
1F	31	Get calling process's default disc DPB address
25	37	Reset the specified drives
27	39	Free specified drives
2E	46	Return unallocated space on the specified drive
30	48	Flush (write) pending internal disc buffers to media
64	100	Create or update a directory label
65	101	Get directory label data byte for specified drive

File Operations

0F 15 Open a file for record access
10 16 Close a file
11 17 Search for first matching file
12 Is Search for next matching file
13 19 Delete a file
14 20 Read records sequentially
15 21 Write records sequentially
16 22 Create a file entry
17 23 Rename a file
1A 26 Set the DMA buffer offset address
1E 30 set file attributes
20 32 Get/set current user number
21 33 Read records randomly
22 34 Write records randomly
23 35 Return the size of a file
24 36 set the random record number in the PCB
28 40 Write random records and zero-fill unallocated blocks
2C 44 Set the system multi-sector count
2D 45 Set filing system error mode
33 51 Set the DMA bufkr segment address
34 52 Return segment:offset of DMA buffer
63 99 Truncate file to the specified random record number
66 102 Return a file's date/time stamps and password mode
67 103 Create or update a files extended FCB
6A 106 Set the default password
98 152 Parse ASCII string and initialise FCB

List Device Calls

05 5 Write a character to the list device
70 112 Write a specified number of characters to the list device

Process Management

00 0 Terminate program (exactly as 8Fh)
2F 47 Load and execute program specified DMA buffer
3B 59 Load CMD file into memory
3C 60 Call RSX program
6C 108 Get/set program return code
8D 141 Delay a specified number of ticks (wait)

```
8E 142  Relinquish processor to other programs (multitask)
8F 143  Terminate program (exactly as 00)
```

Memory Management

```
35 53 Allocate maximum memory available in system
36 54 Allocate maximum memory available at specified address
37 55 Allocate a segment as specified in MCB
38 56 Allocate specified memory at specified address
39 57 Free memory starting from a specified address
```

Time/Date Functions

```
68 104 Set internal date and time
69 105 Get internal date and time
        (NB: Time is in hours and minutes only)
```

Miscellaneous Functions

```
0C  12 Get BDOS version number
31  49 Get/set system variable
32  50 Call BIOS (XIOS) directly (See below)
6B 107 Get the DOS Plus serial number (Not version number)
9A 154 Get Sytem data area (SYSDAT) address
```

INT E0h Function 32h XIOS Subfunctions

Within interrupt E0h is function 32h, which gives another range of subfunctions, direct calls to the XIOS. As usual CL is set to the interrupt function number, are, while DS:DX are used as pointers to the memory location containing the XIOS call parameters

As can be seen, many of the standard CP/M functions are almost identical to the slightly easier to use BDOS functions and hence are of little advantage. The block of functions with values of 80h or greater are peculiar to the XIOS as implemented for Acorn 6502 host systems, while function 80h itself is provided for use within the BDOS and the bootstrap loader.

Hex Dec Function type

```
00  00  Terminate program (Same as BDOS function 0/8Fh)
0l  01  As above
02  02  Check for console input status
03  03  Read character from console
```

```
04  04  Write character to console
05  05  Write character to list device
06  06  Write character to auxiliary device
07  07  Read character from auxiliary device
0F  15  Return list device status

10 16 Calls l0h to l4h are
14 20 reserved
15 21 Device initialisation
16 22 Check console output status

17  23 Calls l7h to 7Fh are
7F 127 reserved

80 128 XIOS version (ACORN specific)
81 129 Get tube semaphore
82 130 Release the semaphore
83 131 Select text-graphics
84 132 Update graphics rectangle (B&W)
85 133 Update graphics rectangle (colour)
86 134 Get release/update mouse
87 135 Get system error info
88 136 Entry in CLOCK called by WatchDog RSP
89 137 BBC OSBYTE function
8A 138 BBC OSWORD function
```

B: DOS Interrupts - INT 20h to 27h

The following is intended to be a guide to the main (user) interrupts provided in DOS Plus 2.1 for the 512. However, if fully detailed specifications of the calls as implemented in MSDOS or PCDOS are required there are many excellent books available covering the subject.

Each of the 512's Dos Plus supported user interrupts between 20h and 27h is shown with any entry conditions and returned values. The operation and purpose is described with brief appropriate additional comments.

INT 20h - Program terminate

This is one of several calls by which a program can terminate execution. It informs DOS that the program is complete and that occupied memory should be released.

Action:	On execution the call restores vectors for INTs 22h to 24h from the program's PSP, flushes any buffers and transfers control to the terminate handler address.
On entry:	CS = Segment address of PSP
Returns:	Nothing
Notes:	The preferred termination method is INT 21h function 4Ch. Since this call has a memory addressing limit of 16 bits, (64k bytes) it is of limited use, and cannot be used in an EXE program.

INT 21h - The general function despatcher

Most of the general functions and services offered by DOS are implemented through this interrupt . The functions available are well standardised and should be common to all MSDOS, PCDOS and DOS Plus systems. Well behaved programs, therefore, should use these facilities in preference to any other methods available for the widest range of compatibility.

INT 21h in the 512's implementation of DOS Plus 2.1 provides 77 official functions, two of which are non-functional and return with no action. Within this range some calls have subfunctions which further extend the range of operations.

In all calls, on entry AH defines the function. Other parameters may also be required in other registers. Where a memory block is used by the call this is specified in the normal segment:offset form. In all cases the general programming technique is to set AH to the function pointer, set up the required register contents (and the memory block if necessary) then to issue the call by the assembly code INT instruction. To call the recommended program terminate routine, INT 21h function 4Ch, the relevant parts of the code would be:

```
; Constant equates
Prog_exit               equ 4ch
Function_despatcher     equ 21h
                        org 0100
                        program code here
                        .......
                        .......
                        .......
exit:                                                   ;
common program exit
point                   mov     al, Return_code         ;
set up result
                        mov     ah, Prog_exit           ;
Set up terminate
                                                        ;
process
                        int     Function_despatcher     ;
and leave
; variable data
Return_code             db ?                            ;
Default 0 success
                                                        ;
set value on failure
```

```
END
```

There are other methods of implementing INT calls, but they are not recommended as normal techniques and are less efficient. The two most likely to be encountered are included here only for infomation.

1. Setting up the entry conditions and executing a long call to 0050h in the PSP (only works in DOS v.2+).
2. Loading the CL register with the function number and executing an intra-sgment call to offset 0050h in the PSP, which contains a long call to the function despatcher. This method only works for function calls of 24h or less, and has the further disadvantage that the contents of register AX are always destroyed.

If calls are made by the approved method the contents of all registers are preserved through calls, except where those registers are used to return results. The obvious exception to this is function 4Bh, EXEC, which transfers control to a spawned program, when the state of all registers except the instruction pointers, but including the stack pointers, should be treated as undefined unless specific returned values are expected.

If spawning is employed register contents which must be be preserved should be pushed onto the stack, and the stack registers themselves (i.e. SS:SI) should be saved in known memory locations for explicit later retrieval.

INT 21h functions 00h to 24h are based on and are, with a few exceptions, direct equivalents to the corresponding CP/M calls. In these calls success or failure is typically signalled by the value returned in register AL. For the remaining (i.e. MSDOS) calls, the carry flag is more usually used, carry clear indicating success, carry set indicating failure of the function, often accompanied by an error code in register AX.

Functions up to and including 57h are documented in this section, all INT 21h functions with a higher number applying to later versions of DOS than 2.11. Note that functions 32h, 34h and 50h and above are included, though they are not supported by DOS Plus, because these do occur in MSDOS version 2.0 and above, and might be encountered in MSDOS v2.0 programs.

Function 0- Program terminate

Action:	On execution the call restores vectors for INTS 22h to 24h from the PSP, flushes any buffers and transfers control to the terminate handler address.
On entry:	AH = 0 CS = Segment address of PSP
Returns:	Nothing

Notes: Equivalent of CP/M BDOS call 00h. INT 21h function 4Ch is preferred.

Function 1- Character input with echo

Action: Reads a character from the standard input device and echoes it to the standard output device.
If no character is ready it waits until one is available.
I/O can be re-directed, but prevents detection of OEF.

On entry: AH = 01h

Returns: AL = 8 bit data input

Notes: Equivalent to CP/M BDOS call 01h, except that if the character is CTRL-C an INT 23h is performed.

Function 2 - Character output

Action: Outputs a character to the standard output device. I/O can be re-directed, but prevents detection of 'disc full'.

On entry: AH = 02h
DL = 8 bit data (usually ASCII character)

Returns: Nothing

Notes:

Function 3- Auxiliary input

Action: Reads a character from the current auxilliary device.

On entry: AH = 03h

Returns: AL = 8 bit data input

Notes: There is no way to read the status of the serial port or to detect errors through this call, therefore most PC comms packages drive the hardware directly, hence their general incompatibility with the 512.

Function 4- Auxiliary output

Action: Outputs a character to the current auxiliary device.

On entry: AH = 04h
DL = 8 bit data

Returns: Nothing

Notes: There is no way to read the status of the serial port or to detect errors through this call.
Comments as Function 3.

Function 5- Printer output

Action: Sends a Character to the current listing device.

On entry: AH = 05h
DL = 8 bit data

Returns: Nothing

Notes: If the printer is busy this call will wait until the data is sent.
There is no way to poll the printer status in DOS.

Function 6- Direct console I/O

Action: Reads a character from the standard input device or returns zero if no character available. Also can write a character to the current standard output device. I/O can be redirected but prevents detection of EOF on input or 'disc full' on output.

On entry: AH = 06h
DL = function requested: 0Ch to 0FEh = output
(DL = character to be output)
0FFh = Input request

Returns: If output - nothing
If input - data ready: zero flag clear, AL = 8 bit data
If data not ready: zero flag set

Notes: This call ignores CTRL-X.

Function 7 - Unfiltered character input no echo

Action: Reads a character from the standard input device without echoing it to the display.
If no character is ready it waits until one is available.

On entry: AH = 07h

Returns: AL = 8 bit data input

Notes: This call ignores CTRL-C, use function 8 if CTRL-C processing is required. There is no CP/M equivalent.

Function 08- Character input with no echo

Action: Reads a character from the standard input device without copying it to the display.

If no character is ready it waits until one is available.

On entry: AH = 08h

Returns: AL = 8 bit data input

Notes: If CTRL-C is detected INT 23h is executed.

Function 09- Output character string

Action: Writes a string to the display.

On entry: AH = 09h
DS:DX = segment:offset of string

Returns: Nothing

Notes: The string must be terminated by the $ character (24h), which is not transmitted. Any ASCII codes can be embedded within the string.

Function 0Ah - Buffered input

Action: Reads a string from the current input device up to and including an ASCII carriage return (0Dh), placing the received data in a user-defined buffer Input can be re directed, but this prevents detection of EOF

On entry: AH = 0Ah
DS:DX = segment:offset of string buffer

Returns: Nothing

Notes: The first byte of the buffer specifies the maximum number of characters it can hold (1 to 255). This value must be supplied by the user. The second byte of the buffer is set by DOS to the number of characters actually read, excluding the terminating RETURN. If the buffer fills to one less than its maximum size the bell is sounded and subsequent input is ignored.

If a CTRL-C is detected an INT 23h is executed. Normal DOS keyboard editing is supported during input

Function 0Bh - Get input status

Action: Checks whether a character is available from the standard input device. Input can be redirected

On entry: AH = 0Bh

Returns: AL = 0 if no character available
AL = 0FFh if character available

Notes: Notes: if an input character is waiting this function continues to return a true flag until the character is read by a call to function 1, 6, 7, 8 or 0Ah.

Function 0Ch - Reset input buffer and input

Action: Clears the standard input buffer then invokes one of the standard input functions.

On entry: AH = 0Ch
AL = number of input function to be invoked, which must be 1, 6, 7, 8 or 0Ah.

Returns: If function 0Ah - Nothing
If function 1, 6, 7, or 8 then AL = 8 bit data

Notes: The purpose of this function is to ignore any type-ahead data and force a wait for new character input which must occur after the call is issued.

Function 0Dh - Disc reset

Action: Flush all outstanding file buffers by physically updating to disc.

On entry: AH = 0Dh

Returns: Nothing

Notes: This call does *not* update disc directories for any open files.
If the program fails to close files before the disc is removed and the files have changed size, their directory entries will be incorrect.

Function 0Eh - Set default disc drive

Action: Sets the specified drive to be the default drive and returns the total number of logical drives in the system.

On entry: AH = 0Eh
DL = drive code (A: = 0, B: = 1, etc)

Returns: AL = the number of logical drives in the system.

Notes: In the 512's DOS Plus (2.1) this call always returns five as the number of logical drives, though a maxirnum of four are supported.

Function 0Fh - Open a file

Action: Opens a file and makes it available for read/write operations.

On entry: AH = 0Fh
DS:DX = segment:offset of the file control block (FCB)

Returns: AL = 0 if file found
AL = FFh if file not found

Notes: This call requires a user allocated memory control block.
If the call is successful the FCB data is filled by DOS.

Function 10h - Close file

Action: Closes a file and updates the directory if the file has been modified.

On entry: AH = 10h
DS:DX = Segment offset of the FCB for the file

Returns: AL = 0 if directory updated successfully
AL = 0FFh if fiIe not found in directory

Notes: This call may only be used after a file has been successfully opened and an FCB created.

Function 11h - Find first file

Action: Search for a specified filename in the current directory of the current drive.

On entry: AH = 11h
DS:DX = Segment:offset of the FCB

Returns: AL = 0 if matching filename found
AL = 0FFh if no matching file found

Notes: It is important to use INT 21h function 1Ah to set the DTA to a buffer of adequate size before using this call.

In MSDOS v2 only the '?' wildcard is permitted. If wildcards are specified the first matching name is returned.

If an extended FCB is used, an attribute byte determines the type of files searched for. INT 21h function 4Eh is preferred to this call.

Function 12h - Find next file

Action: Searches the current directory in the current drive for the next matching filename after a previously successful call to function 11h.

On entry: DS:DX = segment:offset of the FCB

Returns: AL = 0 if matching filename found
AL = 0FFh if no matching file found

Notes: As for Function 11h. Function 4Fh is preferred.

Function 13h - Delete file

Action: Deletes all matching files from the current directory.

On entry: AH = 13h
DS:DX = Segment:offset of the FCB

Returns: AL = 0 if matching fie(s) deleted
AL = 0FFh if no matching file found or all matching files are read-only.

Notes: The '?' wildcard is permitted. If more than one match occurs all matched filenames will be deleted.

Function 14h - Sequential read

Action: Reads the next sequential block of data from a file and increments the file pointer.

On entry: AH = 14h
DS.DX = Segment:offset of previously opened FCB

Returns: AL = 0 if successful
AL = 1 if end of file reached
AL = 2 if segment wrap occurs
AL = 3 if partial record read at end of file

Notes: The record is read into memory at the DTA address specified by the most recent call to function 1Ah. The size of the block read is is specified by the record size field in the FCB.

Function 15h - Sequential write

Action: Writes the next sequential block of data to a file and increments the file pointer.

On entry: AH = 15h
DS.DX = Segment:offset of previously opened FCB

Returns: AL = 0 if successful
AL = 1 if disc full
AL = 2 if segment wrap occurs

Notes: The record is written from memory at the DTA address specified by the most recent call to function 1Ah. The size of the block written is specified by the record size field in the FCB.

Function 16h - Create or truncate file

Action: Creates a new entry in the current directory for the named file, or truncates the length of an existing file of the given name to zero length. The file is opened for read/wnte access.

On entry: AH = 16h
DS:DX = segment:offset of unopened FCB

Returns: AL = 00h if successful
AL = 0FFh if unsuccessful (directory full)

Notes: Notes: This call is the equivalent of 'OPENOUT' in BBC BASIC and should be used with care. By using an extended PCB and setting the appropriate bit the opened file may be assigned an attribute. To create files in other directories use function 3Ch.

Function 17h- Rename file

Action: Renames all matching files in the current directory

On entry: AH = 17h
DS:DX = Segment:offset of special FCB

Returns: AL = 0 if successful
AL = 0FFh if no match found or new filename already exists

Notes:

Function 18h - Reserved

Function 19h - Get default disc drive

Action: Returns the drive code of the current or default drive.

On entry: AH = 19h

Returns: AL = drive code (0 = A:, 1 = B: etc)

Notes: Some DOS functions use drive codes starting at 1 (e.g. function 1Ch) reserving zero for the current dnve.

Function 1Ah - Set disc transfer area address

Action: Specifies the memory area to be used for subsequent FCB operations.

On entry: AH = 1Ah
DS:DX = Segment:offset of DTA

Returns: Nothing

Notes: If this function is not used by a program, the DTA defaults to a 128 byte buffer in the PSP at 080h, the area used to hold the original command tail, which will then be destroyed by any disc activity.

It is the programmer's responsibility to ensure that the DTA is large enough for any relevant disc operation. The only exception is that DOS will abort any transfer which would wrap around within the segment.

This function *must* be called before using functions 11h, 12h, 14h or 4Fh, to ensure that DOS has a large enough scratch area when searching directories.

Function 1Bh - Get current drive allocation data

Action: Obtains selected information about the current disc drive and a pointer to the identification byte of the FAT.

On entry: AH = 1Bh

Returns: If successful
AL = Number of sectors per cluster
DS:BX = Segment:offset of FAT identification byte
CX = Size in bytes of physical disc sector
DX = Number of clusters for the drive
If unsuccessful (invalid drive) AL = 0FFh

Notes: DS:BX points only to the FAT information byte. To read the contents of the FAT into memory use INT 25h.

To obtain infomation about discs other than the default drive use function 1Ch. See also function 36h which returns similar data.

Function 1Ch - Get alloc. data for specified drive

Action: Action: As for Function 1Bh, but any drive can be specified.

On entry: AH = 1Ch
DL = Drive code (NOTE 0 = current, 1 = A:, 2 = B:)

Returns: If successful
AL = Number of sectors per cluster
DS.BX = Segment:offset of FAT identification byte
CX = Size in bytes of physical disc sector
DX = Number of clusters for the specified drive
If unsuccessful (invalid drive or critical error)
AL = 0FFh

Notes: Except for the ability to specify a drive this call is the equivalent of Function 1Bh.

Functions 1Dh to 20h - Reserved

Function 2lh - Random read

Action: Reads a selected record from an opened file.

On entry: AH = 21h
DS:DX = Segment:offset of previously opened FCB

Returns: AL = 0 if successful
AL = 1 if end of file reached
AL = 2 if segment wrap occurs
AL = 3 if partial record read at end of file

Notes: The record is read into memory at the DTA address spedfied by the most recent call to Function 1Ah. The size of the block read is specified by the record size field in the FCB.

If the size of the DTA and the record are such that segment wrap occurs, the call fails with a return code of 2. If a partial record read occurs the remaining space is padded with zeros. The current file pointer is not advanced after this function.

Function 22h - Random write

Action:

On entry: AH = 22h
DS:DX = Segment:offset of previously opened FCB

Returns: AL = 0 if successful
AL = 1 if disc full
AL = 2 if segment wrap occurs

Notes: The record is written from memory at the DTA address specified by the most recent call to Function 1Ah. The size of the block written is specified by the record size field in the FCB. If the size of the record and the location of the DTA are such that segment wrap occurs, the call fails with a return code of 2.

Function 23h - Get file size in records

Action: Returns the record count of a matching file.

On entry: AH = 23h
DS:DX = Segment:offset of unopened FCB

Returns: AL = 0 if matching file found
AL = 0FFh if no matching file found

Notes: Before using this call you must set an appropriate record size in the FCB's record size field. After the call the random record field is set to the record count of the specified file.

Function 24h - Set random record number

Action: Sets the random record field of an FCB to correspond to the current file position as recorded in the opened FCB.

On entry: AH = 24h
DS:DX = segment:offset of previously opened FCB

Returns: Nothing. The random record field in the FCB is updated

Notes: This function is used to change from sequential to random I/O file access.

Function 25h - Set interrupt vector

Action: Initialises an interrupt vector to point to the supplied address.

On entry: AH = 25h
AL = Interrupt number
DS:DX = segment:offset of new vector address

Returns: Nothing

Notes: This is the approved way to amend interrupt vector contents.

Before changing the contents of a vector, Function 35h should be used to obtain the original entry, which should be re-instated when your code terminates. The only exceptions to this rule are interrupt vectors 22h to 24h, which are automatically restored from the PSP on program termination.

Function 26h - Create program segment prefix

Action: Copies the PSP of the current program to a specified segment address in free memory, then updates the new PSP to make it usable by a new program.

On entry: AH = 26h
DX = Segment for new PSP

Returns: Nothing

Notes: This call has been rendered obsolete by EXEC, Function 4Bh in DOS 2.0 and later and should no longer be used.

Function 27h - Random block read

Action: Reads one or more sequential records from an open file starting at the file's current record position

On entry:
AH = 27h
CX = Number of records to be read
DS.DX = Segment:offset of previously opened FCB

Returns:
AL = 0 if all requested records read
AL = 1 if end of file
AL = 2 if segment wrap
AL = 3 if partial record read at end of file
CX = Actual number of records read

Notes: The records are read into memory at the DTA address specified by the most recent call to Function 1Ah. The size and number of blocks read is specified by the random record and the record size field in the FCB.

If the size and. location of the DTA and the number of records to be read are such that segment wrap occurs, the call fails with a return code of 2, possibly after reading one or more records. if a partial record read occurs at the end of the file the remaining record space is padded with zeros The random record, current block and current record fields are updated after this function. The call is similar to Function 21h except that multiple blocks are permitted.

Function 28h - Random block write

Action: Write one or more sequential records to an open file starting at the file's current record position.

On entry:
AH = 28h
CX = Number of records to be wriflen or zero (see notes)
DS:DX = Segment:offset of previously opened FCB

Returns:
AL = 0 if all requested records written
AL = 1 if disc full
AL = 2 if segment wrap
CX = Actual number of words written

Notes: The records are written from memory at the DTA address specified by the most recent call to Function 1Ah.

If the size and location of the DTA and the number of records to be written are such that segment wrap occurs the call fails with a return code of 2, possibly after writing one or more records.

The random record, current block and current record fields are updated after this function. The call is similar to Function 21h except that multiple records may be read.

If the call is executed with zero in CX no data is written, but the file length is set to the value implied by the current random record field and the record size.

This call is similar to function 22h, except that multiple records may be written.

Function 29h - Parse filename

Action: Parses a text string into the various fields of an FCB.

On entry: AH = 29h
AL = flags to control parsing, as follows:
Bit 3: If set, the extension field in an existing FCB will be modified only if an extension is specified in the string being parsed. If clear, the extension field will always be modified. If no extension is specified the FCB extension field will be set to blanks (20h).

Bit 2: if set, the filename field in an existing FCB will be modified only if a filename is specified in the string being parsed. if clear, the filename field will always be modified. If no filename is specified the FCB filename field will be set to blanks (20h).

Bit 1: if set, the drive ID byte in the resulting FCB will be modified only if a drive ID is specified in the string being parsed. if clear, the drive ID will always be modified. If no drive is specified the drive ID in the resulting FCB will be set to zero, (default).

Bit 0: if set, leading separators will be ignored.

DS:SI = Segment:offset of text string
ES:DI = Segment:offset of FCB

Returns: AL = 0 if no global (wildcard) characters encountered
AL = 1 if parsed string contains global characters
AL = 0FFh if drive specifier is invalid
DS:SI = Segment:offset of first character after parsed name
ES:DI = Segment:offset of formatted, unopened FCB

Notes: Permitted separators are: - : . ; , = + TAB and SPACE

The call regards all control characters, the above separators (when trailing) and < > I / " [and [as terninating characters. If no valid filename is present the contents of ES:DI+1 is a blank. If a '*' occurs in an extension, it and all remaining characters in the FCB are set to '?'. This function (and FCBs in general) cannot be used with file specifications which include a path.

Function 2Ah - Get system date

Action: Returns the system day, month and year plus the day of the week.

On entry: AH = 2Ah

Returns: CX = year (1980 to 2099)
DH = month (1 to 12)
DL = day of month(1 to 31)
AL = day number in week (0 to 6 = Sunday to Saturday)

Notes: The format of the registers returned by this call is the same as that required by Function 2Bh. Although shown above as decimal values for clarity, all values are in hex.

Function 2Bh - Set system date

Action: Reset the date held in the system clock

On entry: AH = 2Bh
CX = year (1980 to 2099)
DH = month (1 to 12)
DL = day of month (1 to 31)

Returns: AL = 0 if successful
AL = if invalid date supplied (no change)

Notes: The system time of day is unaffected by this call.

Function 2Ch - Get system time

Action: Returns the time of day as held by the system clock.

On entry: AH = 2Ch

Returns: CH = hour(0 to 23)
CL = minute (0 to 59)
DH = second (0 to 59)
DL = centiseconds (0 to 99)

Notes: The register format returned by this call is the same as that required by Function 2Dh.

Function 2Dh - Set system time

Action: Sets the time of day held in the system clock.

On entry: AH = 2Dh
CH = hour(0 to 23)
CL = minute (0 to 59)
DH = second (0 to 59)
DL = centiseconds (0 to 99)

Returns: AL = 0 if time reset successfully
AL = 0FFh if invalid time supplied (no change)

Notes:

Function 2Eh - Set verify flag

Action: Sets or cancels the system read after write verify flag.

On entry: AH = 2Fh
AL = 0 to turn verification off
AL = 1 to turn verification on
DL should be set to zero

Returns: Nothing

Notes: This call is intended to provide increased data integrity by allowing read after write verification on all data written to disc.

It is the equivalent to the DOS command VERIFY and, like the manual command, is non-functional in DOS Plus 2.1.

Function 2Fh - Get DTA address

Action: Returns the segment:offset of the current DTA for read/write operations.

On entry: AH = 2Fh

Returns: ES:BX = Segment.offset of current DTA

Notes: If the DTA has not been explicitly set it defaults to 080h in the PSP.

Function 30h - Get DOS version

Action: Returns the version number of the Operating System.

On entry: AH = 30h

Returns: AL = Major version number (e.g. 2.10 = 2)
AH = Minor version number (e.g. 2.10 = 0Ah)

Notes: In the 512 this call returns 2.11.

Function 31h - Terminate & stay resident (keep)

Action: Terminate program execution but leave memory allocated.

On entry: AH = 31h
AL = Return code
DX = memory size to be reserved (in paragraphs)

Returns: Nothing

Notes: Notes: TSR programs are usually re-entered by having previously re-directed an interrupt vector to point back into the code. In this way the program may be re-entered as a result of normal system activity, or as a result of an explicit call by another program.

Function 32h - Get disc info (Undocumented call)

Action: Returns the pointer to the specified disc drive information block

On entry:
AH = 32h
DL = drive number (0 = default, 1 = A: etc)

Returns:
AL = 0 if drive valid
DS:BX = segment:offset of disk information block
AL = 0FFh if invalid drive number

Notes: This call is unofficial and is not supported by DOS Plus.

Function 33h Get or set CTRL-BREAK flag

Action: Returns or sets the CTRL-BREAK action

On entry:
AH = 33h
If getting the status of the flag: AL = 0
If setting the flag: AL = 1
DL = 0 to turn CTRL-BREAK checking off
DL = 1 to turn CTRL-BREAK checking on

Returns:
DL = 0 if CTRL-BREAK checking off
DL = 1 if CTRL-BREAK checking on

Notes: Notes: This command is the functional equivalent of the DOS command BREAK. Like that command, in the 512 this call is non-functional.

Function 34h - Find active byte (Undocumented)

Action: Returns the number of currently active processes

On entry:
AH = 34h

Returns: ES:BX = Segment:offset of active byte address

Notes: This call is unofficial and is not supported by DOS Plus. In MSDOS it is mainly used by TRS.

Function 35h - Get interrupt vector

Action: Returns the segment:offset of a nominated vector.

On entry: AH = 35h
AL = interrupt number

Returns: ES:BX = Segment offset of interrupt vector contents

Notes: This is the approved way to read interrupt vector contents. The original contents of the vector, after storage, can be amended by a call to function 25h.

Function 36h - Get free disc space

Action: Gives the number of free clusters on a specified disc.

On entry: AH = 36h
DL = Drive code (0 = default, 1 = A: etc)

Returns: If specified drive valid:
AX = Sectors per cluster
BX = number of available clusters
CX = bytes per sector
DX = Clusters (allocation units) per drive
If specified drive is invalid:
AX = 0FFFFh

Notes: Similar information is returned by functions 1Bh and 10h

Function 37h- Reserved

Function 38h - Get country information

Action: Reading geographically variable system constants.

On entry: AH = 38h
AL = 0
DS.DX = Segment:offset of a 32 byte control block

Returns: If unsuccessful Carry flag set AX = Error code

If successful the 32 byte block contents are as follows.

5 byte currency symbol add, null terminated
2 byte thousands separator, null terminated
2 byte decimal separator, null terminated
2 byte date separator, null terminated

1 byte bit field currency format
Bit 0: clear if currency symbol precedes value, set if value precedes currency symbol
Bit 1: clear if no space between the value and the currency symbol, Set if a space required

1 byte time format
Bit 0: clear for 12 hour clock, set for 24 hour clock

2 words for case map call address
2 bytes data list separator, null terminated
5 words reserved

Notes: Unlike its MSDOS counterpart, this call does not permit the stored information to be amended.

Function 39h - Create subdirectory

Action: Creates a new subdirectory using the specified drive and path data.

On entry: AH = 39h
DS:DX = Segment:offset of ASCIIZ path specification

Returns: Carry: clear if successful, set if unsuccessful, when:
AX = 3 if path not found, 5 if access denied

Notes: The function fails if:

1. Any part of the pathname does not exist.
2. A directory of the same name already exists in the same path.
3. The parent directory is the root and it is full.

Function 3Ah - Delete subdirectory

Action: Deletes a specified subdirectory.

On entry: AH = 3Ah
DS:DX = Segment:offset of ASCIIZ path specification

Returns: Carry clear if succesful
Carry set if unsuccessful, when:
AX = Error code as follows:
3: path not found
5: access denied
6: current directory
16: directory contains files

Notes: The function fails if:

1. Any part of the pathname does not exist.
2. The specified directory is the curmnt directory.
3. The specified directory still contains files.

Function 3Bh - Set current directory

Action: Sets the specified directory to be the current directory.

On entry: AH = 3Bh
DS.DX = Segment:offset of ASCIIZ path specification

Returns: Carry clear if successful,
Set if unsuccessful, when:
AX = Error code 3: path not found

Notes: The call fails if any part of the pathname does not exist.

Commonly the current directory is ascertained by a call to function 47h, then stored by a program so the original current directory can be reset on completion of operations.

Function 3Ch - Create or truncate file

Action: Creates a new entry in the specified directory on the specified drive for the named file, or truncates the length of an existing file of the given name and path specification to zero length. The file is opened for read/write access and a 16 bit handle is returned for future access.

On entry: AH = 3Ch
CX = File attribute:
0 = normal
1 = read only,
2 = hidden,
3 = system
DS:DX = Segment:offset of ASCIIZ file specification

Returns: Carry clear if successful: AX = file handle
Carry set if unsuccessful: AX = Error code as follows
3: Path not found
4: No handle available (too many files)
5: Access denied

Notes: The function fails if:

1. Any part of the pathname does not exist.
2. A file of the same name afready exists in the same path with the read only attribute set.
3. The parent directory is the root and it is full.

Function 3Dh - Open file

Action: Opens a file in the specified or default directory on the specified drive for the named file. A 16-bit handle is returned for future access.

On entry: AH = 3Dh
AL = access mode, where:
0 = read access
1 = write access
2 = read/write access
All other bits off
DS.DX = Segment:offset of ASCIIZ file specification

Returns: Carry clear if successful: AX = file handle
Carry set if unsuccessful AX = Error code as follows
2: File not found
3: Path does not exist
4: No handle available (too many files)
5: Access denied
0Ch: Access code invalid

Notes: Any normal system or hidden file with a matching name will be opened by this function. On return the read/write pointer is set to zero, the start of the file.

The call fails if:
1. Any part of the path does not exist.
2. A read only file is opened for write or read/write access.

Function 3Eh - Close file

Action: Closes a successfully opened file. All buffers are flushed to disc and the file handle is freed for re-use. If the file was modified, the date and time stamps and the file length are updated in the directory entry.

On entry: AH = 3Eh
BX = the file handle

Returns: Carry clear if successful, set if failed, when AX = error code 6, invalid handle or not open

Notes: In MSDOS calling this function with a handle of zero closes the standard input device! DOS Plus does not suffer from this bug.

Function 3Fh - Read file or device

Action: Reads a specified number of bytes from a successfully opened file or device.

On entry: AH = 3Fh
BX = File handle
CX = Nurnber of bytes to be read
DS:DX = Segment:offset of buffer area

Returns: Carry clear if successful
AX = number of bytes read
AX = 0 means that EOF was already reached.
Carry set if failed, and AX = Error code as follows:
5: Access denied
6: Invalid handle or not open

Notes: If reading frorn a character device in cooked mode, a maximum of one line will be read, as a carriage return (0Dh) is treated as a record terminator.

If the carry flag is clear and AX is less than CX a partial record was read or there was an error.

Function 40h - Write to file or device

Action: Reads a specified number of bytes from a successfully opened file or device.

On entry:
AH = 40h
BX = File handle
CX = Number of bytes to be written
DS:DX = Segment:offset of buffer area

Returns:
Carry clear if successful, when
AX = number of bytes written
AX = 0 means the disc is full
Carry set if failed, when:
AX = Error code as follows:
5: Access denied
6: Invalid handle or not open

Notes: If the carry flag is clear and AX is less than CX, this means that a partial record was written or there was an error.

Function 41h - Delete file

Action: Deletes a file from the specified or default disc and directory.

On entry:
AH = 41h
DS.DX = Segment:offset of ASCIIZ file specification

Returns: Carry clear if successful, set if failed, when AX = Error code as follows:
2: File not found
5: Access denied

Notes: This deletes a file by deleting its directory entry. The ASCIIZ string specifying the file may not include wildcards. The function fails if:

1. Any part of the path does not exist.
2. The specified file has a read-only attribute.

Function 42h- Move file pointer

Action: Sets the file pointer to the specified position relative to the start or end of the file, or to the current pointer location.

On entry: AH = 42h,
AL = method code as follows.
0: Absolute byte offset from start of the file. (Always +ve double integer)
1: Byte offset from current location (+ve or -ve double integer)
2: Byte offset from the end of the file (+ve or -ve double integer)
BX = File handle
CX = MSB of offset
DX = LSB of offset

Returns: Carry clear if successful
DX = MSB of new pointer location
AX = LSB of new pointer location
Carry set if failed, when AX = Error code as follows:
1: function number invalid
6: invalid handle or not open

Notes: The method code determines the relative base for the pointer move, which uses a 32 bit integer to set the new location.

Method 2, if called with an offset of zero returns the length of the file as the new pointer value.

Method 1 or 2 can set the pointer to a location before the start of the file, but an error will occur if a subsequent attempt is made to use this pointer location.

For all methods (and results) the returned pointer location is always an absolute byte offset from the start of the file.

Function 43h - Get or set file attributes

Action: Obtains or sets the attributes of the specified file.

On entry: AH = 43h
AL = 0, get file attribute or AL = 1, set file attribute
CX = new attribnte (when AL = 1) as follows:
bit 5 = archive
bit 2 = system
bit 1 = hidden
bit 0 = read only
DS:DX = Segment:offset of ASCIIZ file specification

Returns: Carry clear if successful,
CX = attribute (when AL was 1)
Carry set if failed, AX = Error code as follows:
1: function code invalid
2: file not found
3: path not found or file not found
5: attribute cannot be changed

Notes: This function cannot be used to set a file's volume label bit (3), or the sub-directory bit (4). These may only be changed by using an extended ECB.

Function 44h- Device driver control (IOCTL)

Action: Passes information directly between an application and a device driver.

On entry: AH = 44h
AL = 6 get input status
AL = 7 - get output status
BX = handle

Returns: Carry clear if successful
AL = 0: For device or output file = Not Ready. For input file = Pointer at EOF
AL = FFh: For device, input or output file = Ready
Carry set if failed, when AX = Error code as follows:
1: AL not 6 or 7
5: Acces denied
6: Invalid handle or not open
0Dh: Invalid data

Notes: This call is a partial implementation of the full MS/PCDOS flindion, as it only supports status checkng in the 512.

Function 45h - Duplicate handle

Action: Returns a second handle for a file or device which has already been successfully opened.

On entry: AH = 45h
BX = existing file or device handle

Returns: Carry clear if successful
AX = new handle
Carry set if failed, when
AX = Error code as follows:
4: No handle available
6: Handle invalid or not open

Notes: If the file pointer attached to one handle is implicitly moved by a seek, read or write, the pointer for the other handle is also moved.

The purpose of this call is to force directory updating for a file without having to close it (and then re-open it). After obtaiidng the new handle, the logical file associated with it is closed by function 3Eh, forcing a directory update, but leaving the original handle available for further input/output operations.

Function 46h - Force duplicate of handle

Action: Given two handles, make the second refer to the same file at the same point as the first.

On entry: AH = 46h
BX = first file handle
CX = second file handle

Returns: Carry clear if successful, set if failed when AX = Error code
4: No handles available
6: Invalid handle or not open

Notes: If the handle specified in CX already refers to an open file, that file is closed before this function is performed.

After the call, if the file pointer attached to one handle is implicitly moved by a seek, read or write, the pointer for the other handle is also moved.

Function 47h - Get current directory

Action: Obtains the ASCIIZ string of the current directory's path.

On entry: AH = 47h
DL = Drive code (0 = default, 1 = A:, etc)
DS:SI = Segment:offset of 64byte scratch buffer

Returns: Carry clear if successful, full directory pathnarne is placed in the buffer.
Carry set if failed when AX = Error code as follows:
4: No handle available
6: Drive specification invalid

Notes: The returned pathnarne does not mdude the drive ID, nor is it prefixed with a '\'. It is terminated by a null byte, therefore if this call is issued from the root directory, the first byte in the buffer will be zero.

Function 48h - Allocate memory

Action: Allocates a block of memory and returns a pointer to the start of the area.

On entry: AH = 48h
BX Number of paragraphs of memory needed

Returns: Carry clear if successful when AX = first segment of allocated block
Carry set if failed when AX = Error code as follows:
7: memory control blocks destroyed
8: insufficient memory, BX = size of largest available block

Notes: If the call succeeds AX:0000 points to the start of the block.

When a COM file loads, it conceptually owns all the remainder of memory from its PSP upwards. This call may be used to lirnit a program's memory allocation to its immediate requirements.

Function 49h - Release memory

Action: Releases memory to make it available to other programs.

On entry: AH = 49h
BX = New requested block size in paragraphs
ES = Start segment of block to be modified

Returns: Carry clear if successful, set if failed when AX = Error code as follows:
7 = memory control blocks destroyed
8 = insufficient memory
9 = incorrect segment in ES
BX = Size of largest available block

Notes: This call modifies the size of a block of memory previously allocated through Function 48h. The call *must* be used by a COM program to release all unused memory before spawning by means of EXEC, Function 4Bh. EXE programs may also use this call.

Function 4Bh - Execute program

Action: Loads a program for execution under the control of an existing program. By means of altering the INT 22h to 24h vectors, the calling prograrn can ensure that, on termination of the called program, control returns to itself.

On entry:
AH = 4Bh
AL = 0: Load and execute a program
AL = 3: Load an overlay
DS.DX = segment:offset of the ASCIIZ pathname
ES:BX = Segment:offset of the parameter block
Parameter block bytes:
0-1: Segment pointer to envimmnemnt block
2-3: Offset of command tail
4-5: Segment of command tail
6-7: Offset of the first FCB to be copied to new PSP+5Ch
8-9: Segment of the first FCB
Ah-Bh Offset of the second FCB to be copied to new PSP+6Ch
Ch-Dh Segment of the second FCB

Returns: Carry clear if successful. On return *all* register contents are destroyed, including the stack pointers.
Carry set if failed when AX = Error code as follows:
1: Function invalid
2: File not found or path invalid
5: Access denied
8: Insufficient memory
0Ah: Environment invalid
0Bh Format invalid

Notes: To protect the caller's register contents they should be pushed on the stack and the stack pointers, SS:SP stored in a known location before the call. On return these should be retrieved immediately with interrupts disabled to prevent interrupts occuring before stack integrity is regained.

The ASCIIZ pathnarne must include the drive, path and filename of the program to be loaded. The environment block must be paragraph-aligned and consist of one or more ASCIIZ strings, all terminated by an extra zero byte.

Command tails are in the usual format for PSPs, that is, a count byte and the command tail terminated by a carriage return, which is not included in the count.

All active handles, devices and I/O redirection assignments in the calling program are inherited by the executed program.

Function 4Ch - Terminate program with return code

Action: Terminates execution of a program with return to `COMMAND.COM` or a calling routine, passing back a return code. Allocated memory is freed, vectors for interrupts 22h to 24h are restored from the PSP and all file buffers are flushed to media. All files are closed and directories are updated.

On entry: AH = 4Ch
AL = Return code (Error level)

Returns: Nothing

Notes: This is the approved method of terminating program execution. It is the only way that does not rely on the contents of any segment register and is thus the simplest exit, particularly for EXE files.

The return code can be interrogated by a calling program by means of function 4Dh, and by the batd' file commands, `IF ERRORLEVEL`. Conventionally a return code of zero indicates success, any other value failure. Standard DOS return codes are:

0: Successful operation
1: CTRL-BREAK termination
2: Critical error terrnination
3: Terminated and stayed resident

Return code values can be used at the discretion of the programmer (avoiding codes 1 to

3), thus both success or a wide range of failure types may be indicated by varying the code. For the return of result codes to the caller by an EXEced program a better method is to use other registers, but only the contents of register AL are significant to the batch command ERRORLEVEL.

Function 4Dh - Get return code

Action: Used by a parent task to obtain the return code of a program executed by a call to function 4Bh.

On entry: AH = 4Dh

Returns: AH = Exit type:
0 Normal termination
1: CTRL-C termination
2: Terminated by critical device error
3: Terminated by a call to function 31h
AL = Return code

Notes: This call is a 'one-shot' function, that is, it will yield the return code of a called program once only.

Function 4Eh - Search for first match

Action: Searches the default or specified drive:directory for the first occurrence of a matching filename.

On entry: AH = 4Eh

Returns: CX = Attribute to use in search
DS:DX = Segment:offset of ASCIIZ file specification

Notes:

CX = 0 if successful. The current DTA is filled as follows:

Bytes
0-20: Reserved for use by DOS in subsequent calls
21: Attribute of matched file
22-23: File time stamp
24-25: File date stamp

26-27: least significant word of file size
28-29: Most significant word of file size
30-42:: Filenarne.extension in ASCIIZ string form
Carry set if failed, AX = Error code as follows
02h path invalid
12h: no rnatching directory entry

This call assumes the DTA has heat set up by a successful call to function 1Ah.

Both wildcards (? and *) are permitted in filenames, but only the first matching name is returned.

if the attribute in CX is zero only normal files are searched. If the volume label attribute bit is set only volume labels are returned. For all other attribute settings, (i.e, hidden, system or directory) those files and normal files are searched

Function 4Fh - Search for next match

Action:	Searches for the next matching file after a previously successful call to Function 4Eh.
On entry:	AH = 4Fh
Returns:	Carry clear if successful The current DTA is filled as follows:- Bytes 0-20 : reserved for use by DOS in subsequent calls 21: Attribute of matched file 22-23: File time stamp 24-25: File date stamp 26-27: least significant word of file size 28-29: Most significant word of file size 30-42: Filenarne.extension in ASCIIZ string form Carry set if failed, AX = Error code 12h: no matching directory entry
Notes:	When used this call requires a DTA containing returned data from a previously successful call to function 4Eh or 4Fh. Use of function 4Fh is only relevant when the original file spedacation used in function 4Eh included at least one wildcard.

Function 50h - Get disc information (Undocumented call)

Action: Returns a pointer to the disc information block.

On entry: AH = 50h
DL = drive number (0 = default, 1 = A: etc)

Returns: AL = 0 if drive exists
DS:BX = Segment:offset of disc information block
AL = 0FFh if failed

Notes: This call is unofficial and is not supported by DOS Plus.

Function 51h - Reserved

Function 52h- Reserved

Function 53h - Reserved

Function 54h - Get verify flag

Action: Reads the current state of the verify flag.

On entry: AH = 54h

Returns: AL = 0 if verify off
AL = 1 if verify on

Notes: This call is the countepart of function 2Eh. In DOS Plus AL is always returned as zero.

Function 55h - Reserved

Fundion 56h - Rename file

Action: Renames a file and or moves its directory entry to a different directory on the disc.

On entry: AH = 55h
DS:DX = Segment:offset of current ASCIIZ filename ES:DI = Segment:offset of new ASCIIZ filename

Returns: Carry clear if successful
Carry set if failed, AX = Error code as follows:
2: File not found
3 : path not found or the file doesn't exist
5: Access denied
11h : new name not same device

Notes: The call fails if:

1. Any part of the pathnarne does not exist.
2. The new filename refers to a different disc.
3. The new name is in the root directory, which is full.
4. A file with the new path and name already exist.

Function 57h - Get or set file date and time

Action:

On entry: AH = 57h
BX = file handle
AL = 0 to get date and time
AL = 1 to set date and time
CX = time:
bits 9-0Fh = hours (0-23)
bits 5-8 = minutes (0-59)
bits 0-4 = No. of two-second increments (0-29)
DX = date.
bits 9 - = year relative to 1980 (0-119, i.e. 1980-2099)
bits 5-8 = month of year(1 to 12)
bits 0-4 = day of month(1 to 31)

Returns: Carry clear if successful
If getting date and time:
CX = time (in format above)
DX = date (in format above)
Carry set if failed, AX = Error code as follows:
1 - function code invalid
6 - file handle invalid

Notes: The file must have been previously opened or created by a call to function 3Ch or 3Dh.

For simplicity the date and time formats are shown above in the sequence they are stored in the directory.

This completes the list of INT 21h function codes valid in DOS PIus 2.1.

Functions 58h and above are only available in versions of MSDOS later than 2.11, and with the exception of function 63, later than version 3.0.

Interrupts 22h through 24 are not user callable and are therefore not documented. (See chapter 8).

INT 25h Absolute disc read

Action: Provides a direct link into the XIOS to allow data to be read from a specified memory location to disc, starting at a specified logical disc sector into a specified memory location.

On entry:
AL = Drive number (0 = A:, 1 = B: etc)
CX = number of sectors to read
DX = start sector relative (logical) number
DS:BX = Segment offset of DTA

Returns:
Carry clear if successful
Carry set if failed, AX = Error code, as follows
AL=
00 - write protect
01 -unknown unit
02 - drive not ready
03 - unknowncommand
04 - data error (CRC failed)
05 - Bad request structure length
06 - seek error
07 - unknown media type
08 - sector not found
09 - printer out (of paper)
0Ah - write fault
0Bh - read fault
0Ch - general failure

AH =
80h - attachment failed to respond
40h - seek operation failed
20h - controller failed
10h - data error (CRC failed)
08 - DMAfailure
04 - requested sector not found
03 - write protect fault
02 - bad address mark
01 - bad command

Notes: All register contents except those of the segment registers may be destroyed.

When this call returns, the CPU flags, originally pushed onto the stack, are still on the stack. The calling program should issue a `POPF` instruction or `ADD SP,2` to prevent uncontrolled stack growth and to make earlier data pushed on the stack accessible correctly.

Logical disc sectors begin at one (unlike BBC media), so the total number of sectors per disc is also the number of the last sector. Sectors are read logically by this call so, although it performs as a hardware command, formatting skew factors are catered for.

INT 26h Absolute disc write

Action: Provides a direct link into the XIOS to allow data to be written from a specified memory location to disc, starting at a specified logical disc sector.

On entry:
AL = Drive number (0 = A:, 1 = B: etc)
CX = number of sectors to write
DX = start sector relative (logical) number
DS:BX = Segment offset of DTA

Returns:
Carry clear if successful
Carry set if failed, AX = Error code, as shown for `INT 25h`.

Notes: See `INT 25h`. All comments apply.

INT 27h Terminate and stay resident

Action: Terminates execution of the current program but reserves part or all of its memory so that it will not be overwritten by the subsequent programs. The vectors for interrupts 22h to 24h are restored from the program's PSP. Open files are *not* closed.

On entry: DX = Offset of last byte+1 (relative to the PSP) to be protected
CS = Segment address of PSP

Returns: Nothing

Notes: This interrupt is typically used by programs which are to be re-entered by redirected interrupt vectors, but its use should be strictly avoided unless compatibility with MSDOS v.1 is vital.

The maximum amount of memory that can be theoretically reserved by this function is one segment (64k bytes), but *must not* be used in .EXE programs. In addition the call does not work correctly when DX contains a value in the range 08000h to 0FFFFh (32 to 64k bytes). The high bit is discarded by MSDOS, resulting in a memory reservation of up to 32K bytes less than expected.

The recommended call instead of this one is INT 21h function 31h, which works correctly and also allows any amount of memory to be reserved.

*Original text says "*0FFF1h to 0FFFFh (32 to 64k bytes)*". I believe this may be a mistake - K*

BDOS interrupts - INT E0h

The following is a guide to the BDOS interrupts provided by DOS Plus 2.1 in the 512. These calls are also referred to variously as CP/M calls or DOS Plus system calls. In all cases they are made by calling DOS interrupt 0E0h (224 decimal), with the required BDOS function always held in register CL. In all cases the call is immediately passed to the kernel by the DOS emulator, for execution.

By convention, in DOS Plus and CP/M programming reference manuals, the call function and parameters are often given in decimal. Because all 80186 register and memory values are in hex, all debuggers and monitors report or display in hex which is used exclusively here.

As BDOS calls access the kernel by-passing the DOS emulator, they conform to CP/M call standards which use different register assignments to DOS calls. BDOS cans use the general register scheme shown below.

	Parameter Type	Register
On entry	Function code	CL
	Byte parameter or	DL
	Word parameter or	DX
	Memory address Segment: Offset:	DS DX
On exit	Byte returned or	AL
	Word returned or	AX
	Memory address Segment Offset	ES AX
	Word return	EX (usually the same as AX)
	Error code if failed	CX (Zero if successful)
	Return code	AH

If a call is successful AL and CX return zero. In the event of a failure, AL normally contains 0FFh and AH may contain a reason code, particularly for filing system operations. In addition CX is used to return

error codes, which also indicate the reason for failure if the cause is one of the following:

Value	Meaning
2	Illegal function
3	Insufficient memory
4	Illegal system flag number
5	Flag overrun
6	Flag underrun
0Ch	No free process environments
12h	No available memory descriptors
17h	Illegal drive number
18h	Illegal filename
19h	Illegal file type
1Dh	Error reading file
1Eh	Could not open file
20h	Caller is not owner of resource (illegal access)
21h	No code group descriptor in command header
26h	Illegal password
29h	Error during load time fixups
2Ah	Error loading RSX module
2Bh	Illegal parameter
2Eh	8087 in use
2Fh	No tick interrupt

As can be seen from this general outline of BDOS calls, register contents are more often changed by the call than in DOS calls, therefore more care should be exercised in protecting such data as must be retained when these BDOS calls are used. In addition, BDOS calls tend to operate at a lower level and therefore are slightly more complex to implement than their 'sanitised' MSDOS counterparts.

For all that, BDOS calls and CP/M in general offer facilities which cannot be matched by DOS calls, including those calls provided in later versions. As a result, judicious use of BDOS calls can considerably extend the range of facilities available in DOS Plus programs over those provided by MSDOS or PCDOS calls if exploited to the full.

The following code is an example of a 'BDOS call only' program termination similar in purpose to that shown in the preceding appendix, which used DOS INT 21h function 4Ch. Contrast this example with the DOS program code: in this case two calls are used. The first sets the program return code (which is optional in CP/M), while the second call invokes the program terminate handler routine.

The code example shown here assumes that the value, held in the memory variable `Return_Code`, will have been set elsewhere in the program during execution if the program was unsuccessful and the final return code is to be other than zero (the default value).

```
; constant equates
Prog_code               equ 06ch        ; BDOS set return code
Prog_exit               equ 08fh        ; BDOS terminate call
BDOS                    equ 0e0h        ; BDOS call interrupt No
CSEG
        ORG 0100
start:
; program code follows here
        .........
        .........
        .........
set_result:                             ; Sets up program return code
        mov dx, [Return_code] W         ; Set actual return code value
        mov cl, Prog_code               ; set up BDOS function 06Ch
        int BDOS_Call                   ; INT 0E0h to set return code
exit:                                   ; Final exit point
        mov dl, 0                       ; Free all memory
        mov cl, Prog_exit               ; set up terminate process
        int BDOS                        ; and exit
; variable data
Return_Code     dw ?                    ; Default zero means OK
CODE ends
END
```

The following lists show the complete range of BDOS calls which are valid for the 512. Although these calls can be mixed with DOS interrupt calls in COM, CMD and EXE programs, it should be remembered that only BDOS calls are valid for (the portions of) programs destined to enter the background, or which are to be converted to RSXs.

Function 00h: Terminate program

Action:	Terminates the calling process and returns control to the system.
On entry:	CL = 0 DL = Abort code: 0 = Release all memory 1 = Retain memory allocation
Returns:	Nothing

Notes: This call terminates program execution and optionally releases or retains the memory allocated to the program and its buffers.

If a two-byte program return code is to be set, a prior call to function 6Ch should be made.

This call is exactly the same as function 8Fh.

Function 01h: Read a character

Action: This call reads the next character from the logical console input device 'CONIN'.

On entry: CL = 1

Returns: AL = 8 bit data, BL = AL

Notes: If the character read is a graphics character, a carriage return, line feed or backspace, it is echoed to the console. All other characters are read but not echoed to CONOUT. If the console is in default mode the system intercepts and actions CTRL-C, CTRL-P, CTRL-Q and CTRL-S.

If printer output is active (CTRL-P), characters are also sent to the list device, PRN. if no input character is ready the call waits until a character is returned.

Function 02h: Write a character

Action: Outputs a character to the standard output device, CONOUT.

On entry: CL = 2, DL = 8-bit data

Returns: Nothing

Notes: CTRL-I (tabs) are expanded. If the console is in default mode CTRL-S and CTRL-Q are actioned and characters are sent to the list device if printer output (CTRL-P) is active, otherwise system action is determined by the bits in the console status setting (function 6Dh).

Function 03h: Auxiliary input

Action:	Reads a character from the AUXIN device.
On entry:	CL = 3
Returns:	AL = 8 bit data input, BL = AL
Notes:	Control does not return until a character is read.

Function 04h: Auxiliary output

Action:	Outputs a character to the AUXOUT device.
On entry:	CL = 4, AL = 8 bit data
Returns:	Nothing
Notes:	Control does not return to the calling routine if the device is ready for output.

Function 05h: Write char to the list device

Action:	Sends a character to the current list device, PRN.
On entry:	CL = 5, DL = 8 bit data
Returns:	Nothing
Notes:	If the printer is busy this call waits until the data is sent.

Function 06h: Direct (Raw) console I/0

Action:	Permits the calling process to read or write unedited and unmodified console data or to check console status.	
On entry:	CL = 6	
	DL = function requested:	0FFh -Get Input/status, or 0FEh - Get status, or 0FDh - Get input, or 8 bit character to be output
Returns:	If output:	Nothing

	If get input status.	AL = 0 if no character or AL = Character input
	If get status:	AL = 0 if no character AL = 0FFh Character ready
	If input:	AL = 8 bit data, BL = AL
Notes:	On entry, if DL = 0FFh, either the character entered is returned, or AL is set to zero to indicate no input ready (i.e. this call does not wait for input). If DL = 0FEh on entry, AL returns 0 for no input, or for input waiting. The input character is not returned until a call is made with DL = 0FDh. If the call is made with DL = 0FDh and no character is ready the system waits for input.	

Function 07h: Auxiliary input status

Action:	Checks the input status of the AUXIN device	
On entry:	CL = 7	
Returns:	AL = auxiliary input status:	0 if character not ready 0FFh if character ready
	BL = AL	
Notes:	This call does not read the input data, only the status.	

Function 08h: Auxiliary output status

Action:	Checks the input status of the AUXOUT device	
On entry:	CL = 8	
Returns:	AL = auxiliary output status:	0 if not ready for output 0FFh if ready for output
	BL = AL	
Notes:	Output is not performed by this call, only the status is checked.	

Function 09h: String output

Action: Writes a string to the output console, CONOUT.

On entry: CL = 9, DS:DX = segment:offset of string

Returns: Nothing

Notes: The string must be terminated by the default delimiter character, normally $, (24h), which is not transmitted. The default delimiter can be set to any character by a call to function 6Eh. Any ASCII codes can be embedded within the string, including control characters, which are actioned.

Function 0Ah: Read an edited line

Action: Reads an edited line from the input device CONIN, up to and including an ASCII carriage return (0Dh), placing the data in a user-defined buffer.

On entry: CL = 0Ah, DS:DX = segment:offset of input buffer

Returns: Nothing

Notes: The first byte of the buffer specifies the maximum number of characters it can hold (1 to 255). This value must be supplied by the user. The second and third bytes of the buffer are set to the number of characters actually read, excluding the terminating RETURN. If the buffer fills to one less than its maximum size, the bell is sounded and subsequent input is ignored.

If DX is set to 0FFFFh on entry the default DMA buffer is used, any other value points to a specified buffer offset.

Function 0Bh: Get console status

Action: Checks whether or not a character is available from the input console device, CONIN.

On entry: CL = 0Bh

Returns: AL 0 if no character available
AL = 1 if character available
BL = AL

Notes: If console mode bits 0 and 3 are set this call returns AL = 1 only if CTRL-C has been typed, all other characters are ignored. See function 6Dh.

Function 0Ch: Get BDOS version number

Action: Returns the BDOS version number as a two-byte value.

On entry: CL = 0Ch

Returns: AX = BDOS version number, as follows:
AL high nibble - version number
AL low nibble - revision number
AH high nibble - CPU type
AH low nibble - 0 for CP/M(86), 1 for DOS Plus
BX = AX

Notes: In DOS Plus 2.1, the AX value returned is 01080h.

Function 0Dh: Reset all drives

Action: Restores the status of all logged-in drives to their default settings.

On entry: CL = 0Dh

Returns: Nothing

Notes: This call does *not* update all files or disc directories. If a program fails to properly write and close any open files before issuing this call, data may be lost or directory entries may be inconsistent with the files.

All disc drives are set to read/write, the default drive is set to A: and the default DMA address is set to 080h relative to the current DMA segment address (i.e, the default in the PSP).

Function 0Eh: Set default drive

Action: Sets the specified drive to the default drive.

On entry: CL = 0Eh, DL = drive number (A: = 0, B: = 1 etc)

Returns: AL = Return code, AH = 0 or physical error code (1 or 4)
BX = AX

Notes: The possible values for DL on entry are 0 through 0Fh. If the selected drive is in the reset state, it is automatically logged in.

If an error occurs and the filing system is in default error mode, a message is displayed and the call terminates, otherwise AL is set to 0FFh and AH contains the error code.

Function 0Fh: Open a file for record access

Action: Opens a file for access and activates the FCB for that file for the current user number.

On entry: CL = 0FH, DS:DX = Segment:offset of the file control block (FCB)

Returns: If successful: AH = 0, AL = directory code
If failed: AH = physical error code (1,4,7 or 9)
AL = FFh (file not found)
BX = AX

Notes: If the call is successful the file's directory information is copied to the FCB. If the file is password protected in read mode, the password is placed in the first 8 bytes of the current DMA, or must have been previously set up as the default password. if the file is password protected in write mode and the correct password was not supplied in the DMA, interface attribute F7 is set to one (i.e. write protected).

If a file is opened for write access the access date and time stamps are updated, if applicable. If there is no physical error but the file does not exist, AL is set to 0FFh and AH is set to zero.

Function 10h: Close file

Action: Closes a file and updates the directory if the file has been modified.

On entry: CL = 10h, DS:DX = Segment offset of the FCB for the file.

	F5 interface attribute :	0 = permanent close 1 = partial close
Returns:	If successful:	AH = 0Ah = directory code
	If failed:	AH = physical error code (1,2 or 4) AL = FFh (physical error), BX = AX

Notes: This call may only be used after a file has been successfully opened and an FCB created. A partial close causes the system to update the directory, but the file remains open for further access. If the FCB has not been altered the directory is not updated. If there is no physical error but the file does not exist, AL is set to 0FFh and AH is set to zero.

Function 11h: Search for first matching file

Action:	Search for a specified filename in the current directory of the current drive.	
On entry:	CL = 11h, DS:DX = Segment:offset of the FCB	
	F5 interface attribute :	0 = permanent close 1 = partial close
Returns:	If successful:	AH = 0Ah = directory code
	If failed:	AH = physical or extended error code (1 or 4) AL = FFh (physical error), BX = AX

Notes: The calling program must set the drive field, the filename field and the file type field in the Icts~ before the call. The extent field must be set to zero.

The '?' wildcard is permitted in any filename character position or in the file type specification. If the '?' wildcard is placed in the drive field, the current drive and the current user number are searched. if wildcards are specified in the file's name or type, the first matching name is returned.

If the call is successful, the file's directory data is transferred to the FCB. If there is no physical error but the file does not exist, AL is set to 0FFh and AH is set to zero.

Function 12h: Find next file

Action: Searches the current directory in the current drive for the next matching filename after a previously successful call to function 11h.

On entry: CL = 12h

Returns: If successful: AH = 0, AL = directory code
If failed: AH = physical or extended error code (1 or 4)
AL = FFh (physical error), BX = AX

Notes: There must be no other disc related calls between the call to function 11h and this call, as the FCB is not re-specified in this call. otherwise as for function 11h.

Function 13h: Delete a file

Action: Deletes all matching files from the current directory and/or all matching XFCBs.

On entry: CL = 13h, DS:DX = Segment:offset of the FCB

F5 interface attribute :0 = standard delete
1 = delete only FCBs

Returns: If successful: AH = 0, AL = directory code
If failed: AH = physical or extended error code (1 or 4)
AL = FFh (physical error), BX = AX

Notes: The '?' wildcard is permitted. If more than one match occurs all matched items will be deleted. The file names and file types must be specified in the FCB prior to this call. If any of the files are password protected, the correct password must also have been set up.

If there is no physical error but the filename and type are not matched AL is set to 0FFh and AH is set to zero. For all operations, if any matching file fails the password check or is set to read only, the file (and its FCB) are not deleted.

Function 14h: Read records sequentially

Action: Reads the next sequential one to 128 logical 128 byte records from a file into memory beginning at the current DMA address. The system multisector count determines the number of records, defaulting to one if not set.

On entry: CL = 14h, DS:DX = Segment:offset of previously opened FCB

Returns: AL = ret. code, AH = physical error or record count, BX = AX

Notes: The call can only be made for a file previously successfully opened for input. If the read is to start from the beginning of a file, the CR field in the FCB must be set to zero.

On reading the CR field is updated to the next record position. If the CR field overflows, the system automatically opens the next extent and resets CR to zero for the next read operation. Codes of 1, 9 or 0Ah may be returned on physical error.

Function 15h: Write records sequentially

Action: Writes the next sequential one to 128 logical 128 byte records to a file from memory beginning at the current DMA address. The system multisector count determines the number of records, defaulting to one if not set.

On entry: CL = 15h, DS:DX = Segment:offset of previously opened FCB

Returns: AL = ret. code, AH = physical error or record count. BX = AX

Notes: The call can only be made for a file previously successfully opened for output. If the write is to start from the beginning of a file, the CR field in the FCB must be set to zero.

On writing, the CR field is updated to the next record position. If the CR field overflows, the system automatically opens the next extent and resets CR to zero for the next write operation. Codes of 1, 2, 9 or 0Ah are returned on physical error.

Function 16h: Create a file entry

Action: Creates a new entry in the current directory for the named file for the current user number and activates the FCB for read/write access.

On entry: CL = 16h, DS:DX = Segment:offset of unopened FCB
F6 interface attribute :0 = no password required
1 = password as set in DMA

Returns: AH = 0, AL = 0
If failed: AH = physical or ext. error code (1,2.4,8 or 9)
AL = FFh (physical error), BX = AX

Notes: The calling program must initialise the DR field, the F1 through F8 filename fields, the T1 through T3 file type fields and the extent field must be set to zero. The CR field must be set to zero, prior to writing if records are to be written sequentially from the beginning of the file.

If the file password is to be used, F6 is set to one and the password must be placed in the first eight bytes of the DMA buffer and byte nine of the DMA buffer set to password mode. A new file entry in the FCB is created and an extended FCB is created if password mode is set on a successful call. An file attributes are set to zero.

If there is no physical error but AL is returned as there is no space in the directory.

Function 17h: Rename a file

Action: Renames a matching file

On entry: CL =17h, DS:DX = Segment:offset of FCB
bytes 0 - 16 = old filename
bytes 17 - 27 = new file name
bytes 0 to 7 of DMA = password (if required)

Returns: If successful: AH = 0, AL = 0
If failed. AH = physical or ext. error code (1-4,7,8 or 9)
AL = FFh (physical error), BX = AX

Notes: The calling program must set the DR field, the F1 through F8 filename fields and the T1 through T3 file type fields to match the old filename in the DMA buffer and the old and new filenames in the FCB as shown.

If the file is password protected, the password must be placed in the first eight bytes of the correct DMA buffer, or the password must have been previously set as the default password.
If there is no physical error but AL is returned as 0FFh, the named old file does not exist.

Function 18h: Return 'Login Vector' for all logged in drives

Action: Returns the bit map of all logged-in drives

On entry: CL = 18h

Returns: AX = Login vector, BX = AX

Notes: The login vector is a 16 bit value which shows the current drives known to the system. Bit zero corresponds to drive A:, bit 1 to drive B: and so on up to 0Fh which is floating drive P:.

Each bit which is set to one represents a currently logged-in drive. A logged-in drive has had its directory read into memory and its allocation vectors built. A drive can be logged-in explicitly by a call to function 0Eh, or implicitly by a system or manual file operation which requires access to the drive.

Function 19h: Return the default drive ID

Action: Returns number of the currently selected default drive

On entry: CL = 19h

Returns: AL = drive number (0 = A:, 1 = B: etc), BL = AL

Notes: None

Function 1Ah: Set DMA buffer offset address

Action: Specifies the memory area to be used for subsequent FCB operations. (See also function 33h)

On entry: CL = 1Ah, DX = offset of DMA

Returns: Nothing

Notes: If this function is not used by a program, the DMA address defaults to a 128 byte buffer in the PSP at 080h, the area used to hold the original command tail, which will then be destroyed by any disc activity. In general it is the programmer's responsibility to ensure that the DMA is large enough for any relevant disc operation.

This function should be called before using any disc access functions if the command tail is to be preserved.

Function 1Bh: Get address of disc allocation vectors

Action: Returns the base of the allocation vector for the currently selected drive

On entry: CL = 1Bh

Returns: ES:AX = Segment:offset of allocation vector, BX = AX

Notes: The bits set to one in the allocation vector denote allocated blocks, zero denotes unallocated. The high order bit in the first byte corresponds to block zero. To ascertain the amount of free space remaining on a disc use function 2Eh.

Function 1Ch: Set the default drive to read only

Action: Sets the current drive to read only

On entry: CL = 1Ch

Returns: AL = 0 if successful, AL = 0FFh if failed

Notes: Setting a drive to read-only prevents all updates to that drive until the status is changed.

Caution: Note that, although this call is grouped with drive functions, it really acts on the current disc in the current drive. That is to say, the setting is not permanent and can be amended by simply changing the disc and issuing any disc access command, including a manual 'DIR'.

Function 1Dh: Get Read-Only drive vectors

Action: Returns the bit map of read only drives

On entry: CL = 1Dh

Returns: AX = read only vector, BX = AX

Notes: The call returns a 16 bit value in AX with bits set which correspond to drives currently set to read only. Bit zero corresponds to drive A:, bit 2 to drive B: and so on up to 0Fh which is floating drive P:.

Function 1Eh: Set file attributes

Action: Modifies a file's attributes and optionally sets its last record byte count.

On entry: CL = 1Eh, DS:DX = Segment:offset of FCB

F6 interface attribute:
0 = do not set byte count
1 = set last record byte count

Returns: If successful: AL = 0, AH = 0

If failed: AL = return code, AH = physical error (1, 2, 4, 7 or 9)

Notes: If the file is password protected, the correct password must have been placed in the DMA buffer, or defined as the default password.

Function 1Fh: Get default disc DPB address

Action: Returns the address of the BIOS resident disc parameter block address for the current drive.

On entry: 1Fh

Returns: ES:AX = Segment:offset of DPB address AX = FFFFh if error, BX = AX

Notes: The DPB contains the information concerning the actual physical characteristics of the logical drive. The contents of the DPB, therefore, may change when media is changed.

Function 20h: Get User current user number

Action: Returns current user number for CP/M media.

On entry: CL = 20h, DL = 00h to 0Fh (Set user) or FFh (get user)

Returns: AL = current user number (if get user), BL = AL

Notes: The user number is the method used in CP/M to separate files into exclusive groups, rather than directories. Files can be copied between users by means of PIP.

Function 21h: Read random records

Action: Reads a selected record from an open file.

On entry: CL = 21h
DS:DX = Segment:offset of previously opened FCB

Returns: AL = Return code
AH = Physical error or record count, BX = AX

Notes: The record is read into memory at the DMA address, the record number being indicated by the settings of the random record field in the FCB. The record number can range between zero and 262,143.

If a physical error is encountered AH contains the actual number of records read before the failure.

The file may be read sequentially after this call, but the current record field in the FCB is not advanced by the function, therefore a switch to sequential access will re-read the same record if the CR field is not changed.

Function 22h: Write random records

Action: Writes a selected record to an open file.

On entry: CL = 22h
DS:DX = Segment:offset of previously opened FCB

Returns: AL = 0 return code
AH = physical error (2, 3, 5, 6 or Ah) or record count
BX = AX

Notes: The record is written from the DMA address, the record number being indicated by the settings of the random record field in the FCB. The record number can range between zero and 262,143.
If a physical error is encountered, AH contains the actual number of records read before the failure.
The file may be written sequentially after this call, but the current record field in the FCB is not advanced by the function, therefore a switch to sequential access will re write the same record if the CR field is not changed.
However, the logical extent and the current record values for the file are updated by the call.

Function 23h: Returns the size of a file

Action: Returns the highest record number plus one of the specified file.

On entry: CL = 23h
DS:DX = Segment:offset of opened FCB

Returns: If successful: AL = if file found, AH = 0 if no error

If failed: AL = 0FFh - file not found or physical error
AH = physical or extended error code

BX = AX

Notes: Before using this call you must set the drive field, filename and filetype in the referenced FCB.

If the file has been written exclusively in sequential mode the returned value is the same as the physical number of records in the file plus one. Note, though, that if the file has been written randomly, the record number sequence may well contain gaps. In this case the returned value will not give an accurate indication of the number of records in the file, but instead it gives the logical file size as a potential record count.

Function 24h: Set random record number

Action: Sets the random record field of an FCB to correspond to the current file position as reached by previous sequential access.

On entry: CL = 24h, DS:DX = Segment:offset of previously opened FCB

Returns: Nothing.
The random record field in the FCB is updated

Notes: This function is used to change from sequential to random I/O file access.

Function 25h: Reset the specified drives

Action: Resets to the default state the specified drive(s).

On entry: CL = 25h, DX = Drive vector

Returns: Nothing

Notes: The system resets the drives specified in the 16-bit value in register DX. Bit zero corresponds to drive A:, bit 1 to drive B. and so on. For each bit set on, the corresponding drive is reset to the login state.

Function 26h: Unallocated

Function 27h: Free specified drives

Action: None in DOS Plus.

On entry: CL = 27h, DX = Drive vector

Returns: AL = 0

Notes: This call is non-functional in DOS Plus. If used it always returns zero in AL.

Function 28h: Random block write

Action: Writes one or more sequential records to an open file starting at the file's current record position.

On entry: CL = 28h
DS:DX = Segment:offset of previously opened FCB

Returns: If successful: AL = Ah = Record count

If failed: AL = Return code, AH = Physical error

Notes: This call is similar to function 22h, with the exception that, before writing to a previously unallocated data block, the new block is first filled with zeros. These zeros then can be used to identify unwritten (i.e. non-existent) random record positions within the file.

Functions 29h to 2Bh: Unallocated

Function 2Ch: Set the system multi-sector count

Action: Sets the system multi-sector count for subsequent calls to functions 14h, 15h, 21h, 22h and 28h.

On entry: CL = 2Ch, DL = Number of 128 byte records to be written

Returns: AL = Return code (0FFh = DL out of range), BL = AL

Notes: The system multi-sector count determines the number of 128 byte records which will be read or written by each subsequent single call to the read/write functions shown above.

Function 2Dh: Set file system error mode

Action: Determines the action to be taken when physical or extended errors are encountered in filing system operations.

On entry: CL = 2Dh, DL = System error mode, as follows.
0FFh = Return error mode
0FEh = Return and display
Any other value = default mode

Returns: Nothing

Notes: The action of the filing system in dealing with errors is set through this call. If 'return error' mode is set a program can ascertain a filing system error code and decide how to deal with it. If 'return and display' mnode is set the filing system will both display the error to the console device and return the code to a calling program.

In default mode an error is displayed and the action or program is terminated.

Function 2Eh: Return unallocated space an specified drive

Action: Returns the number of free (128 byte) records on the specified drive.

On entry: CL = 2Eh
DL = Drive number

Returns: If successful: AL = 0, AH = 0
If failed: AL = Return code, AH = Physical or extended error code (1 or 4), BX = AX

Notes: The remaining unallocated space on the drive is returned as a binary number value in the first three bytes of the DMA buffer. The first byte is the low byte value, the last is the high byte value.

Function 2Fh: Load and execute a program

Action: Loads and executes the program specified in the current DMA buffer.

On entry: CL = 2Fh, DMA buffer = command line

Returns: Only if failed:
AX = 0FFFFh (no command line)
BX = AX
CX = error code (see beginning of Appendix)

Notes: The command line consists of an ASCII string terminated by 00h, placed in the default DMA buffer The command line may explicitly include a 'COM', 'CMD' or EXE' filename extension. if it does not the only type of file which will be loaded is 'CMD'.

The file will be searched for in the current and P: drives, under the current user and user zero. if the file is found the current program is terminated, its memory released and the new program is loaded and executed. If there is insufficient memory for the new program, because the calling program is already terminated, loading of the new program is abandoned and a message is output to the console device.

A two byte code may be passed from the caller to the new routine by means of function 6Ch.

Function 30h: Write pending internal disc buffers to media

Action: Forces all unwritten records contained in internal blocking/deblocking buffers to be updated to media.

On entry: CL = 30h, DL = 0FFh if buffers to be purged

Returns: If successful: AL = 0, AH = 0
If failed: AL = 0FFh, AH = Physical error code, BX = AX

Notes: This call is used to ensure that all internally blocked (depending on the multi-sector count) records are written to media.

If DL is set to 0FFh all the calling routines internal record buffers are also purged. This is necessary if read-after- write verification is required, to ensure that the verification data is taken from disc rather than from the program's record buffers.

Function 31h: Get/Set system variable

Action: This is a general-purpose call to return or set system variables which do not have specific separate calls.

On entry: CL = 31h
DS:DX = Segment:offset of system control byte

Returns: Parameter block (SCBPB) bytes laid out as follows:

0 Number of the system variable(s)
1: 0 = get variables, 0Fh = set variables
2-7 (1-5 bytes) The value of the system variable

SCUPB updated if 'get variables'

Notes: The value of the system variable and its possible settings is as follows:
0: Console width. 0-79 (Default 79)
1: Console page length. 1-24 (Default 24)
2: Console page mode. 0 = scrolling, 1 = page
3: System ticks per second (read only, value 50 on the 512 and any European PC, value 60 on USA PCs).
4: Temporary file drive. 0 = current drive, 1 = A: through to 10h for P:
5: The five-byte time and date information is held in the data block in the system area in a similar format to that shown for functions 68h and 69h.

Function 32h- Call the XIOS directly

Action: This function passes transparently through the BDOS to give direct access to XIOS functions by means of a range of sub-functions.

On entry: CL = 32h, DS:DX = Segment:offset of XIOS descriptor, the format of which is byte:
0: XIOS function
1-2: CX parameter
3-4 DX parameter

Returns: AX = AX = XIOS return code, BX = AX

Notes: XIOS functions officially supported by DOS Plus are:
0 Terminate program
1 Terminate program
2 Check for console input status
3 Read character from console
4 Write character to console
5 Write character to list device
6 Write character to auxiliary device
7 Read character from auxiliary device
Fh Return list device status
15h Device initialisation
16h check console output status

The memory block parameters shown as CX and DX should contain the 16 bit values passed in those registers when normal BDOS calls are made.

As can be seen these functions have direct equivalents in BDOS calls and several similar facilities are also available via DOS interrupt 21h.

These calls therefore offer no advantage over BDOS or DOS calls, given that they are more difficult to implement.

Function 33h: Set the DMA segment address

Action: Sets the segment address of the start of the DMA. (See also function 1Ah)

On entry: CL = 33h, DX = DMA segment

Returns: Nothing

Notes: The DMA base address is set to offset 080h in the program's data segment by default. Use of this area overwrites the command tail of the original command rifle calling the program, therefore if this area is to be preserved the DMA is moved by using this call.

Function 34h: Get DMA address

Action: Returns the address of the current DMA buffer.

On entry: CL = 34h

Returns: ES:AX = Segment:offset of current DMA address, BX = AX

Notes: None

Function 35h: Allocate max. memory available

Action: Allocates the largest single block of free memory equal to or less than a specified amount.

On entry: CL = 35h, DS:DX = Segment-offset of memory control block (MCB)

The layout of the MCB is as follows (bytes):
0-1: MCB segment, start of memory
2-3: MCB length, length of allocation
4: MCB extent, additional param./returned byte value.

For this call the MCB segment is set to zero. The MCB length is set to the minimum number of paragraphs required. MCB extent is set to zero if the memory is to be released after program termination, or 2 if the allocation is to be retained after termination.

Returns: If successful: AX = 0
If failed: AX = 0FFFFh, CX = Error code
In either case the following is also returned
BX = AX

MCB segment:	Base of allocated memory
MCB length:	Number of paragraphs actually allocated
MCB extent:	0 if no more memory is available 1 if additional memory is available

Notes: The MCB segment is always expressed as an absolute paragraph number from the start of memory. The MCB length always contains the size of memory to be allocated in paragraphs. The returned value in MCB extent indicates whether this allocation was the last free memory block in the system.

Caution. This call allocates the whole of the largest single memory block in the system, not limited to the amount requested.

Function 36h: Allocate maximum memory at specified address

Action: Allocates the maximum memory available from a specified start paragraph.

On entry: CL = 36h,
DS:DX Segment:offset of MCB (see function 35h for MCB layout) with data as follows:
MCB segment = start paragraph required
MCB length = Minimum No. of paragraphs required
MCB extent = 0 if area is to be released
= 1 if allocation is to be retained

Returns:

If successful: AX = 0

If failed: AX = 0FFFFh, CX = Error code

In either case the following is also returned:
BX = AX

MCB segment	= Base of allocated memory
MCB length	= No. of paragraphs actually allocated
MCB extent	= 0 if no more memory is available = 1 if additional memory is available

Notes: Operation is as for Function 35h, except that the caller specifies the start address for the required allocation.

However, all the remainder of that block is allocated, not limited to the amount specified.

Function 37h: Allocate exact amount of memory

Action: Allocates a precise amount of memory.

On entry: CL = 37h,
DS:DX = Segment:offset of MCB (See function 35h for MCB layout) with data as follows:

MCB segment	= start paragraph required (zero)
MCB length	= Minimum No. of paragraphs required
MCB extent	= 0 if area is to be released = 1 if allocation is to be retained

Returns:

If successful: AX = 0

If failed: AX = 0FFFFh, CX = Error code

In either case the following is also returned:
BX = AX

MCB segment = Base of allocated memory

MCB length = No. of paragraphs actually allocated

MCB extent = 0 if no more memory is available
= 1 if additional memory is available

Notes: Operation is as for function 35h, except that the calling program specifies the precise quantity of memory to be used for the required allocation. Note that in this call the location of the allocation is not controlled.

Function 38h: Allocate exact amount of memory at specified location

Action: This call reserves only a specified quantity of memory, beginning from a specified address.

On entry: CL = 38,
DS:DX = Segment:offset of MCB (See function 35h for MCB layout) with data as follows:

MCB segment = Start paragraph required
MCB length = Exact No. of paragraphs required
MCB extent = 0 if area is to be released
= 1 if allocation is to be retained

Returns:

If successful: AX = 0

If failed: AX = 0FFFFh, CX = Error code

In either case the following is also returned:
BX = AX

MCB segment = Base of allocated memory

MCB length = No. of paragraphs actually allocated

MCB extent = 0 if no more memory is available
= 1 if additional memory is available

Notes: Operation is as for function 35h, except that the calling program specifies the precise quantity of memory to be used for the required allocation. This call is generally preferred to the previous calls since they allocate the whole of the (smallest) single memory block in which the allocation can be accommodated and/or allocate the memory at an undefined location. This call reserves only the precise amount requested at a specified start location (usually the program's own PSP start address).

The preference for function 38h is because in a single user system the smallest available memory block is also often the only one, extending to the top of RAM. if any of the preceding calls are used and MCB extent is set to two, it may be that no other program can be run subsequently, even after termination of the current task, as the entire memory may remain allocated.

Function 39: Free memory at a specified address

Action: Frees the memory allocation stating at a specified paragraph address to the end of the allocated segment.

On entry: CL = 39h,
DS:DX = Segment:offset of MCB (See function 35h for MCB layout) with data as follows:

MCB segment = Start of allocated memory
MCB length = No. of paragraphs to release
MCB extent = 0 if the specified part of the allocation is to be released
= 0FFh if the entire allocation is to be released

Returns:

If successful: AX = 0

If failed: AX = 0FFFFh, CX = Error code

In either case the following is also returned:
BX = AX

MCB segment = Base of allocated memory

MCB length = No. of paragraphs actually allocated

Notes: Memory is freed from the paragraph address specified in MCB segment to the end of the allocation that contains that paragraph. Although part of an allocation can be freed, only a contiguous block from the specified start to the end of the allocation can be released.

It is not possible to release a block in the middle of an allocation, or which includes the start but does not extend to the end.

Function 3Ah: Unallocated

Function 3Bh: Load 'CMD' file into memory

Action: Loads a CMD or RSX file into memory

On entry: CL = 3Bh
DS:DX = segment:offset of FCB

Returns: If successful: AX = Base page address, BX = AX
If failed: AX = 0FFFFh, CX = error code

Notes: This call loads a CMD or RSX program module into memory, returning the base page address of the loaded module in AX.

Note, the program is not called for execution by this call. If the loaded module is an RSX, both AX and BX are set to zero.

Function 3Ch: Call RSX program

Action: Allows direct calls to be passed to RSX modules.

On entry: CL = 3Ch,
DX = RSX parameter block address. The RSX PB layout is as follows (bytes):

0	RSX sub-function number
1	No. of word parameters (n)
2-3	Parameter one
4-5	Parameter two
...	Parameter words as required
(2 bytes)	Parameter n-i
(2 bytes)	Parameter n

Returns: AX = 0FFFFh if no RSX function, otherwise nothing

Notes: RSX modules intercept and filter all BDOS calls by default, but this call permits a direct call to the currently loaded RSXs.
If no resident RSX recognises the sub-function code the system returns 0FFFFh in register AX.

Functions 3Dh to 62h: Unallocated

Function 63h: Truncate file to specified random record no.

Action: Sets the length of an existing file to the random record number specified in the FCB.

On entry: CL = 63h,
DS:DX = segment:offset of FCB

Returns:

If successful:	AL = 0, AH = 0
If failed:	AL = 0FFh, AX = Physical or extended error code (1,2,3,4,7 or 9), BX = AX

Notes: The calling program sets the drive field, the filename and file type fields and the random record field in the file's FCB prior to this call. The specified file is then truncated to the given random record number.

If the file is not found, the truncation length is not less than the current file length, or no such record number exists in the file (as it may not in random files) the call returns an error.

Function 64h: Create or update directory label

Action: Creates or updates a directory label for the specified drive.

On entry: CL = 64h,
DS:DX = Segment:offset of FCB

Returns:

If successful:	AL = 0, AH = 0
If failed:	AL = 0FFh, AX = Physical or extended error code (1,2,3,4,7 or 9), BX = AX

Notes: The calling program must set the file name and type fields of the FCB which are to be used as the directory label. If the directory is password protected the password must be placed in the first eight bytes of the DMA. The extent field should be set to define the user specification of the directory label data byte. If bit 0 of the byte is zero a new password is to be set, which must be placed in the second eight bytes of the DMA.

Function 65h: Get directory label data byte for specified drive

Action: Returns the existing data byte for the specified directory label in the specified drive.

On entry: CL = 65h, DL = drive number (A: = 0, B: = 1, etc)

Returns:

If successful:	AL = Dir. label byte (0 = no label), AH = 0
If failed:	AL = 0FFh, AH = Physical error code (1 or 4), BX = AX

Notes: The possible values for BL on entry are 0 to Fh. If the selected drive is in the reset state it is automatically logged in.

If an error occurs and the filing system is in default error mode, a message is displayed and the call terminates, otherwise AL is set to 0FFh and AH contains the error code.

Function 66h: Return file date/time stamp and password mode

Action: Returns the date/time information and the password mode for the file specified in the FCB.

On entry: CL = 66h, DS:DX = segment:offset of the FCB. The required data is as follows:

Byte 12:	Password mode field bits set Bit 7 Read mode Bit 6: Write mode Bit 5 Delete mode
or	
Byte 12	= 0 means no password is set.
Bytes 24-27:	Create or access time-stamp
Bytes 28-31	Update time stamp field

If the time stamp fields are set to binary zeros date/time stamping is not enabled.

Returns: If successful: AL = 0, AH = 0
If failed: AL = FFh (file not found),
AH = physical error code (1, 4 or 9), BX = AX

Notes: If the call is successful the date/time information and password mode are updated from the FCB.

If a file so marked is opened for write access the date and time stamps are updated if applicable.

If there is no physical error but the file does not exist, AL is Set to 0FFh and AH is set to zero.

Function 67h: Create or update extended FCB

Action: Updates a file's existing XFCB or creates an XFCB if no existing XFCB for specified file.

On entry: CL = 67h, DS:DX = Segment:offset of the FCB for the file

Returns: If successful: AH = 0, AL = directory code
If failed: AH = physical error code (1, 2, 4, 7 or 9), AL = FFh (physical error), BX = AX

Notes: The calling program must set the drive, filename and type fields in the referenced FCB. The extent field should be set to define the password mode and whether a new password is to be assigned. The bits of the extent field have the significance shown:

Bit 7:	Read mode
Bit 6:	Write mode
Bit 5:	Delete mode
Bit 0:	Assign new password

If the file is password protected the correct password must be placed in the first eight bytes of the current DMA. If bit 0 of the extent field is zero (new password) the new password must be placed in the second eight bytes of the DMA.

If there is no physical error but the file does not exist, AL is set to 0FFh and AH is Set to zero.

Function 68h: Set internal date and time

Action: Resets the system's internal clock for user date and time.

On entry: CL = 68h
DS:DX = Segment:offset of the DAT structure, the format of which is as follows (bytes):

0-1	Date field
2	Hour field
3	Minute field

Returns: Nothing

Notes: The date is represented as a 16-bit integer value of the number of days elapsed since January 1st 1978 inclusive. The time is stored as two separate bytes, one each for hours and minutes, each held as a BCD digit.

Function 69h: Get internal date and time

Action: Returns the current date and time from the system's internal clock.

On entry: CL = 69h,
DS:DX = Segment:offset of the DAT structure (See function 68h for the format)

Returns: AL = seconds,
BL = AL, DAT filled in days, hours & minutes

Notes: The format of the information is the same as for function 68h, with the exception that the clock seconds are returned in AL. Note that this value cannot be reset.

Function 6Ah: Set the default password

Action: Specifies a default password to be used by the system for protected files.

On entry: CL = 6Ah, DS:DX = Segment:offset of the password

Returns: Nothing

Notes: The password must be specified as an eight-byte field pointed to by DS:DX. When the system accesses a password protected file both the current DMA password and the default password are checked. If either matches the file's password access is permitted.

Function 6Bh: Get the DOS Plus serial number

Action: Returns the six-character serial number of the DOS Plus Operating System.

On entry: CL = 6Bh, DS:DX = Segment:offset of serial number field

Returns: Serial number field filled

Notes: The serial number is placed in the six-byte field pointed to by DS:DX. Each byte is an ASCII character.

Function 6Ch: Get/set program return code

Action: Sets a program return code prior to termination or returns the code from the most recently-terminated program.

On entry: CL = 6Ch
DX = 0FFFF: Get return code
DX != 0FFFFh: Set return Code

Returns: AX = Program return code, BX = AX

Notes: This call permits a program to pass on a two-byte value which can be interrogated by a subsequent program by means of the same call. If DX = 0FFFFh the return code of the last terminated program is returned. Any other value sets that return code as the current program's own code.

The return code can therefore be used by a calling program, or by the batch file command `IF ERRORLEVEL`

Return codes are:

0000h-FEFFh	Successful
FF00h-FFFEh	Unsuccessful
FF80h-FFFCh	Reserved
FFFDh	Fatal system error
FFFE	CTRL-C termination

A default return code of zero is set by the system unless the program was loaded by a previous program, by means of function 2Fh.

Function 6Dh: Get/set console mode

Action: Sets or returns the current console operation rode.

On entry: CL = 46h *(I suspect this should be 6Dh - editor)*
DX = 0FFFFh: Get console mode
DX != 0FFFFh: Set console mode

The console mode variable is a 16-bit word with significant values as follows:

Bit 0 = 1: CTRL-C status only
= 0: normal status function
Bit 1 = 0: Start/stop scroll enabled
= 1: Start/stop scroll disabled
Bit 2 = 0: Normal console output mode
= 1: Tab expansion/printer output (CTRL-I, CTRL-P) disabled
Bit 3 = 0: Enable CTRL-C interception
= 1: Disable CTRL-C interception

Returns: AX = Console mode or no value
BX = AX

Notes: This call controls the action of the system in processing (or not) control characters received from the console input device, and the values returned by interrogating the console status variable by means of function Bh.

Function 6Eh: Get/set output delimiter

Action: Returns or sets a new system output delimiter.

On entry: CL = 6Eh
DL = New output delimiter (set) *or*
DX = 0FFFFh Get output delimiter

Returns: AL = Output delimiter or no value, BL = AL

Notes: The output delimiter is the character the system recognises as the string terminator in function 9, string output. The default delimiter is the dollar sign ($).

Function 6Fh: Write a specified block of characters

Action: Writes a specified block of data to the logical console output device, CONOUT.

On entry: CL = 6Fh, DS:DX = Segment:offset of character control block (CHCB)
The layout of the CHCB is as follows (bytes):

0-1	Offset of character string
2-3	Segment of character string
4-5	Length of character string in words

Returns: Nothing

Notes: If the console is in default mode CTRL-I tabs are expanded and CTRL-S and CTRL-Q stop and start scrolling. Also CTRL-P Characters within the block will send subsequent characters to the printer.

If the console is not in default mode the actions are determined by the current mode setting.

Function 70h: Write characters to list device

Action: Writes a specified number of characters to the logical list device, PRN, from the area of memory indicated by the character control block.

On entry: CL = 70h, DS:DX = Segment:offset of CHCB (See function 6Fh for format of CHCB)

Returns: Nothing

Notes: If the logical list device is not available this call waits until output can be completed.

Functions 7lh to 8Ch: Unallocated

Function 8Dh: Delay a specified no. of ticks

Action: This call causes a wait of the specified time before execution is resumed. During the delay other processes may use the processor.

On entry: CL = 8
DH:DL = Number of ticks to delay

Returns: Nothing if successful.
If DL was 0: AX = 0FFh delay not supported

Notes: The minimum delay is 0.02 seconds, the maximum is 65,535 * 0.02 seconds, or 21 minutes 50.7 seconds.

Function 8Eh: Relinquish processor to other tasks

Action: This call relinquishes processor control to other programs running concurrently.

On entry: CL = 8Eh

Returns:

AX = 0:	Control returned, no problems
AX = 0FFFFh:	Program dispatch not supported

Notes: This call provides a means of giving up the processor to other programs. Each program is given an amount of processor time determined by the value of the SLICE command before control is passed on.

Function 8Fh: Terminate program

Action: Terminates execution of the calling program.

On entry: CL = 8Fh, DL = Abort code

Returns: Nothing

Notes: This call is functionally identical to BDOS call 0.

Functions 90h to 92h: Unallocated

Function 93h: Detach program from console

Action: Disconnects console I/O from the program which is then running in the background

On entry: CL = 93h

Returns:

If successful:	AL = 0
If failed:	AL = 0FFh, BX = AX, CX = Error code

Notes: The system supports up to three background tasks. If no (more) background tasks are possible an error is returned. It should be noted that programs which enter the background may no longer issue DOS Plus calls. Only BDOS and XIOS calls are supported.

Functions 94h to 97h: Unallocated

Function 98h: Parse ASCII string and init. FCB

Action: The system parses an ASCII file specification and prepares an FCB.

On entry: CL = 98h, DS:DX = Segment:offset of parsed filename control block (PFCB)

The PFCB contains the offset of the ASCII file specification, followed by the offset of the target FCB to be filled by the system. A minimum length string of 128 bytes is permitted, while the length of the target FCB must be 32 bytes. The filename is in the format:

```
(d:)filename{.ext} {;password}
```

where each of the items enclosed by `{ }` is optional.

Returns: AX = 0 if end of line or end of string. FCB as shown below
AX = 0FFFFh if error, when CX = error code, BX = AX
The FCB is initialised by this call as follows (bytes):

0	Drive ID. If not specified, default is used
1-8	Filename
9-11	Filetype (extension)
12-15	Zero filled

Notes: Leading blanks and tabs in the input string are ignored, but ASCII characters in the range 1 to 31 in the input string are treated as an error.

The system regards any parsing delimiter in the input string which is out of context with its location in the file specification as the end of the string.

The following are recognised as string delimiters, unless they are logically appropriate to their location in the ASCII file specification:

```
00h   null
0Ch   tab
0Dh   return
20h   space
2Ch   , (comma)
2Fh   | (solidus)
2Eh   . (period)
3Ah   : (colon)
3Bh   ; (semicolon)
30h   < (less than)
3Dh   = (equal)
3Eh   > (greater than)
5Bh   [ (left square bracket)
5Dh   ] (right square bracket)
```

All filenames and extensions are written to the FCB in upper case and blanks are used to pad the filename to eight characters and the extension to three as required. If a '*' occurs in a filename, that character and all remaining characters in the FCB entry are set to '?'.

Function 99h: Unallocated

Function 9Ah: Get system data area address

Action: Returns the segment address of the system data area, SYSDAT

On entry: CL = 9Ah

Returns: BX = 0, ES = SYSDAT segment

Notes: This call provides facilities for advanced programmers to directly manipulate system variables and data not catered for in the standard function.

It should be noted that the location of SYSDAT data is not fixed and may vary from version to version of DOS Plus, therefore access to this area should be used with discretion and only when absolutely necessary.

Functions 9Bh to ABh: Unallocated

Function ACh: Read characters from AUXIN

Action: Reads a specified block of characters from the auxiliary input device directly into a specified memory location.

On entry: CL = 0ACh, DX = CHCB offset address (See function 6Fh for format of CHCB)

Returns: AX = Number of characters read, DX = AX

Notes: This call returns the number of characters actually read, which may be less than the number specified in the CHCB. The call returns when either the auxiliary input device indicates there is no more data, or the buffer becomes full, whichever occurs first.

If no data is ready when the call is issued it waits until at least one character has been received. If "input exhausted" then occurs before the specified number of characters have been read, the call returns immediately

Function ADh: Write characters to AUXOUT

Action: Writes a specified block of characters to the auxiliary output device directly from a specified memory location.

On entry: CL = ADh
DX = CHCB offset address (See function 6Fh for format of CHCB)

Returns: AX = Number of characters written, BX = AX

Notes: This call returns the number of characters actually written, which may be less than the number specified in the CHCB. The call returns when either the auxiliary output device indicates no more data can be sent (the buffer is full), or the specified data has been transmitted, whichever occurs first.

If no data can be sent when the call is issued it waits until at least one character can be transmitted. If "output full" then occurs before all the specified data has been sent the call returns immediately.

Functions AEh to FFh - Unallocated

D: Example CP/M and DOS disc structures

Below are examples of sorne of the disc formats that may be encountered. Although this information is intended to be reliable the format and layout of any alien disc should be very carefully before details of its structure are used.

Refer to Chapter Nine for further information on the internals of disc structures. Remember that the number and position of FAT sectors and the root directory may vary from disc to disc. For DOS formats the relevant information should be contained at the beginning of sector zero.

DOS terminology is used here, though it should be noted that many of these formats are CP/M rather than DOS.

Acorn DOS boot disc (DISK 1)

Capacity	640k
Sectors per track	16
Bytes per sector	256
Directory entries	112
Sectors per cluster	8 = 2048 bytes
Reserved tracks	0
Number of FATs	2

Acorn 512 DOS Plus

Capacity	800k
Sectors per track	5
Bytes per sector	1024
Directory entries	192
Sectors per cluster	1 = 1024 bytes
Reserved tracks	0
Number of FATs	2

Acorn Z80

Capacity	400k
Sectors per track	10
Bytes per sector	256
Directory entries	128
Sectors per cluster	8 = 2048 bytes
Reserved tracks	1
Number of FATs	

Archimedes MS DOS 800K

Capacity	800k
Sectors per track	5
Bytes per sector	1024
Directory entries	192
Sectors per cluster	1 = 1024 bytes
Reserved tracks	0
Number of FATs	2

Almarc Spirit 16

Capacity	790K
Sectors per track	9
Bytes per sector	512
Directory entries	128
Sectors per cluster	4 = 2048 bytes
Reserved tracks	2
Number of FATs	

Altos 586

Capacity	720K
Sectors per track	9
Bytes per sector	512
Directory entries	176
Sectors per cluster	8 = 4096 bytes
Reserved tracks	2
Number of FATs	

Gemini CP/M

Capacity	790K
Sectors per track	10
Bytes per sector	512
Directory entries	128
Sectors per cluster	8 = 4096 bytes
Reserved tracks	2
Number of FATs	

IBM PC CP/M-86 single sided

Capacity	160 Kb
Sectors per track	8
Bytes per sector	512
Directory entries	64
Sectors per cluster	2 = 1024 bytes
Reserved tracks	0
Number of FATs	

IBM PC CP/M86 double sided

Capacity	320 Kb
Sectors per track	8
Bytes per sector	512
Directory entries	128
Sectors per cluster	4 = 2048 bytes
Reserved tracks	1
Number of FATs	

IBM PCDOS 160Kb

Capacity	160 Kb
Sectors per track	8
Bytes per sector	512
Directory entries	64
Sectors per cluster	1 = 512 bytes
Reserved tracks	0
Number of FATs	2

IBM PCDOS 180 Kb

Capacity	180 Kb
Sectors per track	9
Bytes per sector	512
Directory entries	64
Sectors per cluster	1 = 512 bytes
Reserved tracks	0
Number of FATs	2

IBMPCDOS 320 Kb

Capacity	320 k

Sectors per track	8
Bytes per sector	512
Directory entries	112
Sectors per cluster	2 = 1024 bytes
Reserved tracks	0
Number of FATs	2

IBM PCDOS 360Kb

Capacity	360 k
Sectors per track	9
Bytes per sector	512
Directory entries	112
Sectors per cluster	2 = 1024 bytes
Reserved tracks	0
Number of FATs	2

ICL PC CP/M

Capacity	720K
Sectors per track	9
Bytes per sector	512
Directory entries	128
Sectors per cluster	4 = 2048 bytes
Reserved tracks	4
Number of FATs	

Motorola VME/10 CP/M-86

Capacity	320K
Sectors per track	8

Bytes per sector	256
Directory entries	128
Sectors per cluster	8 = 2048 bytes
Reserved tracks	2
Number of FATs	

Nokia PC MS DOS format

Capacity	720K
Sectors per track	9
Bytes per sector	512
Directory entries	144
Sectors per cluster	2 = 1024 bytes
Reserved tracks	0
Number of FATs	2

OTRONA Attache

Capacity	400K
Sectors per track	10
Bytes per sector	512
Directory entries	128
Sectors per cluster	8 = 2048 bytes
Reserved tracks	1
Number of FATs	

Proteus

Capacity	800K
Sectors per track	10
Bytes per sector	512

Directory entries	128
Sectors per cluster	8 = 4096 bytes
Reserved tracks	2
Number of FATs	

Philips PG9000 GEMDOS

Capacity	640K
Sectors per track	16
Bytes per sector	256
Directory entries	128
Sectors per cluster	4 = 1024 bytes
Reserved tracks	2
Number of FATs	2

RML Nimbus MS DOS format

Capacity	720 k
Sectors per track	9
Bytes per sector	512
Directory entries	112
Sectors per cluster	2 = 1024 bytes
Reserved tracks	0
Number of FATs	2

Tandy model 2000 MS DOS

Capacity	720K
Sectors per track	9
Bytes per sector	512
Directory entries	112

Sectors per cluster	4 = 2048 bytes
Reserved tracks	0
Number of FATs	2

E: Tube Host Code

Tube Host Code

No listings or specifications were available for any of the code examples shown here, therefore they were saved from a live 512 system and disassembled. This has two implications. Firstly, they should be accurate, and secondly there is unfortunately no way to determine the precise purpose of some areas of data storage or some parts of the code in detail for all circumstances.

In the original disassembly there were short sections of code in page 6, now omitted, which were not called from any of the tube roufines here or elsewhere. Direct calls to ROM addresses within the code tend to suggest that they were incidental to tube operation. If they have any relevant function it is possible that these portions of code are used only during initial machine power-up and are subsequently overwritten. More probable however, is that they are downloaded simply because they just happen to occupy the last few bytes of that page, the bulk of which is relevant and remains.

The code shown below, which in essence and function is common to all BBC micro second processor support, consists of two parts. The first resides in page zero and is called at &16 on execution of a BRK instraction and is used to inform the co-processor of host software errors. During tube initialisation, when a hard break has been issued, if the tube is found to be active the break vector at &202 is therefore re-directed to point to &16 in zero page.

The second portion of the code is located in pages 4,5 and 6, normally reserved for the host's current language, which of course is irrelevant when running a co-processor. Page 7, normally the input strearn buffer, is used as a 256 byte data buffer for various transfers and calls. This code calls the zero page code at both location &32 and &36. In turn it is itself called by the zero page code. The entry addresses for this depend on the function called for by the co-processor. This is indicated by a single hex digit, detected within the idle loop at &36.

This is where the 6502 'waits' in the host code for co-processor dialogue when there is no current activity. The code entry points for the various function calls are located in a table of twelve addresses at &500 in both examples.

The two sections of the code are downloaded from the filing system ROM, provided that it includes support for the tube. The original DFS 0.9 did not, but all Acorn disc filing systens since DNFS 1.2, including the 1770 DFS and all versions of ADFS have, as have the alternatives from Solidisk and more recent versions from Watford Electronics. No Watford code has been checked, but like the STL version which has, it is likely to be identical byte for byte with the Acorn code, and the entry points, functions and returned values must be.

Note that the ADFS in the Master's 'Mega-ROM' is different to that in the stand-alone ADFS ROMs used by

model B or B+ machines. The Master version of this code is therefore shown in the second example of the host code. The major differences are a slight 'shuffling' of the order of several of the routines and the saving of a few bytes of memory by using a 2 byte 65C02 relative branch instruction (BRA) instead of a three byte absolute JMP.

These differences are not in the interests of efficiency, as in places this technique is clearly less efficient. It is rather a by-product of the fact that Acorn's ADFS code is produced by a macro assembler rather than being hand-coded, with the result that some of the generated code is highly stylised. As the entry points in the code must remain fixed (i.e. 6502.SYS does not configure itself to suit the host type or its MOS) any bytes saved are wasted, as shown at &47h. Also, for example at &65F, there are no less than 2 relative branches followed by an absolute jump where only an absolute IMP would normally be used.

For both examples of this code shown here the functions are substantially similar, therefore to aid comparison the same labels have been used where direct equivalents of operations exist. The latest incarnation of the Master's 'Mega-ROM', released in November 1989 (since these disassemblies were produced and too late for inclusion) is largely similar to the Master version shown and 'hackers' should have little trouble in dis-assembling and understanding the new code. For all versions of the code the entry points, the functions, temporary workspace and zero page code are the same.

For a 512, regardless of host type, additional functions are required over and above those of a 6502 co-processor. rfl the case of the 512 therefore, extra code is loaded from the DOS boot disc. Initially this is transferred across the tube to the 512 from the DOS file called 6502.SYS, contained in the boot disc. Before system initialisation this file is transferred back across the tube and located at &2800, after shadow RAM has been turned off by an OSBYTE 114. The 6502.SYS code is then initialised, which results in the redirection of three more of the host's vectors. The user-vector at &200 is re-directed to &2803, the interrupt request 1 vector at &204 is redirected to &2834 and the event vector at &220 is re-directed to &2831.

6502.SYS performs many functions not provided by the standard host code. These include very fast screen updates (by direct pokes, which is why the host's shadow is not active), direct programming of the 6845 CRTC chip for the various IBM screen formats, special actions for keyboard activity scanning and reading mouse or tracker ball movement via the user-port.

6502.SYS must also perform some direct floppy disc control as CP/M formats, PC formats, as well as the 512's 800K format are not supported by ADFS, although they are by the WD1770 or 1772 if it is programmed directly. It will be found by reference to the 6502.SYS disassembly that, unlike 6502 second processors, the host code is called by the 512 via 6502.SYS only when standard MOS or ADFS filing system calls are required, though for speed 6502.SYS also issues several MOS calls directly itself. The host code is therefore of secondary irnportance to the 512, for which the majority of functions are provided by its own specialised code.

```
; ******************
; 6502 TUBE-HOSTCODE
; ******************
```

```
; ******************************
; Tube register data declarations
; ******************************

r1-stat=&FEE0
r1-data=&FEEl
r2-stat=&FEE2
r2-data=&FEE3
r3-data=&FEE5
r4-stat=&FEE6
r4-data=&FEE7

; *************
; Host OS calls
; *************
; Filing system
; *************

osfind=&FFCE
osgbpb=&FFD1
osbput=&FFD4
osbget=&FFD7
osargs=&FFDA
osfile=&FFDD

; *******
; general
; *******

osrdch=&FFE0
oswrch=&FFEE
osword=&FFF1
osbyte=&FFF4
oscli=&FFF7

; ********************
; zero Page work area
; ********************

; The first two bytes are used for indirect loading 'from'
; when relocating a language across the tube in 6502 co-processors.
; In the 512 system the first 16 bytes are used as the parameter
; block for various OSWORD or filing system operations
```

```
.romadd
00 EQUD 0000
04 EQUD 0000
08 EQUD 0000
0C EQUD 0000

10 EQUW 00

; Bytes &12 and &13 are used for the hi-lo-byte pair for
; block transfers of memory across the tube

12 EQUW 00

14 EQUB 0                                   ; Tube status - &8O = free /
zero = busy
15 EQUB 0                                   ; Current owner ID - £80 = nobody

; Status bits are as follows
; Bit 0 = &O1 is the Interlock for FDC and IRQ in use
; Bit 1 = &02 is the Bad FSmap flag
; Bit 2 = &04 is the MON flaq
; Bit 3 = &05 is the *compact flag (i.e. 'Do not disturb') Not used by
DOS
; Bit 4 - &10 is set during an atomic series of operations for the MFM
; Bit 5 - &20 is set it there's a Winchester controller
; Bit 6 - £40 is set when tube is in use
; Bit 7 - &80 is set if tube is present

; The following is the BRK handling code located at byte &16 in zero
page.

                                    \


.brk handler
016 LDA #&FF                                \Load A with 255
018 JSR write_R4_data                       \Write A to tube data register A
01B LDA r2-data                             \Load A from tube data register
2
01E LDA #0                                  \Load A with zero
020 JSR write_R2_data                       \Write A to tube data register 2
023 TAY                                     \Set Y to zero
024 LDA (&FD),Y                             \Load A with first error byte
026 JSR write_R2_data                       \Write to tube data register 2
029 INY                                     \Increment Y
```

```
02A LDA (&FD),Y                         \Load second and subsequent
error bytes
02C JSR write_R2_data                   \Write to tube data register 2
02F TAX                                 \Transfer A to X
030 BNE &29                             \Was byte zero (termination)?-
No repeat

.reset_stack
032 LDX &FF                             \Load X with 255
034 TXS                                 \Transfer to stack pointer
(reset stack)
035 CLI                                 \Clear interrupt disable flag

; This is the tube-idle loop which keeps the 6502 amused
; when there is no current I/O activity or tube transfer
; in progress. It continually checks register 1 status
; and if nothing is required checks register 2 - until
; either requires attention this is repeated endlessly.

; Register 1 is reserved for the current output channel
; (screen, printer, RS232) hence an immediate OSWRCH is
; expected if R1 is pending - thus if an output delay is
; caused by the host or peripherals the co-pro is also
; forced to wait. It is the entry for all other activity
; therefore, even if R2 is pending. R1 is serviced first.

; The R2 entry causes an indirect jump to various code
; entry points in page 5 to give access to the required
; facilities.

.tube_idle_poll
036 BIT ri-stat                         \test tube register 1 status
039 flPL &41                            \Not ready - try reqister 2
03B LDA ri-data                         \Read tube data register 1
03E Jsn anwrch                          \write to current output stream
041 SIT r2-stat                         \Test tube register 2 status
044 BPL '36                             \not ready - try register 1
again
046 BIT ri-stat                         \Register 2 ready - recheck
register 1
049 assI &33                            \negimter 1 not, ready - do
that first
045 rnx r2-data                         \Read function cods tram data
register a
```

```
OAE Sn' &51 ~store at j~var. modifyina it- address
050 JMP (~WnpVHX to modified address Cs500+x - dirty!)

iL-var
051 EOUE C                              \variable 10-byte of j~ point
in page 5
052 EQUn &05                            \~ixed hi-byte of iL- point
(page 5

This is the 4 byte address used for language re-location across the tube

.reloc_data                             \used for 4 byte memory
addressing.
063 FOUR 0                              \LO-byte of low order of 4 byte
address
054 ~ous 0                              \Hi-byte of low order of 4 byte
address
oss tat's 0                             \These bytes contain 'rrFF for
the host
056 EQUE C                              \and &0000 upwards for the co-
processor

Model BBC B+ source listing

Tube host code for model n. n+ machines using stand-alone Ants
```

This is the start of the code which occupies pages 0 through 6.
Page 7 is also used as a buffer for 256 byte transfers

```
400 JMP &404                            \Entry to copy language across
tube
403 JMp &0A7                            \Entry to copy ESCAPE flag
across tube

This is the only entry point from 6502Sys

tube_entry                              \A-t29 for tube releace, 193
for claim
IC' CKP iSSO                            \Entry for 6502sYs - Tube claim
or release?
408 BCC &435                            \&t's neither - must be data
transfer
40A CKP faco ~Is it a tube claim?
40c ECS a428                            \Yes - branch to claim_tube
```

```
40E ORA • ;&4O                          \no it's a release - set bit 7
on
410 CMP &15                             \Are ~e the tube owner?
412 ENE &434                            \If not branch return - do
nothing
414 PflP                                \save processor status on stack
415 SE' ~set interrupt disable flag
416 LDA *5                              \Lcad A with 5
416 JSR write_R4_data                   \write A to tube R~
419 LDA i's                             \Load A with tube owner
AiD asa write RI data                  \Write A to tube RA
420 FLP                                 \Restore processor status from
stack
421 LDA t&60                            \load A with 128
423 STA &15                             \store in tube owner (&15)
42S STA '14                             \and in tilbe status (&14
427 RTS                                 \Return
-claim tube
428 ASt ~l4                             \Shift left tube status
42A Bes $432 ~Tube wa3 tree - now it'S ours - store owner
42C ChP &15                             \Tube busy. As it us in &15?
42E BEQ &4a4                            \Yes it~s already ours - exit
430 CLc                                 \clear carry - claim failed
431 RTS
\Return

.store owner
432 stA &15                             \store A in zero paqe byte &15
'34 nTs                                 \Return

data transfer
435 PHP                                 \save processor statue on stack
43' SEx                                 \Disable Interrusts (time
critical transfer)
437 STY &~S                             \store y (lo-address byte in '13
439 5,,' &12                            \Stort X (hi-address byte in &12
433 ask write_RI_data                   \write A to tube data register 4
43! TAX ~and also store in 'C
43? LDY #3                              \Load r for 4 byte transfer
441 tUn &15                             \Load A froit zero page
tube~owner
443 JSn write_~4_data                   \write A to tube data register A
446 WA ('12 ,Y                          \Load A fr~ indexed zero page
address
```

```
44S Jsn write_RA_data                    \write byte to tube data
register A
44fl flEy                                \Decrement count
44c BPL &446                             \co~1ete? No - loop back and
repeat
44E ~Y jale                              \Load Y 24
450 S~Y ri_stat                                 \write to tube
registeri status
453 WA ~51S.X                            \Load from temporary_data
45' STA ri_stat                                 \write register 1 status
459 LS't A                               \Divide value by 2
Ask tSR A                                \and again (- Divide value by 4
45a ncc &463                             \nranch if old bit 1 was C
4SD PIT r3_data                                 \waste a bit of time?
These two
460 BIT r3_data                                 \instructinns Set p but
do not affect A
463 JSR write R4 data                           \which is written to
tube data register 4
4C6 SIT r4_Stat                                 \Tnnt register 4 status
469 nvc &466                             \Not ready - loo~ until it in
463 BeS &47A                             \status ears terminate Operation
46D cpx ;4                               \was original A - 4
46? linE &4e2                            \no - branch to return

.soft_break
471 JsR &414                             \Inform co-pro of tube release
474 Js~ write_Ra_data                    \write A to tube data register S
477 JMP &32                              \J~ to reset_stack

.terminate_operation
47A LSR A                                \Clear top bit
475 BCC ~462                             \Branch if bottom bit was also
clear
47D WY f&aa                              \confirm action terainated
4~T Sty ri_stat                          \write y to tube register 1
status
462 pIr                                  \Rsstote processor status reg
from stack
4SS KTS
copy_langtiage

464 cLr                                  \Enable interrupts
```

```
465 MOS '49$                            \rf catry branch to claim tubs

467 mm &48c                             \rf not zero we~ve not finished
branch
489 JMp &59C                            \J~ to termination at
send_c~1etion byte
48c Lnx 40                              \Load X tero
4SE WY *&FF ~Load Y 255
490 WA flFD                             \Load A 253
492 JSR osbyte                          \Read~write last break
495 TXA                                 \Last break byte in X, transfer
to A
~96 BEQ &471                            \soft break - branch to
soft_break
49e LDA #~rr                            \Load A 255 (all bits on)
49A JSR &406                            \Exsc tube claim code (hard
break - startup)
49D BCC &496                            \If not successful repeat until
it is
49F JSR ~4D~                            \check current ~on, set up
reloc.rt lan4"age
432 LDA *7                              \Load A with 7 for call to
tube_entry
4A4 JSR &4C3                            \Jup to tube entry (with return)
437 LDY *0                              \set Y counter to zero
4A9 STY 0                               \Store in romadd (10-byte of
transfer addr
4An LDA (O~,Y                           \toad byte to be transferred
4AD STA r3_data                                 \write A to tube data
register 3
4B0 NOP                                 \These Mops are included s~ly
to cause a
AB1 NOP                                 \de1a~, as the as transfet is
'slow
4B2 NOP                                 \cThia extra NO? needed for
6502 and ZOO)
4Ba INY                                 \rncrwnent Y - Finished current
page?
4BA BNE &4AD                            \No repeat until 256 bytes
tranaterred
4B~ INC 654                             \Increment hi-byte, lot, order
of co-~ro addr
439 ENS &4C0                            \64K transferred? No-branch
update-host-page
4SA INC &55                             \Next 64K in co-pro. Inc. 10-
```

```
byte high order
4fle stat &4c0                          \16Mb co~1ete? no.branch npdate-
host-~age
4BE INC &56                             \:nc co-pro hi-byte hiah
ordercnext 1~nb!

update_hostjage
'Co INC 1                               \Incr-ent RON address hi-byte
~C2 BIT 1                               \Teat > ~ bit. In 'Ci (2~14 -
"'C eadresaed)
.C. BVc ~4~                       \rt tEK co-let. branch for tub, data
xfer
4C~ asa &4D2                            \Check current no" and set up
reloc.!! lsng
Ac' WA #4                               \Load A .ith 4 for tube
transfer type
403 WY                                  \Load Y (la-byte for tube data
transfer
4cn rnx ~&53                            \Load X (hi-byte rot tube data
transfer
4cr JXP ~4O6                            \J~ to tube_entry

check_roTa_type
4D2 LDA ~&SO                      \ncad A 120
404 STA &54                             \set co-pro relocation addre5s
to &0000e000
'De STA 1                               \And set hosttXfer ircit'
address to 'SOOC
ADS IDA t&20                            \set bit ~ Ott all othnr5 off
ADA AND &SOO'                           \Aflt current RON tyna byte -
is it a lang
cnn TA?                                 \Put result in Y in case At's
not
INTh s~Y &s3                            \put result in 'sa in case it~s
not
AED BEQ &4Th                            \means RON in NOT lang - branch
no_reloc
4:2 lAX 58007                           \Load X with ~I copyright
pointer offset

read C
4E5 Iba                                 \Inc X to point to firat C of
(C string
4s6 WA &8000.x                          \Load A with copyright bytes
until byte - 0
```

```
4E9 BNs~&4E5                            \Until 0 it~s the c'right notice
AEn LDA &6001,X                                \Load 1st byte of ca-
flrocessot reloc address
4EE STA &53                             \Store in zero page byte &5a
470 WA ~SOO2.x                          \boad 2nd byte of relocation
address
4r3 STA &54                             \Stote in zero page byte &S4
4r5 'fly &6003,X                               \Load Y with 3rd byte
of relocation address
~rs LDA aa004,x                         \Load A with 4th byte of
relocation address
reloc
4FB STA &5e                             \store accumlator in zero page
byte 6~6
4FD STY &55                             \store Y in zero page byte &55
4FF k~S                         \Return

; These are indirect jumps taken from the tube idle loop
; cede at &036 to &050 The a~ras5 In &O51 Is pnknd to
; index one of the following indirect jump addresses.

500 EOUW &3705                          \was 0 - jump osrdch_call
502 sOUW &9605                          \ was 2 - ju'np oncli_call
504 ECUW ar205                          \ ~n2 was 4 - 5u~ shor~ _nabyte
soe EQUW &0706                          \n2 was 6 - j~ long-osbyte
508 ~auw &2706                          \n2 was S - jump OSWORD_call
50A EQUW &6806                          \R2 was A jump cawordo-call
SOC EQUW &5305                          \ ~a3 C - ju~ osarga_call
SOE EQUW &2n05                          \ft2 was E - ~ o,bget_call
510 sOUW &2005                          \ was 10 - zump osbput_call
512 KOOW &4205                          \ ~n2 was 12 - itimp
osfind_call 1
514 EQuw &A~O5                          \ was 14 - ju~ osfilo call
516 tom' ~DlO5                          \ was 16 - j~ osgbpb_call

.te~rary-data
518 SOUw 00
51k RQWI OO
510 EQuw OG
51E RQUS OO
.onbput_call
520 JSn rend_'12 data                   \Load A with fil. handle from
register 2
523 TAY                                 \File handle must be in Y
```

```
524 Jsn read_fl_data                    \Load A with tube data register
2
527 Jsn asbpnt                         \Put a single byte to file
52A JMP                                 \Jmp to send completion byte

.osbget_call
520 JSR read_~_data                     \Load A with file handle from
r~i5ter 2
$30 TAY                                 \rile handle mu5t be in V
531 JSa osbget                          \c;et one byte from file
5a4 SM? asak                            \aun~ to transfer_one_byte

.osrdch call
53~ JSR

.transfer_one_byte
SSA flea A                              \shift tight one bit
San JsR write ~ data                    \write A to tube data reginter 2
sat 'tot a                              \Shift left one bit
3)4? &5~t                               \J-p to send unshifted byte

.osfind_call_1
542 Jsn read_RZ_data                    \Load A with tube data re~ister
2
545 BEQ osfind_call_2                   \zero byte means close a file
SA' PRA ~stnre A on ntack temporarily
541 JSa block transfer                  \Transfer file-naras from co~ro
54fl Pu                                 \nestore A (Contents - access
mode)
sac jsn osfind                          \open the named file
54F JXP &s9E                            \Jmp to return file handle

.osfind_call_2
55~ JSR read_fl_data                    \toad A with tube data register
2
555 TAY                                 \File handle must be in Y
556 LDA *0                              \Lcad A with zero
5se Jsn ostind                          \Clone file
SSn OMP &59c                            \Jnrnp to send conpietion byte

.osarga_call
55E Osri read_R2_data ~Load A tram tube data register 2
sel TAY                                 \put file handle in Y
562 LDx ~4                      \Set up loop count
```

```
564 3sn read_fl_data                    \LOad A fro~ tube data
register 2
567 S~A &r~,x                           \store at bottom of zero page
569 flEX                                \Decremnt count
SeA ENS &564                            \coa~1ete? No - loop back and
re~at
56c Jsa read_RI data                    \Load A with osarg~
call type
SEF JsR onarga                          \nead(write tile attrihutea
572 3srt write_R2_data                  \write A to tube data
register 2
575 rnx *3                              \Set up loop count for 4 bytes
577 WA O,X                              \Load A with returned attributes
579 Jsn write_'2_data                   \write a to tube data
re4ister 2
57C DEX ~D.crmnnt count
57fl BPL 6577                           \c~1et,? No - loop back and
repeat
$77 JwP '36                             \Jnnp to tube idle polling loop

-block transfer
582 tux to ~2oro
sac INY IC                              \and
Ses JSR read_fl_data                    \Load A from tube data
register 2
509 STA data,Y                          \store in 256 byte buffer at
&700
Sec '~                          \Incr~nt Y
SaD BEQ end_block                       \Finished 2S~ bytes? -
Yes
5SF am ~Nn                              \ wa3 byte CR (end of transfer
5~1 SNE isee                            \NO - Loop back for next byte

end-block
$93 ~Y ~?                               \Finisbed! - Load Y with 7
595 't~s                                \Rsturn

-ouch call
5~c Jsa block_ttansfer                  \Get cli string
S9~ JSR ascii                           \Call "0S co-and line
interpreter

send_copletion_byte
5~C ~A *'7r                             \Load A with 127 (co~1etion
```

```
byte)
SSE flI' r2_stat                                \Test tube register 2
status
Sal Evo ~59E                                    \not ready - repeat
5A3 STA rZ_data                                 \write A to tube data
register 2
5A6 JNp '35                                     \3~ to tube idle polling loop

osfile call
5A9 IDX talc                                    \Set up for is byte transfer in
x
SAn asa read_fl_data                            \toad A frOm tube data
register 2
SAL STA &i,x                                    \5tore at romadd+1,x
sao flEx                                        \Decreraent count
531 SNE &SAB                                    \1eto7 - Na loop back and repeat
5B3 JSR block_transfer ~transfer bytes
Sne STX 0                                       \Raset filename hi-byte to zero
SBR STY 1                                       \Set filena~ pointer la-byte to
Y
SnA rnY *0                                      \set x-Y to point to 16 byte
parameter block
5Bc SSR read_R2_data                            \Lcad A trOta tube data
register 2
SEF JSR osfile                                  \Ca1Y osfile - A defines
required action
5c2 JSR write_RZ_data                           \write result code to
tube data register 2
5C5 LDX #610                                    \set up for 16 byte transfer out
5C7 Ink 1,X                                     \Load from romadd+1.x
509 JSR write_R2_data                           \write A to tube data
register 2
SOC DEX                                         \Decro~nt count
5cn slit 'Sd                                    \c~lete? - '10 loop bac~ and
repeat
SCF BEQ &SAE                                    \2 stage j~ back to tub.
_idle_loop

.osqb~_call
sni inx ~&n                                     \toad X counter with length of
param- block
5n3 JSR read_fl_data                            \Load A from tubs data
register a
506 S~A &rr,x                                   \store at bottom of Page zero
```

```
(1 to aD)
SflS DEX                                \necremant count
Sflg BIlE &5D3                          \cc~let.~ - No loon back and
repeat
5DB JSK read_RI_data                            \Load A frog~ tube data
tegistet 2
SDE WY 40 ~sot Y to la-byte of parems
5t0 Jsn osgbpb                          \call osgetbytelputbyte
5E3 PRA                         \preserw A on stack
5E4 W~                          \set loop count
5~6 ~A O,X                              \boad returned file info~tion
SEa Jsa write_fl_data                           \write A to tube data
register 2
5ES flEx                                \Decrereflt count
5EC flPt &5E6                           \Complctc? No - Loop back and
re~at
SEE PLA                                 \flestore A from stack
5SF JMP $53A                            \Ju=p to transfer_one_byte

.short_osbyte
5F2 JSR read_n2_data                            \Load A tram tube data
re~ister 2 SF5 TAX                              \set X paramter
SF6 Jan read FIS data                           \Load cabyte n~ber from
tabe data reg SIrS JSR osbyte                           \Call osbyte

SF0 BIT r2_stat                                 \'remt register 2 status
srr nvc aSFO                            \Not ready - loop back until it
is
601 STX rZ_data                                 \write returned X value
to tube data reg 2
604 JKp &36                             \3wnp to tube idle polling 100p

.long_osbyte
607 JSR read RI data                            \Load A frog~ tube data
register 2
EGA TA)[                                \set osbyte 'C paramtar
Con Jsn read_R2_data                            \Load A from tube data
register 2
COE TAY                                 \set osbyte Y parameter
SOF Jsn read_R2_data                            \load A from tube data
register 2
612 JSR osbyte                          \call osbyte
615 EoR *&9D                            \was it osbyte 157 (fast tube
flPUT~?
```

```
617 BEG &604                                \Yea - Branch to idle poll j~
619 KOR A                                   \Not 157 - shift right one bit
GiA JSft write_R2_data                              \writo A to tube data
register 2
Gin BIT r2 stat                                     \Test tube register 2
status
620 ave &61n                                \NOt ready - loop back until it
is
622 STY rZ_data                                     \write Y to tube data
register 2
625 nvs aSFC ~nranch to write X value to register 2

.osword call
627 Jsn read R2_data                                \Load A from tube data
register 2
62A TAY                                     \Save read/write action in Y
enn nIT r2 _stat                                    \Test tube register 2
s~atus
62E BPL &S2a                                \Not ready - 1cc~ until it is
630 rnx r2_data                             \Load X with param. block offset
633 DEX                                     \Decrement X - If it~s zero no
parameter.
634 ml' &645                                \ now ~t?? - Yes branch to
csword_setup
636 BIT r2_stat                                     \lest tube register 2
status
639 nPL &~~6                                \not ready - loop back until it
is
63B Kak r2_data                             \ ~flead tube data register 2
63E STA &126,x                              \Store in paramter block at
offset
641 DEX                                     \necrement count
642 flPL &636                               \c~lete? no - loop back and
repeat
644 TYA                                     \nsstore read/write action from

.osword_setup
645 ~X ~&28                                 \Load X with patam. block
address lot, byte
647 Wy ~1                                   \toad Y with param. block
address high byte
644 asa onword                              \Ezecute osgord using parm.
block at &126
64c nIT z2_stat                                     \Test tube register 2
```

```
status
64r BPL &64c                    \not ready - loop back until it
is
551 LOx r2_data                 \Load A from tube data register
2
65~ flEx                        \Decrement x
655 SM' &6e5                    \cor~lete? Yea -
657 In? &12s,x                  \Else read paranstet into
~SA nIT r2_stat                         \.!est tube re~ister 2
statun
650 avc SeSA                    \not ready - loo~ back until it
is
65r STY r2 data                         \write y to tube data
register 2
'62 flEx                        \DecremeQt count
663 BPL &657                    \con~lete? no - loop back and
tepeat
s'5 S"? &36                     \Jtimp to tube idle polling loop

.oswordo_call
6's Lox #4                      \set loon count to 4 in x
SCA Jsn read_;12_data                   \Load A fro~ tube data
register fl
6~fl STA O,X                    \store at rmadd+x
66? flEx                        \necrement count until 'C - 'Fr
670 apt SGGA                    \5 bytes Xferred? - No loop
back and repeat
6~2 "ix                         \Incr~nt X (back to zero)
673 WY *0                       \Set Y to zero
675 TXA                         \Set A to zero
676 JSR ozwcrd                  \oet characters fro~ current
in~ut stream
679 BCC a6eo                    \RSturfl preaned? - Yes -
branch to wtite
G79 WA ~&FF                     \Escape pressed! - load A with
255 and
67D JMp &59E                    \J~ to write tegister 2

.read_string
~ao wx ~o                       \zeto loop counter
6S2 LOX *&7F                    \Load A with start string byte
6S4 Jsn write_&12_data                  \write A to tnbe data
registar 2
6S7 WA data.x                   \toad from 256 byte block at
```

```
&700
ERA asri write_R2_data                          \write A to tube data
register 2
Sen "ix                                 \Incr~nt count
'SE OMP l&D                             \End of in~ut warked by cMa$13?
6~O fiNE ~6S7                           \No - loop back for next byte
692 aMP &36                             \rndirect it- to tube idle
flolling loop

.write fl data
695 BIT r2_stat                                 \Test RZ status
696 BVc write_~_data                            \wot ready - loop until
it is
69A STA r2_data                                 \write A to tube data
register 2
'SD RTS                                 \RSturfl

write R4 data
6~E nIT r4_5tat                                 \Temt n4 status
Gal BYC write_R4_data                           \Not ready - loop until
it is
'A3 STA r4_data                                 \Write A to tube data
register 4
6A6 RTS                                 \Keturn

-copy_escape_tlao

6A7 WA &rr                              \Laad Accunnjlatcr with 255
6A9 SEC                                 \Set earrr flag
6AA "Ca A                               \notate right Accnmnlator bits
EAB ml! write_Ri_data                           \Branch on -ve flag
(bit 7. prev carry
SAD PHA                                 \?teserve A on stack
SAE InA 40                              \toad A with zero
630 Jsn write_RI_data                           \write A to tube data
register 1
6B3 7YA                                 \transfer content3 of Y to A
SEA JSn write_RA_data                           \nrite A to tube data
register 1
Sn7 Tm                          \Transfer content5 of X to A
GnS asa write_Ri_data                           \write A to tube data
register 1
EBB PIA                                 \Restore A ftoa stack
```

```
-write Ri data
6nc BIT ri_5tat                                  \rest al status
ear nvc write_RY_data                            \ae~at - Ri not ready
(bit 6 clear) Gd STA ri_data                            \write Ri data
from A
'CC ~fS ~Return

-read R2 data
5c5 BIT r2_stat                                  \Test Ttfl status
'CS B~t read_'t2_data                            \Repeat - ~ not ready
(bit 7 clear
ECA WA r2_data ~nsad regiater 2 data into A
600 aTS                                     \neturn

700 .dat.                                   \flata buffer far (up to) 256
bytes

; Master 128 source listing

; Tube host code for Master 128 machine, using Mega-ROM ADFS
; This is the start of the code which occupies pages 4 through ~
; Page 7 is also used. as a butter for 256 byte transfers.

400 JMP &4C2                                \Entry to cony language across
tube
403 JIw ~675                                \Entry to copy ESCAPZ flag
across tube

.tube_entry                                 \On entry A~129 for tube rel,
193 for claim
406 cXP ~aeo                                \Entry for 6502sYs-Is it tube
claim or tel?
409 Thea ~433                               \rt's neither - rust be data
transfer
40A CMp ~&co                                \!3 it a tub, claim?
40c ncs ~426                                \Yes - branch to c1a~_tube
lot OAR ~s40                                \NO it~s a release - set bit 7
on
'10 CKp als                                 \Are we the tub, owner?
~12 ENE ~4~2                                \If not branch return - do
nothin0
414 eMP                                     \save processor statun on atack
415 SEI                                     \set interrupt disable flag
416 ~A $5                                   \boad A with 5
```

```
41S JsR write_RA_data                          \write A to tube a'
41n Jsa write n4_tube_owner                           \Load A with
tube otmer and writ. to Re
41K pLp ~Reator.                               \rooessot status tram
stack
417 IflA ,&E0 ~load A with 128
421 STA &15                                 \Store An t~ owner (&15
423 STA &14                                 \and in tube stattis (&14
~25 Ft's                                    \'etnrn

.claim_tube
426 ASL &14                                 \shift left tube status
428 Ecs &430                                \ uaa free - now it'a ours -
store own~r
42A CPe '15                                 \Tube bnsy, is it us in &15?
42e etc &432                                \yes it~s alreadr ours - exit
42& crc                                     \Clear carry - claim failed
42r R7S                                     \Return

.store owner
430 STA &15                                 \store A An z~ro page byte ~15
4~2 RTS                                     \Rsturn

.data_transfer
Ass pnp                              \save processor statue on stack
434 SEI                                     \Disable interrupt , (tine
critical transfer)
4~5 STY '13                                 \store Y (10-address byte in &lj
41~ STX &12                                 \starn X (hi-address byte) in
&12
439 JSR write_R4_data                       \write A to tube data register 4
43c TAX                                     \and also store in x
43D LDY .3                                  \Load Y for A byte transf.r
4SF Jsn write_R4_tube_ovaer                           \Load tube
owner and write to Ra
442 InA c&12}.y                               \Load A frm indexed
zero page address
444 Jsn writ._rt4_data                        \write byte to tube
data register A
447 flEX                                    \Decreent
448 aph ~4~2                                \comp1~t,? No - loop bacir and
repeat
44A WY isle                                 \Load Y 2~
440 STY ri_stat                               \wtite to tube
```

```
tegisteri status
''F WA '5i6,X                                \Load frort te-rary_data
45~ STA ri_stat ~write regist,r 1 status
455 LSR A                                    \Divide value by 2
456 LSR A                                    \and again C- Divide value by 4)
457 ncc aAsr                                 \ntanch if old bit 1 Was 0
ASS BIT r3_data                                     \waste a bit of tint?
These two
45C sI'r ra_data                                    \insttuctions set F but
do not attect A
4Sr JSR write_R4_data                               \Wh*ch in written to
tube data register 4
462 srT r4 stat                                     \&para;est regi5ter 4
statun
465 Bvc ~4~2                                 \Not ready - loop until it is
467 acs &476                                 \statn5 says terminat, op.ration
469 cpx #4                                   \wa5 original A - I
'GB BRE a47E - branch to return
160 Jsn '414                                 \Inform co~ro of tube release
470 JSn '561                                 \wrAt. A to tube data register 2
473 JMp ~32                                  \J~ to reset_stack

.terminate_operation
47' 1&rt A                                   \Clear top bit
4~7 scc &47E                                 \Branch if bottom btt was also
clear
47~ tar ~'ae                                 \confirm action terminated
'75 sTY ri_stat                                     \write Y to tub, rgi,t.
r 1 status
47t Pt,                                      \Rostore processor Status reg
fr~ stack
47F RTS                                      \Returfl
460 Wx &29n -                                \s copy of last BREAK type byte
4e3 BEQ &46D                                 \O-soft - branch inform 512,
reset stack
4S5 ~A ~&FF                                  \Load A 255 (all bits on)
4$, Jsn &406                                 \Ex tube claim code (hard break
- startup)
48A floe &4e5                                \If not successful repeat until
it is
4Sc JSR &4C9                                 \Ch.ck current RON, set u~
reloc.if lang
4er pHp                                      \Save processor status
490 S~X                                \Disab1e interrusts
491 IrA ~7                                          \~ad A with 1 for call to
```

```
tube_entry
4,3 JSR SAaB                            \J~ to tube entry (with return)
496 tnT 40                              \Store in rgmadd (la-byte of
xfer address)
49$ STZ a                               \store in romadd (10-byte of
kfer address)
49A rflA (O),Y                          \Load byte to be transferred
49c STA r3_data                         \wtite A to tube data register 3
49r flop                                \These flOps are included
sin~ply to eanse a
4A0 NOP                                 \delay. as the Ra transfer is
slow'
4Al NOP                                 \(Extra NO? needed for the 6502
and zec)
4A2 INY                                         \Increment y - Finished
current page?
4As BNE ~49A                            \No - repeat until 256 bytes
transferred
4A5 PLY                                 \aestore processor flags
4A6 Inc &s4                             \~nc hi-byte. low order of co-
pro address
AAS ENE &4B0                            \64~ xferred? Nn - branch
update-host-page
4AA INC ass                             \Next 64K in co-pro- Inc- ic-
byte high order
4AC BNs SABO                            \1&Wb c~lete? No-br. update-
host-page
                                \increment fiflM address hi-byte

4AE INC &5E                             \Inc co-pro hi-byte high ordcr
(next 16Mb-)

.update~hcst~age

4n0 INC ~l                              \:ncre~nt ROM address hi-byte
4n2 BIT al                              \test > 6 bits in &O1 C2~14 -
16K adnressed
454 Bvc '4sf                            \If 16K c~1ete branch for tube
data xter
dnA 3sn &4C9                            \check cnrrent RON and set up
reloc-if lang
4S9 LDA ~4                              \Load A with 4 for tube
transfer type
49S ~Y *0 ~Lcad Y (10-byte) for tube data transfer
```

```
4nn rnx ~&53                                \Load X (hi-byte) for tube data
transter
43? OMP &4C6                                \Jti'ap to tube_entry

copy_language
4c2 CLI                                     \Enable interrupts
4c3 ncs a4as                                \If carry branch to claim tube
4c5 flNr &480                               \If not zero we've not finished
- branch
AC? BRA &52A                                \a tc termination at
send_completion_byte

.check_ram_type
(Cs tAX saso                                \Load A ~2B
4CR STA &54                                 \set co~pro relocation address
to &0000s000
4Cn STA I                                   \And net host ~Xter frOTh
address to
ICF ILA ~a20                                \sat bit 6 on, all others off
4n1 AND &BOOC                               \Abtj cnrrent ROM type byte -
is it a lang?
4D4 TAY '                                   \Pnt result in y in casa it's
not
Ans STY &53                                 \put result in &53 in case it's
not
4D7 BEQ &4t2 - RON is NOT a language - branch no_reloc
4D9 LDx &e007                               \Load X with $tO~ copyright
flointer offaet

read C
Cflc INX                                    \Inc X to first of '(C), atring
4DD LDA &a000,x                                    \Load A with copyright
bytes until byte - 0
4s0 nnr &4nc                                \untii it¹s the Cc notice -
branch read_C
4E2 LDA &E00!,x                                    \Lnad 1st byte of co-
pro reloc address
4E5 STA &53                                 \store in zero page byte &5~
4E7 LDA ~8OO2,x                                    \Load 2nd byte of
relocation address
4EA SIA &54                                 \store in zero page byte &54
4EC LDY &8003.x                             \LOad Y with 3rd byte of
relocation address
4EF LDA &E004,x                                    \Load A with 4th byto
```

```
of relocation address

no reloc

4F2 STA &56                                  \store accuinulator Yn zero
page byte &56
4r4 STY &55                                  \Store Y in zero flage byte &55
4F6 RTS                                      \Return
4F7                                 \These addresses are undefined- They
are                                 \unused because the code was re-written
AFA                                 \for the master so as to use less
'reraory.
4F0                                 \but the addresses at &500 cannot be
moved
ArE                                 \therefote these bytes are redundant.

These are indirect j~s taken from the ttte idle loop code at £036 to
&0S0-
The address in '051 is poked to index' one of the following indirect
jump addresses.

500 EQUW &3505                               \n2 was a - i~ osrdch_call
502 EOns &6605 was 2 - ii- ouch call
504 FOUW sokos wan 4 - it- short_osbyte
SOS EQUW &EBOS                               \R2 was 6 - j~ lon~-osbyts
508 EQUw ~O7O6 was S - it- OSWORD_call
50A ECUW &3606 was A - j~ OSWORDO-call

50C EQUW &5905 was C - ii- osarga_call

SOE EOtM &2005 was S - j~ osbget_call

510 EQUW &2005 was 10 - ii- csbput_call

512 EQWI &3r05 was 12 - jur~ osfind_call_1

514 EQUW &8205 was 14 - jump osfile_call

5h EQUW &9A05 was 16 - jump osgbpb_call

.t~otary_data
SIB EQLM OO
51A EQUW OO
```

```
Sic tQWI OO
SlE EQUW 00

.osbpnt_call
520 JSR read_n_data                         \toad A with file
handle tram rnginter 2
$23 TAY                                  \File handle ~ust be in r
524 asa read ~ data                         \Load A with tube data
tegister 2
527 asa osbput                           \pnt a single byte to file
52A flRA aSSE                            \Branch to send completion byte

.osbget_call
52c JSR read_~_data                         \Load A with file
handle from register 2
52F TAY                                  \rile handle mnst be in Y
530 JSa os~et                            \Get one byte tram file
533 BRA &538                             \atanch to transfer_one_byte

.osrdch call
535 JSR osr&h

.tranafar_one_byte
53B ROR A                                \shift riQht one bit
539 JSa &661                             \write A to tube data register 2
53C ROL A                                \shift left one bit
53n BRA &590                             \Branch to send unshifted byte

.osfind_call_1
5SF JsR read_~_data                         \Load A with tube data
register 2
542 nEC &54~                             \zero byte means close a file
544 P'h                                  \store A on stack te~orari1y
545 JSR block transfer                      \transfer file-name
from co-pro
546 pLA                                  \Reatora A (Contents - access
made
549 JaR csfind                           \open the named tile
54C BRA &590                             \nranch to return tile handle
ostind call_2
54E JSfl read_RZ_data                       \Load A with tube data
register 2
551 TAY                                  \rile handAc must be in Y
552 tak                            \Load A with zero
```

```
554 JSR onfind                          \Clnse file
55, flRA '56&                           \Jump to send con~1etion byte

.osargs_call
55* asa read_~_data                             \Load A fror tube data
register 2
55C TAX                                 \put file handie in Y
55n ~x .4                               \set up loon count
55r Jsn reftasarams                             \Read parameters into
zero page
562 JSR osargs                          \Rsad~write file attributes
565 asa write ~_data                            \write returned A value
to data register 2
sea 'nx .3                              \Set up loon cOunt fat 4 bytes
SEA boA O,X                             \Load A with returned attributes
Sec asa write_fl_data                           \write A to tube data
register 2
5SF flEx                                \Dectenent count
570 Ept &56A                            \co~1ete? No - loop back and
repeat
572 BRA &59e                            \Indirect branch to tube idle
polling loop

.block transfer
574 wx 40                               \zeto x
57~ WY *0                               \and Y
57$ JsR read_fl_data                            \Load A from tube data
register 2
57B S~A data,Y                          \Store in 256 byte buffer at
&700
57E 'NY                                 \rncrement Y
S7F nEO end_block                               \Fininhad 256 bytes? -
Yes
501 cMp ~&D                             \No - was byte CR (end at
transfer
593 EnE &57a                            \no - Loop back for next byte

end block
5SS ~Y ~7                               \rinished! - Load Y with 7
5S7 RTS                                 \flrturn

oseli call
sea asa block_transfer                          \Get cli stting
San 3sn ascii                           \Call "Os car-and line
```

```
intarpreter

send_cc~letion_byte
SeE InA ~&7F                          \Load A with 127 (co~1etIon
byte)
5~O BIT r2 stat                              \Test tube register 2
status
593 Bvc ~59O                          \Not ready - repeat
595 STA ri_data                       \writs A to tnt,. data regimt.r
2
596 WtA LSES                          \Indirect branch to tube idle
polling lo~
-osg~b_call
S9A WIC ~sn                           \Load X counter with length of
param- block
59c as,' read~ramn                           \Raad zero ~age ~aram
trO# tube register 2
Sgr w~ 40                             \set y to la-byte at parama
SAl JSK os~pb                         \call osg,tbyte~putbyt.
5A4 PHIL                              \Pr.serv. A on stack
5A5 Lox IaC                           \set loop count
SA7 LnA O,X                           \Load returned tile information
5A9 JS't write_RQ_data                       \write A tn tube data
register 2
SAC flEx                              \Decrement count
SAD Ert s5A7                          \conp1ete? No Loop hack and
repeat
SAF PTA                               \nestore A tr~ stack
5B0 BRA &53e                          \etanch to transfer_one_byte

onfile_call
5B2 Lnx ,&1o                          \set up for 16 byte transfer in
5n4 Jsn read_~_data                          \Load A from tube data
register 2
5S7 S~A 1,X                           \Store at romadd+1,X
5B~ fltx                              \flecrement count
SNz t5n4                              \co~lete? - No loo~ back and
repeat
SEC JSa block_transfer                       \transfet bytes
5Br STX 0                             \Raset tijename hi-byte to zero
Sal STY 1                             \Set tiYename pointer b-byte to
Y
sca 'fly ~o                           \set x-Y to point to 16 byte
parameter block
```

```
505 JSn read_pt2_data                          \Lcad A tror tube data
register 2
505 aSa osfile                                 \Call ostile - A defines
re~uired action
ScB Jsa write_RZ_data                          \Nrite result code to
tube data register 2
5CR Ifix '&10                                  \Set u~ for 16 byte transfer ont
Sno KnA 1,X                                    \Load tram rmadd+1,X
5n2 JSR write_~_data                           \write A to tube data
register 2
5D5 DEX                              \flecremCnt count
5D6 SNE s500                                   \cc~lete? - No loop back and
repeat
5DB BRA &5E8                                   \2 stage in- back to
tube_idle_loop

short_osbyte
SDA Jsn read_x_and_A                           \Read X an~ A params
for short osbyte
5DD JSR osbyte                                 \ca11 osbyte
5~O nIT r2_stat                                \Teat register 2 status
SES nvc &SEO                                   \not ready - lccp back until it
is
SES STX r2_data                                \write teturned X value
to tube data reg 2
5E8 JKp 636 to tube idle pol1in~ loop

long-osbyte
5E3 JSR read_x_and A                           \ncad X and A params
for osbyte
5EE TAY                                        \~ove A parameter to Y
SEF JsR read_R2_data                           \Load A from tube data
register ~
SF2 JSR osbyte                                 \call osbyte
5F5 Eon *&9D                                   \was it osbyte 157 (fast tube
flPUT}?
5F7 BEQ 'SEa                                   \yes - Sranch to idle poll j~
579 ROR A                                      \Not 157 shift right one bit
STA asa write_fl_data                          \write A to tube data
register 2
Sm BIT rZ_3tat                                 \Test tube re4ister 2 status
600 EVa SSFD                                   \Not ready - 100P back until it
is
602 STY r2_data                                \write y to tube data
```

```
re~ister 2
'05 Sak &5E0                          \Branch to write X value to
regAster 2
.OSWORD call
~C7 asa read_%2_data                          \Load A from tube data
register 2
60A Thy                               \save read~write action in Y
603 JSR read_x_n_data                         \~ad X from tube data
register 2
~OE SM!                          \61A now &FF? - Yes branch to
OSWORD_setup
610 JSR read_n_data                           \Rsad patanrbera from
tube data register 2
613 SZA ~l2S,X                        \stnre in paranter block at
offset x
'16 DEX                               \necrersent connt
617 SPL &610                          \c~lete? no - loop back and
re~at
619 TYA                               \aestore read/write action from
y

.onward_setup
61A LDX ~$28                          \toad 'C with patam. block
address low byte
610 KnY 41                            \r'oad Y with param- block
address high byte
EYE JSR osijord                               \SXecut, O5WO~ using
param block at &12e
621 asa read_x_R2_data                        \Read X from tube data
register 2
624 3"' &5Ee                          \coapl,te? Yea
626 WY &129,x                         \E1S. read parameter into Y
62g SIT t2_stat                               \Teat tubs registet 2
status
62c Bvc &62~                          \Not ready - 1oo~ back until it
is
~2E sty rQ_data                               \writ. Y to tubs data
register 2
631 flEx                              \D~erement count
632 EPt ~62~                          \co~1ete~ No - loop back and
tepeat
634 BRA asEe                          \rndirect branch ~o tube idle
polling loop
```

```
.cswordO-call
636 ~X                                  \Set loop count to ~ in 'C
GSa JSR read_R2_data                     \Load A tram tube data register
2
E3B sta o,x                              \Store at rcu-dd+X
63n Dtx                                  \Decremsnt count until X - 'Fr
63t S?t &630                             \5 bytes xterred? - No loop
back and repeat
640 Ibix                                 \rncrtment X (hack to zero
641 Tm ~set A to zero
~42 TAY                                  \Set Y to zero
643 Jsn caword 'Get character. frorn current Snout stream
646 scc &GAD                             \Return pressed? - Yen - branch
to write
64S LDA i&FF                             \Escape pressed - load A with
255 and
64A JMP &590                             \J~ to write register 2

.read_string
SAC Inx *0                               \zero loop counter
64F InA 4&7t                             \Load A with stact string byte
&51 3SR write_n_data                            \write A to tube data
register 2
654 WA data.x                            \Load from 256 byte block at
6700
657 JaR write ~ data                            \write A to tube data
register 2
'SA INX                                  \Incre=flt count
CM? ~&fl                                 \tnd of input tarked by CHR$13?
CSD ENE &654                             \No - loop back for next byte
6SF BRA &634                             \Indirect iL- to tube idle
polling loop

.write_R2_data
661 BIT r2-stat                                 \test ~ statns
GSA Eva write_R2_data                           \Nat ready loop until
it is
665 STA r2_data                                 \write A to tube data
register 2
669 RTS                                  \Return

.write R4 tube owner
6~A Ink &15                              \Load current tube owner
```

```
write nA data
Sec BIT r4_stat                          \Te.t Re status
Ser nvc write_RA_data                   \Not ready - loop until it is
671 STA rC_data                          \write A to tube data regAster C
674 RTS                                  \Retnrn

.copy_escape_flag
675 LDA #FF                              \load Accumulator with 255
677 SCF                                  \set carry flag
676 ROR A                                \Rotate right Acurulator bAts
679 BA write_Ri_data                             \Braneh on negative
flag (bit 7. prev carry)
67B PHA                                  \preserve A on stack
670 LDA #0                               \load A with zero
67E JSR write_Ri_data                            \write A to tube data
regist,r 1
601 TYA                                  \transfer contents ot Y to A
602 JsR write_Ri_data                    \write A to tube data regist.r 1
685 TKA                                        \Trannter contenta of K
to A
6as JSR writ._Ri_data                    \write a to tube data register 1
604 PLA                                  \Ra5tcre A from stack

.write Ri data
65A arT ri_stat                          \Test Ri status
65D nvc write_Ri_data                    \n~ps.t - Ri not ready (bit ~
clear)
65F STA ri data                          \Write Ri data frm A
662 RTS                                  \Return

.read~atams
693 asa read_32_data                     \load A irm tube data register 2
6SE STA &FT,x                            \store at bottom of zero page
6~8 flEx                                 \necrermant count
699 ENE rcad~rninn                       \c~lete? no - loo~ bac~ and
tepeat
69B SKA read_fl_data                     \Branch read Accumulator from ~

read X and A
69D Jsn read_fl_data                     \Read data register 2 into A
and transfer
6A0 TAX                                  \the value to x, then ta~1
througb to..
```

```
.read_R1_data
SAl SIT rQ_stat                         \Test ~ status
6A4 flPt read_~_data                            \Repeat - n not ready
(bit 7 clear
EAE Efla r~_data                                \Read register 2 data
Into A
EAS RTS                                 \neturn

.read_R2_data
EAA SIT r2_stat                         \Test R2 status
EAD EPL read_x_w_data                   \Repeat - ~ not ready bit 7
clear)
EAr U)Z r²_data                         \Read register 2 data into X
6n2 DEX                                 \Decrenent the value
'Ba RTS                                 \Return

700 data                                \Data buffer for up to 256 bytes
```

F: 6502.SYS code

The following two sections of code are 'specials' for the 512. In both cases the code was actually saved in a live machine and disassembled, so as to ensure complete accuracy and to permit mernory addresses to be included in the listings, as both sections of code are position dependent. (i.e. slightly dirty and self modifying). DisassembIy was also used for two other reasons. First for 6502.SYS it was done to produce a source listing which looks more like a normal assembly code program to 6502 users, necessitated by the reason given below. The OSWORD &FA code was disassembled from a live system for the simple reason that neither source code nor source documentation was available in any form other than the input parameter spedfications detailed in chapter 6. The strange techniques employed in this code are due to two factors. The first is that Acorn commonly used relative addressing so as to make the code relocatable as far as possible, eg:

*0 clear A
BEQ so-where Branch sc-where - always

The second is that all Acorn 6502 code appears to be produced using MASM, which simply does not produce the same effidency as hand written assembly code.

The first listing is of the tube handling code which resides in the 6502.SYS file on the DOS boot disc. Interestingly, the original source was actually written in DOS and assembled using a cross-assembler the source for which looks slightly strange if you are not familiar with 8086 assembler.

The reason a DOS cross-assembler was used was that it appears from other information we cannot publish that the source code was at one time intended to be easily modifiable for use by other hardware than a BBC rnicro and a 512!!

The first section of code is the program from 6502.SYS proper, while following it there is the patch which resides at &2300 to handle the OSWORD &FA, which was originally also in 6502.SYS. OSWORD &FA was removed from disc for practical reasons as well as 'neatness'.

The code was separated so it could be put into the 512's boot ROMS to enable it to assist virtually irrunediately with the rapid tube transfers used on initial system start-up, rather than itself first having to be '*LOAD'ed from an ADFS disk, which would look untidy and of course would take longer.

6502.SYS provides the main support for cross-tube dialogue between the 512 and the host for the majority of 512 operations. Only when a tube claim or release is called for, or a completely standard MOS or filing system function is required, is the tube host code, resident in pages 4 to 6 of the host called.

6502.SYS source listing

```
TUBE handling code for the Master $12
```

This code uploads to the 512 during initial system boot.

The boot strap loader looks for a file 6502.SYS in the root directory of the DOS+ boot disk and uploads it to the 512's MX alonq with DOS+.

Before initialisinq the system it downloads this code to 6502 memory at offset £2800.
It then sets the naer vector to point to the code's first executable instruction at £2803

To make the code accessible to the boot-strap loader:

1. Assemble code at offset &2800 and save to disc
2. Boot DOS+
3. '~SAVE' the assembled machine code tile from the ~~ Thicrn~a dine to the file 6502.SYS in the root directory of the DOS+ boot disc.

```
NOTE - that scras of the code here is position critical and much of
it is also time critical.
Various addresses and data declarations for the 6502 MOS calls and
hardware interfaces follow.

ZERO PAGE VARIABLES

ken - &EC new ks~ press in zero page

- 'ED old key press in zero ~aqe

STANDARD MOS CALLS & VECTORS USED BY THIS CODE

keyvec - &22B The keyboard vector
oswrch - aFFEE Console (screen) ontput
osbfte - &FFF4 General MOS call entry

ADDRESS OF ZnttflcSPTEfl BRC MICRO VECTORS
userv - ~2OO user vector
brkv - &202 The break vector
irqvl - &204 Interrupt requeSt 1
eventvec - &220 The event vector
```

ENTRY TO BBC n:cao~s STAhrDUO ~UBE CODE
tube_entry - &40~ c1aAjn~re1ease/pro4raTh tube n"I

SHeILA K-fly MAPPED ADDRESSES
ic~ge - &rsoo ban, addrena of SHEILA
erte_addr - &FE00 eels CRTC address register
crtc_data - &FE01 'e45 CRTC data register
unerport - &FEEC user-port data I-a
up_rdwrt - &FEE2 User-port data direction
AcRu - &Ftea 6522 auxiliary control register
Ihtu - &FEED Iaterruflt flag register
IrRn - &FEGD Interrupt flag register
IERU - &FESE Interrupt enable register

1770 FDC OFFSETS FROM &FEOO
Master fdc base - &24
Master_fdn_stat - &28
Master_fdc_data - &2B
BBC_fdc_base - &80
BBC_fdc_stat - &84
BBC_fda_data - &e7

THE TUBE STATUS AND DATA REGISTERS
****
tubeaistat - &ttto Register 1 status
tubenidata - ~FEE1 Register 1 data
tube~stat - &FEE2 Register 2 status
tubefldata - &FEE3 Register 1 data
tuben3stat - aFERA Register 3 statue
tubefladata - Register S data
tubsR4stat - aFEES Register 4 status
tubeR4data - &FU7 Register A data

OSWOSD TYPE DEC~TIoNs
GRAPH_OSWOafl - fast graphics update
nrsK_oswonn - STE hard disk read~write (unirr.lemented)
RESERVED - &FD was tert OSWORD
CRTC OSSIORD - SFC curmor, soft scroll, mouse etc
FDC_OSSORD - iFfl seek/read/write
RESEflVZfl - bik xfer onward

FILING SYSTEM Co~*4ANDs

```
usfind - &FFCE o~nIclose file

osgbpb - &FFfl1 read~vrite open file

filefdc_atri - 0 select drivt/flt-flhl~.

fdc_atat - 4 nRC, INDEX,RMV,~C, -
fda_=td - 4 REsTORS~sEEKIRtAn/WRrTE fdc_trek - S 0-79
fda sect - 6 1-10
tdc_data - 7 data for PEAfl/watT~, track for SEEK

"MI DECLARATIONS
nnd code &OnOO MM! handler location - FIXEnt
=r_stat - &0D02 direction patch location
nrni rd - &ODOB direction patch location wr aODOE direction ~atch
location

NMI ZERO PAGE WORKSPACE
nmiwso &A0 temporary status storage
nadwal - &A1 end of ccrmnand semaphore
nmiwsfl &AS
nmiws3 &AS
nmiws4 - &A4
nTniws5 = &A5
nrniwsS &Ae
nn!ws7 &A7
;Aliases
fdc bans - nmiwso indirect all accesses to FDC
fdo base 10 - nmiwso
fdc base_hi - nmiwsl

BEGINNING Cr 512 TUBE COOS
2100 JMP &2500 not one of the following
so j~ to OSWOTID ~FA code
2e0a atp .0-il_OSSORD graphics update?
2005 REQ &262B ~--4 to_gra~h_'trite
2107 CbtP &~cRTC_OSWORD cursor OSSORD?
Zecs %zQ ~2O26 beq to_nuword_fe
260B CMP &~DIs~_oswonn hard disk OSWOflfl?
280D BEC $2825 b&~q to_oaword_fe
2B0r CMP *TDc_osWOstn floppy disk contro~lnr OSNORD?
```

```
2B11 EEC &le2E beq to_onvord_fda
2613 JkP &2600 else default code

n-ny RESERVED FOR TRANSIENT STORAGE Temporary storage for FDC
operations
2S16 cpa_dras EQUD C
281A tam_statu. EQUB 0
211Th &~rw aid ~fl C
2S1e .s,ctor_count 501,3 0
2am error_-sir EQUB 0
2S1E ran_c-rd EQUB 0
261r &~vatchdogl EQUB 0
2020 -watchdogz scum 0

screen address mn~d mouse pointer co-ordinates
2S21 &~moua.jr~hi tat's 0
2022 &~mau5e~~lo taun 0
2S23 mouse_x_hi ECUB 0
2S24 won,._x_10 SOUTh 0

The following code overcomes the liatted brancbin4 range of
conditional it-s from the initial
entry by converting them to direct unconditional jumps to OSWORD fe

2B25 JMP &2c2r J~ to CO-On RTS
to osgord tc
2S2e JMP ~2A7F write 6B~5 data - atip onword_fc
to_graph_write
2B2a JMP ~29r8 Graphics output - 5~ graph_write
to osuord fdc
282t axp &28E0 Floppy disk - j~ caword_fdc
-newevont
~sS1 aMp &2Arc J~ to event handler - eventeode
.n~w_irq
2834 JMP &2ne0 J~ to nouirql

**ttt~ The main operational mutinca now follow

get one byte fr~ register 2
.getflldata
2837 LDA tubeUZatat Teat bit 7 of n 5tatus
2S3A BPL &2940 Not set - branch to tu$~_oavrch
```

```
2B3C LDA tube~data Read fl data
20SF a's
-tube oswrch
2640 LDA tubeRlatat Test bit 7 of Ri statu5
2eI~ SPL get~oata J~ to getfl if set
2S4S LDA tubeRidata Read Ri data
2646 JSR os'&~rch write character
264n JMP tube_osvrch Repeat; write on. byte to register 2
write fl
284e SIT tubePUstat Test bit 6 of fl stat
2851 avc write_~ Not set - loop until it is!
2853 STA tnbe~data write 't2 data
2856 RTS
Get one byte fr~ register C
read RI
2057 BIT tubeRcatat Teat bit 7 of Re stat
2e5A BPL read_Ra Not 5et - loop nntil it is!
2S5c LDA tubencoata asad R4 data
2asF RTS
write one byte to register 4
'trite RI
2S60 BIT tubea4stat test bit ~ of R4 stat
2e63 BVC write_Rd Not set - 1o~ until it is!
2S~5 STA tuben4data write RI data
29~ a's
write one byte to register I
write al
2069 BIT tubenistat Teat bit 6 or Ri stat
286C BVC write_Ri Not set - loop until it i5'&~
2S6t STA tubenidata write RA data
2671 RTS

Foe OSSORD for "1)1770 5 1~4" r's'~ floppy controller
the floppy controller uses five registers

0 special control latch
4 ca-and/status
5 track register
6 sector register
7 data register
&~'' wArtNIN~! - POSITION DEPENDENT CODE FOL~S

.fdc exec
```

```
2B72 InA *0
2S7~ S~A i2a1A sta ram_etatus ensure clear initially
2e77 WY *7 idy &~fdc_data - ensure drq not pending
2879 LDA C&AO},Y ida (ido_base ,y
2R75 JSR ~et&2data
287t STA &2a1n eta error_mask - save tar dSsk mt.
2981 JSR qet~data
2864 StA &2e1c sta sector_count - multi sector count
20S7 Jsrt getfldata jar getnidata - read coinmand from 1R6
289A STA &281E sta rain_cmr'd - for multi sector xfer
296D JSR &2072 jsr tube~rograrn - set dm addr in 186
zego LDY 44 lay &~fdc_mand
2092 rflA &291E ida ram
2895 STA (&AO).Y ata (fac_base},y - send cur-and to f&
2897 mm *&co I. it group S~4
2899 CMP *&CO were top two bits set 7
209e BNE &26cE bne fdc_e7 - No! leava to interrupt .
289n InA &~&3C
2S~r STA &2SiF sta watebdogi
28fl InA 40
2BA4 STA &2620 sta watchdoq2
20k? TAX Set up tic-out Counter5
2SAS WY *4 idy *fdc_stat - point to atatus reg
-fdc el
2SAA flEx
2SAn BNE s2SAA Loop fdc_ci until status goam busy
.fdc e2
2SAn LDA $1 I Te5t busy flaq
2SAr AND t'AO~~Y ANn (fac_base~,y in the status register

~INC! - the foliowina insttuction must not lie
on an a&Irtas where the bottom 3 bits are set
zael nno s28c5 Branch fdc_e4 If no longer busy
2Bn3 DEC &2820 dec watchdog2
29B6 BNE &2SAA bne fda_el

28RS nEC &2e1F dec watchdogl

2mB mit aQOXA bne fdc_ci - else tall thru on timeout

20nn Ink faFF and return a timeout error
```

```
28SF BWE &zecs bne fdc_es (will always branch)

-fdc e4
28C1 WY *4 Arty itdc_stat - get the status
2SCS LOA c&AO) ,Y ida (fdc_base ry
.fdc es
2Bc5 STA &281A Save status for return in ram_status .fdc_eS
2Bc6 SET Disable interrupts
2Sc9 asa 'nAEF generate mc event to return status
2S00 CLI Enable inter~pta
28CD RTS and return
.tdc e7
2Sct BPL &2SDF bpl fdc_e9 - ctp 1 co-wade return now
2Sn0 u)y *0 Others don't, no load ti'tSout counters
2sn2 Wx *0
fda as
28n4 LDA ~~9t6 ida status~nding and check Fnc atatna
28fl7 ENS &2scs bne fdc_e6 - it finished return status
2Sn9 flEx Ir not finished decrement counter
28DA mm &29D4 Not done, btanch fdc_en - wait a while
28nc oty
28nn BrIE &2eD4 bne fdc_es it still not co-lete. else
.fdc eg
2Sflr RTS Timer event returns status

mc OSWORD proper

&~OSWORD fdc
28E0 CLI , Enable interrupts again
28E1 cw Ensure operation in binary
29E2 asa qetfldata Read tube RZ data - get a cormnd
28E5 WrE aQete , It > 0 jnp to tube call 1 - else- - -
fdc OSWORD ret
28g7 RTS , Done!

I The5e routines claim or release the tube, set up the l'70 addresses
for the host taacbine,
or write data to nemory mapped 1-0 In SHEILA.

tube call 1
29E6 CKP &~i Master 128 co~nd?
2eEA BITE &28F3 No! - Try tube_call_2 CB/B+
```

```
Qote jsrt tube_setup Get tube params - store at rw_cmd &281B
2OEr JSR move_NM:_data Copy and adapt nmi routine
26r2 Jsn &293A jar Master_setup - set up regs
26r5 Jsn &2872 jsr fdc_exec - qet and execute connand
2SFe JMP &26E0 OMP csword_tdc - get another byte
tube call 2

2er9 t *2 aec ~'~+' coninand?

nero ant tube_call_3 no! - Try tube_call_3

28FF JSR tube_setup Get tube params - store at rw_cmd a2a1n
2~O2 Jsn move_NMI data copy and adapt the nm* routine
2905 Jsn &2932 jar nBc_setup - set up reqa for BBC
290S JSR &2972 jsr fdc_exec get and e~ecute command
290s JMP &28E0 imp OSWORD_fda get another byte tube call S

a(p *3 Claim tube required? 2910 BNE tnbe_call 4 No! try tube_call_4

2912 asa &2983 jar anti_claim - qet control of rynit
2915 JSK &2956 Get tube - ist tube_claim
2918 J,(p &26E0 JKp osuord_fac - get another byte
tube call_4
2919 CMP #4 Release tubs?
2910 nt;E &2926 No! branch other_out~ut
291r JSR &2s7n jsr tube_release - give up tube
2922 JSk &2990 Call fluff_release give up NMI
2925 Jxp &26E0 JMP onward_fdc - qet another byte

.other_output
2929 TAX port n-er in x
2929 JaR qet~data Read tube ~ data - get data byte
292c STA &FEOO,X atore in 4o~age, X - write it to ~rt
292F Jnp 62BEQ 3Wp onward_fdc - qet anothet byte

.SW_setup
2932 'Dx &~&e4 Sec_f& stat
2g34 Kny &~&87 sBc_fdc_data
2~36 Lok IaSO nnc_fdc_bane
293S -E '2940 bne Setupi - always
flaster_setup
293A LOX &~&26 Manter_fdc_stat
```

```
293c Lay &~~2n Master_fdc_data
2~3R LDA &~&24 Master_fdc_base
setupt
2940 STA &Ao sta fdc_base_lo - patch for indirection
2942 LDA farE Load in~age (Sheila) high byte
2944 STA ski sta fdc_base_hi
2946 S7X &n02 str rflfti_stat - Patch nini status read
2e49 JSR geta2data Are we perfcrminq write?
2g4c EKO &2952 zero is No! go to setup2 for read
294E STY aDot sty nifli wr - patch the rual write code
2951 RTS
setup2
2952 STY aDDS sty stat_rd - patch the nrni read code
2955 RTS

.tube_claim
2956 InA #&Cl A 1~3 to claim tube
2~5S Jsft tube_entry Call tube data transfer at &4O~
295R flea tube claim If c-C tailed, repeat until success
295n RTS

.tube_setup
295E LCX SD Fill in dma address

.fdc amal
~96O csn getn2data Read tube R2data - get 1 Cf 4 bytes
2~6a STA &2816,x Store it in epra_dina,x
2966 IflX
2967 crx *4 Done 4 times?
2969 Sflt &2960 No! - Repeat fac dinal
2968 JSR getn2data Read tube n2da~a get nw coffrnand
296s STA &2a1B Store tube n2data at rw-Crnd
2971 RTS

.tubejrcqram
2972 tflA &2s1n A - stored rw cmd - dna claimer type
2975 IsX *&16 sot X to cflra_dma low byte
2977 WY t&29 Set Y to cpm_dran high byte
29?9 asa tube_entry call tube data transfer at &406
297c RTS
-tube release
297D LDA &~&ei A - 129 to release tube
297r JSR tube_entry call tube code at &406 - release tube
```

```
2982 RTS
```

These routines claim or release NMI ownership according to whether tube data transfer is to take place or disc access is required. In practical terms only one of three facilities can be the current NMI owner, the disc system, the network handler (econet), or the tube. A copy of this routine's NMI code is kept at nmi_image (29DB) and transferred to page &D when NMI ownership is required. The routines are called by the code in tube_call_1 to tube_call_4.

```
.nu~i claim
2993 LLA iSAF Set n~ paged ROM service request
29e5 LnX *&C for N"r clajaL
2987 WY ISFF
2~s9 JSR oubyte Claim NMI routine
29S0 STY s299A Store oriqinal IlMI RON No- at nmi owner
29SF RTS
risi release
2990 ~r &299A Load Y with original ami_owner
299a LDA &~~8r set up Paged Row serviec re~est
2995 tox &~as for NMI release
299~ JKP osbyte Release NMI to previous owner
owner
299A ~Un 0
move N'S' data
~)( Move 32 bytes of NMI data to page 6D

.rmd nut
2990 IDA &29na.x ida n~i_irftaqe,x - (saved orig IlMI data)
29A0 STA &0DOO,x ata rud_code.x (Restore to NMI paqe &0
2~A3 flEx Completed 32 tim..?
29A4 SPL &299D No! - repeat tram nini_nut
29A6 ft~s
-nmi next sect
29A7 WY 94 idy tfdc_stat - status req offset
29A9 Lox (iAo~~Y ida (fac_base),y - qet the status
29as STA &281A sta ram_status - save for return status
291& AND &281D and error_mask - any error bits on?
29n1 nNt &29n5 bne ntai next exit - Yes' - abort it
29B3 LDA ~2BlE Ida ram_cmnd-test for groupi (seek ea
29e6 Ert &29D5 bpl arni_next_exit - Yes! Then finished
2~SS u)A &2810 ida sector_count -get remaining sectors
~~SS CMF 41 c~ *1 - is it the last one?
29BD BEQ a2SDS beq nri_next_exit - signal final mt
29nr DEC &261c * dec sectot connt - count down sectors
```

```
29c2 WY 56 ldy &~tdc_sect
29c4 LDA (~AO),Y ida cfdc_base~,y - get the sector req
29c6 CLO Clear carry tlaq
29C7 AbC *1 Move to next sector
2909 STA (&AO~.Y sta (tdc_base~,y - and insert new value
29GB LDA &2S1E ida ram_0=4 get read~write
29CE AND &~&F3 Remove head settle delay
29D0 WY &~4 Idy &~tdc_crand
29n2 STA (&AO},Y sta (fdc_base),y - fire off again
29D4 RTS
.nni next exit
29D5 LDA &~SFF Flag an FDC event
~9D7 STA &29F6 Set status~nding to return statu5
2~nA RTS

-flrai_image
aBNE pni save context onoo
29nc LDA SEECO 6945 address register ODOl
29Dr Ann Get error bits 0n04
29E1 CKp &~3 + Busy ~ ODOS
2StS mm ~2SED Final interrupt - nimi_exit ODOR
29E5 LDA tuben3data tither this or the next inn ODOA
2~tS StA tabertadata patcb.d to paint to FDC ODin
fl9En puk Restore context 0Db
29Ec RTI Return from interrupt onli
nad_exit
29~ TYA 01)12
aSEK Pr" Save Y onla
29Er JSR &29A7 Next sector or finish 0014
2S72 PEA Restore context 0017
29F3 TAY Restore y ODiS
2SFA era Restore a 001~
Z9r5 RTI Return fr~ interrupt OnlA
status~ndinq
29r~ EQUB C onin
- return
29F7 RTS Return from OSWORD

Copy bit patterns to 6502 for block write & ~raphics

This part of the code fills 6502 screen RAM with bit patterns sent
across by the 186.
This can either be font bytes in a1~ha ~de or virtual graphics screen
```

```
bytes in graphics
mOde Operations are:
a) send screen address MSB - or zero to sto~
b) send screen address LSB
c) send S font bytes, Ct 1 byte for background~space fill
d) send sync byte - or iFF to repeat from a)
a wait for sync response f) increment pointer, qo to C
-graph_write
2SFS eLr Enable interrupts
QSFS cLD Ensure we operate in binary
next addr hi
29rA JSR qet~data nead fl for hi-byte until data received
29FD SEQ &29F7 no data? - finished - go to RTS
29FF S~A ~71 Store n content. at zero page &71
next addr 10
2AO~ LDA tube~stat Read '12 status until ready
2A04 BPL &2301 Not ready? - repeat, branch
next addr 10
2A06 LDA tube~data Read 't2 for 10-byte
2A09 TAr and keep in Y
2A0A LDA *0 Clear low byte
230c STA &?O Zero ~7O (real low byte is in Y)
```

Writing to the screen is optimised for maxirnum possible speed, hence these routines are written 'longhand and so avoid the use of counters, instructions to decrement them and the tests to check when the loop is complete. Spaces are further optimised because they are the most frequently written character and only need a sinqle fill byte. They are therefore the easiest to write too as no reading of tubendata is required between bytes. `transfer_loop1` writes spaces, `transfer_loop2` other bytes.

```
transfer_loop1
2AOE LDA tubenstat Read n2 status - wait for vayne byte
2A11 BPL &2AOE Repeat until S bytes ready in ni buffer
2A13 LDA tuben2data Read R2 data-remove sync byte~cbeck chr
2A16 SEQ s2A43 If 0 qo to transfer_1c0p2 - qet S bytes
2A1B OMP *&F~ ~ither a space or the end of output
2A1A EEC &29FA aFF-end. branch next_addr_hi to repeat
2A1C LDA tubeRidata Read al data for sflace fill pattern
2AiF STA (&70} ,Y * and move same byte into main screen RAM
2~~1 INY S times an fast an ponsible
2fl2 STA (&70}.Y
2A21 INY
2~~5 STA (s7O ,Y
2A27 INY
```

```
2A28 STA C&70 ,Y
2A2A INY
2112 STA (&70),Y
2fl9 INY
2A2E STA (670 ,Y
2A30 rNY
SASi STA (&70 r
ZAS3 INY
2A34 STA (&70) Y
SA~6 INY
2A37 BNE &2AOt wait for sync unless on page boundary
2A39 rNC &71 Increment hiqh byte first
2A3n flPL &2AOE check for wrap around - bpl
transter_loopt
2A3n LDA t'40 40 back to bottom of screen
2A3F STA '71 by resetting hi byte
2A41 SME a2A0E cuickest j~ to transfer_loopi
transfer_loop2
2A43 LDA tubauldata Read Ri data - 1st byte - 7 left
2A4~ STA c&703 ,Y
ZACS INY
2A49 LDA tubeuldata Read at data - 2nd byte - 6 left
2A4c STA (&10) ,y
2A4t INY
234r LDA tubeRld&ta Road RI data - 3rd byte - 5 lett
2An2 STA (~7O~.Y
2AS4 PHY
2MS InA tubealdata Read Ri data 4th byte - 4 left
2A56 STA (&70),Y
2A5A INY
2A55 LDA tubealdata Read ni data - 5th byte - S jeit
2ASE STA (&70 ,r
2A60 INY
ZASt Ink tubealdata Read at data - 6th byte - 2 left
2A64 STA (&70).Y
2A6~ INY
2A67 LDA tubenidata Read Ri data - 7th byte - 1 left
2A6A STA (&70) .Y
2A6c Iwc
2A6fl LDA tubealdata Read ni data - 8th byte - buffer empty
2A70 STA (&70 ,Y (unless already re-filled by lee
2A72 INY
2A7S nnE &2AOE gait for sync unless on page boundary
```

```
2A~S INC &71 Increment high byte first
2A77 BPL &2A0E Check wrap around - b~1 transfer_loopt
2A79 LDA &~&40 Go back to bottorft of screen
2A7B STA &71 by resettin4 hi byte
2A7D mm &2A0E Branch to transfer_loopi- - - alway5

at? OSWORn - programs CaT controller. handles initialisation

onward fc
2A7F JsR get112data Get CaT controller tegister number
2Ae2 ~r &2A90 Return on 'FE
2Aa4 STA 'rEDO write 6845 CRTC address register
2AS7 JSfl getfldata Get data byte for 6945
2AAA S~A &rECl write 6945 Cfl~C data register
2A9D JMP &2A7? Repeat until &rr received
onward to 1
2A90 TAX Get cornand code
2A91 INX In it aFF?
2A92 nrc exit Yes! - finished
2A94 rRx Is it arE?
2A95 BNE &2Aoc bn, onward_fc_2 - initialise ~ouse code
2A97 SE: set interrn~t disable
2A9e LnA ir~1 Load oric;inal mt. 1 vector la-byte
2A~n S~A &2aeA Store in oldirqt+1
2A9E LDA irqvl+1 Load original mt - 1 vector hi-byte
2AA1 s,!A &2nen Store in oldirql+2
2AA4 LDA &~newirql+1 "OD 25e new_i~ low byte
2AA6 STA irqvt Stare in 'Rd vector low byte
2AA9 LDA *newirql fl&v 256 new_lrq high byte
2AAB STA irq;ri+1 Store in rRo} vector hiqh byte
2AAE CLI re-enable interrupts
2AAF LDA SO Enable user-port input
2n1 STA ~rE62 Store An up_rdvrt - userport data dir
2AB4 LDA &~&98 enable CB1, CB2 interru~ts
2As6 S?A artet wrtte interrupt enable register
2An9 LDA &FE6fl Read auxiliary control register values
2ABC AND Si leave PA, disable Pn latching a timers
2ABE STA aFE6B write auxiliary control register
2A01 LDA aFEEC Read peripheral control register
2AC4 AND #&F Clear Cal, can bits
2Ac6 STA &FESC write peripheral control register
2Ac9 JIW &2A7r J~ OSWORD_ic to wait for cormaends
-onward fc 2
```

```
2ACC nix is it 'FO (intercept events~?
2ACD mm &2AE6 bne os~ord_te_3 - No, continue
2ACF LDA eventvec Yest~ - load event vector low byte
2AD2 STA oldevent+1 Store it
2An5 LDA eventvec+1 Load event vector high byte
2AD8 STA oldevent+2 store it
2ADB LDA Inewevent Mon 256 Load new event low byte
2ADn STA evontyoc Replace oriqinal
2AE0 LDA taewevent DIV 256 Load new event high byte
2AE2 STA aventwc+1 Replace original
2A&5 JMP &~A7r Jump o5word_fc to wait for cor-ands
onword fc S
2AE6 "ix is it arc (waa write mouse port
2AE9 mm exit No! - continne
2AEB JMP &2A7r rest - next coninand
exit
2AEE RTS return from OSIfORD
ide event
QAEF LDX &261A icix ram_status - mc status ret~d in x
2Ar2 WY *0 zero Y for no',
2Ar4 STY &29F6 zero the status~nding flag
2AF7 LDA faA Generate event 10 - wa result waiting
-oldevent
2AF9 EQUB &4C JMP instruction - for saved_vector
-event 10 Jump address of oriqinal event code
2AFA EQUB C
-event_hi
2Arn EQUB 0
TWO KEY ROLLOVER pROC~SStflG DONE HERE ON VSYNC EVENT
.evtntccde
2AFC CMP *4 Event 4 Cvmync)?
2AFE BNE &2AF9 Not - branch oldevent
2B00 LDA &29r6 Read Status~ndInq - FnC status 0/s?
~flOS BEQ &2308 no! - Continue
2B05 Jart &2AEF asa fdc_event to ret-pending FflC status
-eventccdel
2308 JsR keyscan asa sHFT~cTRL scary at &2n7n
snoB PH? save etatus flags on stack
250c LDA keyl Read current ~ey ~ressed
2HO~ Ann i&7F Clear top bit
2B10 PLP Recover statu3 flags from stack
2B11 pHP then re-save them again for SHIFT
2B12 BPL anere Branch no_etri if CTRL not pressed
```

```
2B~4 OFLA &~&8O CTRL was pressed - set top bit
no ctrl
2316 TAX store current key in x
2B17 LnA key2 Read last key pressed
2B19 AND &~a7r Clear top bit
2pm Pt? Recall scan status for SHIFT
neic nvc &2a20 Branch no_ahift if SMIFT not pressed
2a1E eRA isso SHIFT was pressed - set top bit
no shift
2B20 ~AY store last key in Y
2B21 LDA *4 Re-load vsync event
2B2a asa &2AF9 Re-issue original event

The current key (if any in now in X - The top bit always set
if CONTflOL was pressed, even when no other key was pressed.
The previous key, (if any) is floe in Y - The top bit always set
SHIFT waz pressed. even when there is no previous key press.

asyne_ca-and
2B26 asa read_n4 Read tube a4 data
2329 BEQ &~B7A exit if n4 - zero - finished
2n2n TAX store in x
2n2c flEx Decretnent n4 value
2B2D BNE &~ssE Pt4 was > 1 - go to asyne_mouse

upnA~z 6845 CRTC CONTROL ksGISTEflS

r4-l
.anync_cursor
2B2F JSR read_n4 Read tube aA data - get 6945 req. addr.
2BS2 STA &FEOO write 6045 CRTC address reginter
2BS5 asa read n4 Flead tube R4 data - get the data byte
2B30 STA 'FEC! write 6845 Cfl~C data register
283B JMP &2B26 aepeat async_cointand
READ USERPORT AbiD TRANSFER DATA

r4~2
-async_mouse
ZB3E flEX 2 - mouse
2n3F HIrE &2R64 flA was > 2 - go to asyne_leds
2B4i LDA arAX fl4 wan 2 - read arax
2544 pnp Store flags for z
2a45 LDA userport Read unerport
```

```
2B4S eLp Recover z flag
2a49 BEC &2n4F Not AKX, tr~ba11?-brancb asyric_mouse_1
2n45 ROL A AMX buttons ate top bits
2n4c Rot A so trove them to the bottom
2B4D ROL A
2B4E ROL A bits 5, 6 and 7 are now 0, 1. 2
&~async_mOuse 1
2B4F AND *7 mask off the buttons
2B51 EOR 57 invert the bits
2553 Jsn write_at write tube nidata to send to 1S~
2B56 IDX 43 set count for 4 bytes of co-orda
&~async_mouse 2
2n~S ~A &2e21,x acad monSeft!~hi.x
2953 asa write al write tube aldata - send mouse data
295E flEx from &2821
285r Ept &2958 four times So branch asyno_nouse_2 if
inco~lete
2R61 abip &2n26 JMP async_cc~and

SET KEYBOARD STATUS AND LEns r4~3

-asyno leds
2BG4 DEX
2365 BNE &2526 z4>3 - read tube aqain at asyne_co-and
2B67 JsR read k4 Read tube nadata
2B6A TAX put in x
2n6n LDA saca set up FX 202 - write keyboard status
2B6D LDY #0 Y - 0, X paramter from RA
2n6F ask osbyte set keyboard statua according to R4
2B72 LDA t&76 set up FX lie
2874 JSR osbyte Set keyboard LEDs
2377 JKP &~B26 3)4? asyric_command

-exit
2B7A RTS

-keyscan scans shift and control key presses
2n7n etc clear carry and overflow flags to
2a7c CLV , indicate botb SHIFT+CTRL scan re~ired
297D a}qp (keyvec Exit through orig- Sibri vector routine,
```

WARNING! MASKABLE INTERRUPT CODE FOLLOWS

X and Y must be preserved here. The original contents of A are preserved by the Has, in zero page &FC. 'rhia also must be reloaded before exiting - with an RTI it we process the interrupt . or if we j~ to the original vector because we're not interested

```
.newirql
2B00 LnA &FESD Load Ant flag register - in it 'muse?
2flS3 AND tale rnask-00011000 -cnl and c32 active edqe?
2B85 aflE &2n8c AL leant one - qo to new_ir~code
2n87 LDA &FC Not CB1 not Cfl2 - restore A, foraet it

.oldirql Drop thru to here on any other mt
2369 EQUB &40 JMP - for saved_original vector

-ir~1c saved it- address of oriq md code
2neA aRK

.ir~hi
Zafin BRK

- new~ir~code
2360 STA &2e2A ~asked mt flags in RAM in itt_copy
2B6F LDA userport Load VIA input reg S - tracker ball?
2B92 STA s2c2n store it at orb_copy
2n95 AND fals Mask it - these bits always hi for ANX
2B97 OMP noth set?-if either low mINT be t~ball
2B99 BEQ &QnAA Yes! - AMX no new_ir~code2
~fl9fl LnA *0 Mark as tracker ball by atorinq 0
QflSfl STA &2c27 in AMX to override default
2BA0 LDA IS y quad signal is different to default
2BA2 STA &2c28 store it in RAM - tn rousey~qnad
2BA5 ~A Islo and S quad signal is different too
2nA7 STA s2c29 store it in RAM in mouse_x_quad

-new_ir~code2
2BAA tDA &2c2A masked mt flaq req copy - ifr_copy
23AD Ann &~&1o 'qank with 00010000 is it x?
2nAF BEC &2BnF No! - must be Y, branch new~ir~cnde5
2BB1 LDA &2c2E Get pezi~heral control copy - fler_copy
2flR4 EOR lab Invert pos~neg edge bit
2flfle STA &2C2E Store in per_copy - update Pcn edge
23B9 LDA &2CSC toad x_edge - get edge triggering mode
2BBC FOR #aFF Invert it
```

```
2nnE STA &2c20 re-store in x_edge
2nd EOR &202B ouad sig, orb_copy, invert if pos edqe
2nc4 AND &2c2e X component in rnousejt~q'aad
2nc7 BNE &2flfl4 necrease? - Yes - branch new_iz~code3
2nc9 INC &2824 Increment the low byte - monse_x_lo
2ncc BNE &ZBDS and if that beco~s a - new~ir~code5
23CE INC &2823 Inc the bi~h edqe - TROUSC_x_hi too
2nn1 JMP &2BDF Jump to new~iz~code5 for Y component

new ir~code3
~Bfl4 IDA &2824 Dec - so load the low eaqe - rouse_r_la
2nD1 BNE &23DC If not 0 jump new_ir~code4, skip
2~n9 nEC &2923 Decrement of mouse X hi

.new~ir~ccde4
2BDC DEC &2824 Decrement raouse X lo

.new~ir~code5
2BDF IDA a2C2A Read masked mt flag contents- ifr_copy
2BE2 AND &~a Is there any input from Y?
2nEc flRO &2C14 No! - jtirap new~ir~code8
2BE6 LDA a2C2E Get peripheral ctrl register - per_copy
2nES EOR &~&40 Invert pos~neg bit
2n~s STA &2C2E Update peripheral etri req - PC?_COPY
2nEE LDA &2c20 Get edqe triggering mode - y_edqe
2nF1 Eon SaFF myort It
2nF3 STA &2c2D Store in y_edge
2nF6 EOR &2c~B Quad 5143. Invert it pr's - orb.co~y
2BF9 AND s2c2S Look at mouse~~quad
2nFc "~y &2009 Incr~nt? - No! branch new~ir~codeG
2nrE INC &~S22 Incr~~nt as per X motise~~1o
ZCOi mm s2c14 Not zero? - branch new~Ir~codee
2c0S INC &2821 Increment moune~~hi
2c06 axe &2C14 J~ new~ir~codea

-new~ir~code6
2C09 LDA &2e22 Decrement as per X - mouse~~1o
2COC mm &2c11 Not zero? - branch new~ir~code7
2c0E DEC &2621 Decrement rnouse~~hi

-new~{r~code7
2c11 nEC s2e22 Decrement mouse~~lo
```

```
-new~ir~codeS
2c14 LDA &FESC Read real peripheral control reqiater
2017 mm preserve the bo~tnm nibble but set
2c19 ORA &2c2E the remainder to our copy in pcr_Copy
2C10 STA aFEEC Re-write peripheral control register
'air LDA lila Load cHi and cB2 active edge bits
2021 STA &FEGD write IFR to clear our interrupt
2c24 Lak &rc Recover A to intorru~t entry state
2c26 nTr Return from interrnpt
2c27 -arAX EQUn aFF default is AMX (0 - Tracker)
2c28 .rncuseflr~qnad EQUB 1 0Th - AM)C, (OS - Tracker
2c2~ mouse_x_quad ECUB 4 AMX, (10 Tracker)
2c2A -ifr_copy flaic RAM copy of Ira state
2C2B -orb_copy BRK RAM copy of user port B
2C20 K_e~e BUK defines pos/neq edqe triggering
2c2t -y_eaqe BEUC detines p05/nag edge triqqerinq
2C2E -pcr_copy flRK Local copy of PCR
osijord fe

2c2r RTS

NOTE:- A local copy of the pcripheral connrol register is

naintained at per_copy because someone Is reprograr~Inq the VIA when
they shouldn't be,
leading to mouse reversal suspect a Master 12a hardware prohlera (2?)

HAFW DISK OSWCflfl

-OSWORD fe

2C30 RTS not ir~1emented

-pad EQUB 0

t**~*~*~****t******* EMD OF CODE ~
```

OSWORD &FA source listing

It should be noted that major bugs exist in the OSWORD &FA code which are virtually guaranteed to cause problems when the function is called from DOS Plus. Users are therefore warned that they should

proceed with extreme caution when testing new routines which call this code and should consider the following points if (i.e. when) problems are encountered.

First, the original designer of the code seerns to have been unaware of various differences between the three host machifles' MOS variables and functions. For example OSBYTE &n has no fundion in a model B or B+ but is cailed in the routine at &2517 to d'e& the current shadow setting. Also reading or writing the contents of ROMSEL at &FE30 may have undefined implications in a model B.

Worst than this however, is that certain areas in memory mapped I/C, specifically ACCCON at &PE34, are not common to all versions of host. An attempt to read ACCCON in any host except a Master produces a nonsense result. Although this is in itself harniless, the result of an attempt to write ACCCON in either a model B or B+ (as at &26E3) is totally unpredictable and will almost certainly cause an immediate crash.

The code shown here is downloaded from the 512's ROMS on either a hard or a soft break. It therefore does not vary with the host type and can be guaranteed to hang the system in a TnodeI B or B+ because of the instruction at &26E5 which attempts to re-instate ACCCON's contents as originally recorded at &251C.

In a Master 128 ACCCON is designated as read/write, but readers should be aware that in spite of this fact the CSWORD &FA routine is not reliable in a Master either. While investigating the disassembly

provides no clue as to a reason, in all the tests conducted by myself calling OSWORD &PA from DOS Plus hangs a Master 128 too!

In addition word transfers seem to be inconsistently unreliable. Single byte transfers of types 0 and I and page transfers (types 6 and 7) appear to be reliable provided that writing to ACCCON is prevented. However, word transfers from the 512 (type 2) sometimes do not function correctly, sometimes doing nothing at all (including the transfer) occasionally hanging the system.

On balance however, type two transfers probably function correctly more often than not. The fault is extremely obscure, but so far as testing has been able to determine, it appears to depend on a combination of both the host target address and the type of transfer performed the last time the routine was called.

The only consistent fact to emerge from lengthy tests is that if any specific type 2 transfer in a sequence of transfers does fail, it can be reproduced and will fail consistently unless one of the attendant conditions is changed. if this particular problem is encountered users are advised to avoid it if possible by changing the target address rather than anempting to find the cause.

```
OSWORD &FA code
ZERO PAGE &~.oRKSPACE
```

```
&7o - LSB of parameter block in host RAM
&71 - MSB of parameter block in host RAM
&72 - storage of current paqed no~ number
&73 - Storage of current shadow setting
a74 Lsn of host RAM data address
675 - NS3 of host FIRM data address
&76 - MSS of 1en~th of data to be transferred
&77 - tse of lenqth of data to be transferred
flOT~&~- rarameters &76 and &77 are NOT in error- They
are reversed from the usual low-byte-high-byte format

Zero page 1105 variables
&F4 - romnum NOS copy of the current PtOM nuraber
Mos calls used by this code
osbyte &FrF4

SHEILA ~ty mapped I/O addresses
&FE30 - ronwel The paqe no'. select latch
&FE34 - acecon The paged ROM access control

Initia1 entry to thin code is after all other poanibilitien have been
exhausted,
by tirst the SOS. then by 6502&~sYs-
If the call is not an OSWORD aFA control is passed to the sos default
handler,
giving the familiar 'Bad corrinand error which is passed acroas the
tube as described
in chapter 4-

2500 CLa clear carry for tube claim
2501 nad a2505 branch to oswc~ tent - alwaya
default_handler
2503 flRK Contains the address of the Has
2504 nRK unknown OSWORD' default handler

&~oswotd teat
2505 alP taFA 1 15 this an OSNORD &FA?
2S07 BEQ a250C Yes - branch OSWORD_confirrw~d
2509 CMP (&2503) No - it- to default_handler

&~OSWORD_confirmed
```

```
250c ST~ &70 Store low byte of paramter address
250E STY &71 store hi byte of parameter address
2510 PHA Stote A on the stack
2511 Ink &~&FB Set u~ OSBYTE 251 to read
2513 inX 40 the state of the host's currant
2515 WY &~&FF shadow/main RAM selection
2517 JSR osbyte Current settinq returned in x
2SlA STX &7a Store current shadow setting
251c LDA accoon Read contents of &FE34 in SHEILA
251r PHA Store it on the stack

-tube claim
2520 LDA jac? Load tube claim identifier (This
code pretends to be a video dine
2522 JSn tube_entry claim the tube
2525 SCC &2$20 Failed? - repeat till success
2527 WY $0 Index to para~ter 1
2529 LDA ('70) .Y Read total ni-er of parameters
2523 CMP &~&fl an - RAM access, jE paged RON access
252~ PEP Store flags - z - 0 means normal MM
252E LDA &F4 Read MOS co~y of curr- paqed flox nurter
2530 STA &72 Store in zero pa4C
2532 LDY &~&D rndex ot n~mory access byte parameter
2534 LDA (&70) ,Y Read ~mory access type byte
2536 TAX Transfer to x
253~ WY $2 Inde'r to host RAN address parameter
25S9 Ink (&7O~,Y need tsB of host memory addtess
25S3 STA &74 store in rero page base
25at INY Increment index
253E LDA &70) ,Y Read MSR of host memory address
2540 STA a75 store in zero page base
2542 PLP pnl1 flaqa stored at &2$2D
2543 BEQ a25a5 Branch on normal RAM access
2545 TXA Kernory access type byte to A
2546 pHA Store memory access type on stack
2547 nm 4&40 Isolate screen mnmOrv bit (Sm)
2549 BNE &255E Not ~ero means write screen RA)q only
254n TXA Recover access type byte aqain
254c AND &~&20 Inolate screen address bit (m~a}
254E UNE &2554 Not zero means use shadow screen

set main access
2550 rnx 40 zero in 'C
```

```
2552 EEC &2556 Branch select ram - always

-set ahadow access
2554 Wx *1 Set access for shadow screen

select rem
2556 LDA t&6c OSflr!E lOB sel. 3cr. for direct access
2558 asa osbyte on contents of 'C - 0 - rain, 1 - shadow
255a JMP '2577 Ju~ get_roTh_nun~er
255E rnA *&e4 OSBYTE 132 read top of user RAM (HIMEM)
2560 JSR osbyte ROturns X - low byte, Y - high byte
256j CPY laflO HI-i &E00o? Yes means shadow
256$ BNr &256F no - branch to shadow_test
2567 LDA #1 Load A with 1
2569 CKp &73 cor~are to current shadow setting
25~n nfl& &~554 Not equal branch to net-shadow_access
256D BEQ &2550 Branch set_main_access - a1~ays
shadow-test
2S6F LDA *2 Load A with 2
2571 CMP &73 Co~are to current 5hadcw setting
257S BNE &2550 Not equal - ~ranch to set_main_access
251$ BEQ &2564 Branch to set_shadow_access - always
.~et_rom nunber
2577 PM pull memory access typo trom stack
2576 TAX store in 'C
2579 AND &~&1O Isolate nOM type nunber (bit 4
2573 BNE &2565 Not zero - use clirrent no" nuriber
257D Tm Recover memory access type byte
257& Mm Isolate required RON nunber (bits 0-3)
2580 STA &F4 stare in MOS copy of ROM flo.
25B2 STA roawel Store at &FE30 in SHEILA
- RON_nnrnher set
2565 ~~Y &~&A Index at transfer lenqth paraneter
2567 InA (&70) ,Y Read LSfl of transfer lenqth
25S~ STA &77 Store in zero page
2585 INY Increment index to parameter Sn
25S0 LDA (&7O~,Y Read NSa of transfer length
Q5SE STA &76 Store in zero page
25g0 ORA &77 Check transfer length LSB NSfl - tero
2592 BNE &2596 Not zero - continue
2594 BEQ &2604 LSS = NSE - 0 means complete
2596 InA &77 Reload transter length Lse
2598 BEQ &259c Inteqral page transfer?
```

```
25SA Inc s76 No - Increment transfer length NSB
25~c my :ncrement index to flaraneter 'C
259t LDA (&70 ,Y Read transfer type (0 to 3, 6 or 7

set_transfer_type
259? PHA store transfer type on stack
25A0 IDA &7~ Reload transfer length Lsn
25~~ ~EC &25n5 It LSB - 0 transfer is whole paqsn
25A4 ILA 676 Load transfer length MSB
25A6 CMP #1 single page
25kg BNE &25n5 NC - it~s a multi-page
25AA PLA Recover tranater type from stack
25AS PRA and re-store for further time
25AC ~~~ Is it type 6 or 7 (256 byte transfer)
25AE BCC &25B5 Yes branch to get_address
2513 PTA Neither - qet transfer type
25n1 SEC Set the carry flag
25B2 SBC *6 subtract 6
25B4 PHA Store the resnit on the stack
-get_address
2535 LDA &70 Load the low byte ~arareter address
25n~ CLC Clear carry
25n8 ADC #6 Add 6
25SA TAX Transfer to x
25nn LDA #0 Clear A
25an loc '71 Load the high byte parameter address
253F TAX Transfer to Y
25c0 PLA Recover the transfer type byte
2SC~ PHA and re-atore for further use
25c2 JSR tube_entry Call tube host code
25c5 LDX &77 LSB at transfer lenqth into r
25C7 PLA ~ecover transfer type
25c9 CLY 40 Clear Y
25CA CMP *0 Transfer type 0? Cl byte to host)
25cc BEQ &25EC Yes - branch to jur~~_type_0
250K CMP 41 Transfer type 1 (1 byte from host
2500 BEQ &2607 YES - branch to type_1_transfer
25D2 CMP #2 Transfer type 2 (2 bytes to host)
25D4 BEQ &261fr Yes - branch to type_2_transfer
25D6 CMP &~3 Transfer type 3 (2 bytes fror host
25DB BEQ &2e4A Yes - branch to type_a_transfer
2SDA CXP #6 ~ransfer type 6 (1 page to host
S5~C BRQ i2SEG branch to it-_type_6
```

```
2Sn~ ~~~ *7 Transfer type 7 (i paqe from host)
26E0 BEC &25E9 Branch to j~_type_7
25E2 LDA #0 must be finished
25E4 flZO a2604 indirect jump to release tube

.jump_type_ 6
25E5 JMP &2675 Long jump to type_ 6_transfer

.jump_type_7
25n9 JMP &2eA3 Long j~ to type_7 transfer

.jump_type_C
25Ec JSR &2SF4 waste some time

-type_zero transfer transfers single bytes from 512
to host at 24~secs(byte
25FF LDA tubeR3data Read data from tube register 3
25r2 STA (&74).Y Store it at indexed host address
25F4 JsR &2er4 waste some time
25F7 INC &74 Increment host address low byte
2SF~ ~~~ I If we havgn~t crossed a page boundary
25FB INC &75 else incr~nt the addrems high byte
25Fn DEx Decrement the count
25FE BNE &2SEF Finished 256 bytea? No - Re~at
2600 DEC &'' Decrenent '(SB at length
2602 BNE '25FF and go back for the next byte
2604 CMP &26nA J~ to release_tube

type_1 transfer Transfers single byte. from host to 512 at 24 usecs/
byte
2607 IDA (&74 ,Y Read data frO~ host memory
2S09 STA tubea3dnta Write it to tube register S
26OC JSR &26r4 waste some time
260? INC &74 IncreTmOnt host address low byte
2611 BNE &2615 If we haven~t crossed a page boundary
2613 INC &75 Incre~nt the boat addreaa high byte
2615 ~EX and decrement the count
2616 BNE &2607 and go back for the next byte
261e DEC &7~ I Decreemnnt MSfl of ~enqth
261k mm &2c07 I Not tern - not finished
261c JMP Jump to release_tubO
type_2 transfer Transfers words from 512
```

```
to host at 26~secs~word
261r JsR &26F4 Waste some time

.2_bytes_in
2622 LDA tuben3data Read data from tube register 3
2625 STA (&74) ,Y Store it at indexed boat address
2627 INC &74 Increment host address low byte
2629 ~~E &2620 If we haven't crossed a page boundary
262E INC &75 Incre~nt the host addresss high byte
262D NOP Delay for 2 cycles
262F NOP Delay for 2 cycles
262r tnA t~benSdata Read data fr~ tube register 3
2632 STA (s74) ,Y Store it at indexed host address
2634 INC &74 morement host address low byte
2636 BNE i26aA If we haven~t crossed a page boundary
26a0 INC &75 Increment the high address byte
263A JSR &26F3 waste 12 clock cycles
263D NOP Delay for 2 cycles
263E NOP Delay for 2 cycles
263F DEX decrement the count
2640 DEX decrement the count
2641 BNE &2622 Not zero? branch for next 2 bytes
2643 DEC 576 else decrement the MSfl of length
2645 BNE &2622 Not zero? branch fro next 2 bytes
2647 JMP a26flA Jua~ to release_tube

type_3 transfer ?!ransfers words frOTh host
to 512 at 2~jtseca~word
264A LDA (&74},Y Read data tr~ host rn-cry
264C STA tuben3data write it to tube register 3
2e4r INC &74 Increment host address low byte
2~5l DEC &2656 tf we haven't croaned a ~age boundary
2653 NOP Delay for 2 cycles
2654 B~R S265e Branch if page boundary crossed
2656 INC &75 rncrement the host address high byte
2656 ~~A a73 waste 5 cycles (loads shadow setting
265A ~~A (&74) ,Y Load tram indexed host a-cry
265C STA tubeaSdata write to tube regAster S
265? INC '74 Increment the low address byte
2661 BEQ &2666 If we haven't crossed a paqe boundary
2663 NOP Delay for 2 cycles
2664 BNE &2665 If "C haven't croased a page boundary
2666 INC &75 else increment the hi~h adarcan byte
```

```
2666 JSR &26F3 waste 12 clock cycles
266n DEX deore~nt the count
2G6c DEX decretment the count
266D ~~~ &264A Not zero? branch for next 2 bytes
266F DEC &76 else decrement the Msn of lenqth
2S71 BNE &2e4A Not zero? branch for next 2 bytes
2S7~ DEC &2EDA Branch to release_tube

-type_6_transfer Transfers 256 byte block from 512 to host at
1OLtnacs~byte
2675 3Sfl &26S4 waste some time

qet_tube_byte
2678 LDA tube~data Read data frorfl tube register 3
267B STA (&74),Y store it at indexed host address
2S7n NOP nelay for 2 clock cycles
267E NOP Delay for 2 clock cycles
SelF NOP Delay for 2 clock cycles
2680 INY Increment the index
neal BNE &267a Branch get_tube_byte if < 256 bytes
2683 CPX 40 Have we finished?
2605 BNE &269a No - Repeat for next page
2687 DEC &76 Decrement MSS of length
2689 BEC ~26DA Branch to release_tube

- RspeRTSaae
Q6aB JSR &26cE Incr~nt 512 page address
266E LDA 46
2690 JMP &259r at- to set_transfer_type

set_type_0
2693 DEC '76 decrement MSB of length
2695 LDA &16 load LSB of length
2697 CMP Si Is this the last page?
2e99 BNE &266n No - branch to re~eRTSage
269B ASR &2sCe Incr~nt 512 page address
269E LDA *0 set type zero transfer
26A0 JMP &259F Jmp to set_transfer_type

-type_7 transfer
26A3 INA (&74 ,Y Read data fr~ hont netftory
26AS SXA tubensaeta write it to tube register 3
```

```
26A6 NOP Delay for 2 clock cycles
26A9 NOP Delar for 2 clock cycles
26AA NOP Delay for 2 clock cycles
26As INY Incrgrntnt index
26A0 BNE $26A3 branch type_7_transfer if C 256 bytea
26AE CPX to Is lenqth LSB - zero?
2630 ~~~ &26nE no - branch decrement_length_mb
26n2 DEC '76 Decrement ~ISfl of length
26n4 BEQ &2~DA If c~lnt, branch to release_tube

-set_type_7
2636 JSR &2~cE Increment 512 page address
26ns LDA &~7 set transter type 7
2633 JMP ~259F ~ to set_transfer_type

dacremant_length_lab
26~~ BEc &76 decrement MSB of length
26C0 LDA S1~ Load MSB of lenqth
26C2 CMP *1 ra this the last page?
26C6 JSR &26CE Increment 512 page address
aec; LDA 41 Set transfer type 1
26CR JMP &259F JL- to set_transfer_type

increment_512_offset    Increments the MSB of 512's offset
                        and the host adeirean by one page

26CR INC &75            Increments the host address high byte
QEDO INY #7             load index to parameter block
26D2 LDA (~7O~.Y        Load MSB of 512 address
26n4 CLC Clear carry
26n5 ~~~ Add 1 page to 512's segment offset
26n7 STA c'70}.Y Store at MSB of 512 offset address
2GD9 RTS Return

release tube
26DA IDA &a87 Load A for tube release
26Dc JSR tube_entry Call tube host code
26Dr LDA &72 Load original current no.' n-er
26E1 CMP xs it consistent with NOS?
26E3 BEQ &26eA Yes it~s OK - leave it
26E5 STA &r4 No - restore ~40S copy of ROM ni-er
26E7 STA roTRSel and restore ~FE3O in sHErn&
```

```
26~A PLA Recover contents of ACCOON
26~~ STA accoon and store at &FE34 in SHEIUL
2EEE ~~X £70 Restore x
26F0 ~~Y all aestote Y
26F2 PEA Restore A
26F3 RTS exit
26F4 JSR ~2Sr3 Occupies 24 clock cycles in all

~ End of OSWO~ &FA code ***tt******t****r,***
```

G: Hardware Projects

CLOCK add on

For under £2.00 some 512 users can extract a 20% speed improvement from their 512 board. However, be aware that there are potential problems and success is not guaranteed. The practicalities are simple and if you are competent with a soldering iron this modification can easily be tried and more importantly, reversed if it fails.

All that is required is to remove the 512 board from your machine, desolder the 20Mhz crystal (X1) and replace it with a 24Mhz crystal. These are obtainable from any decent electronics component supplier.

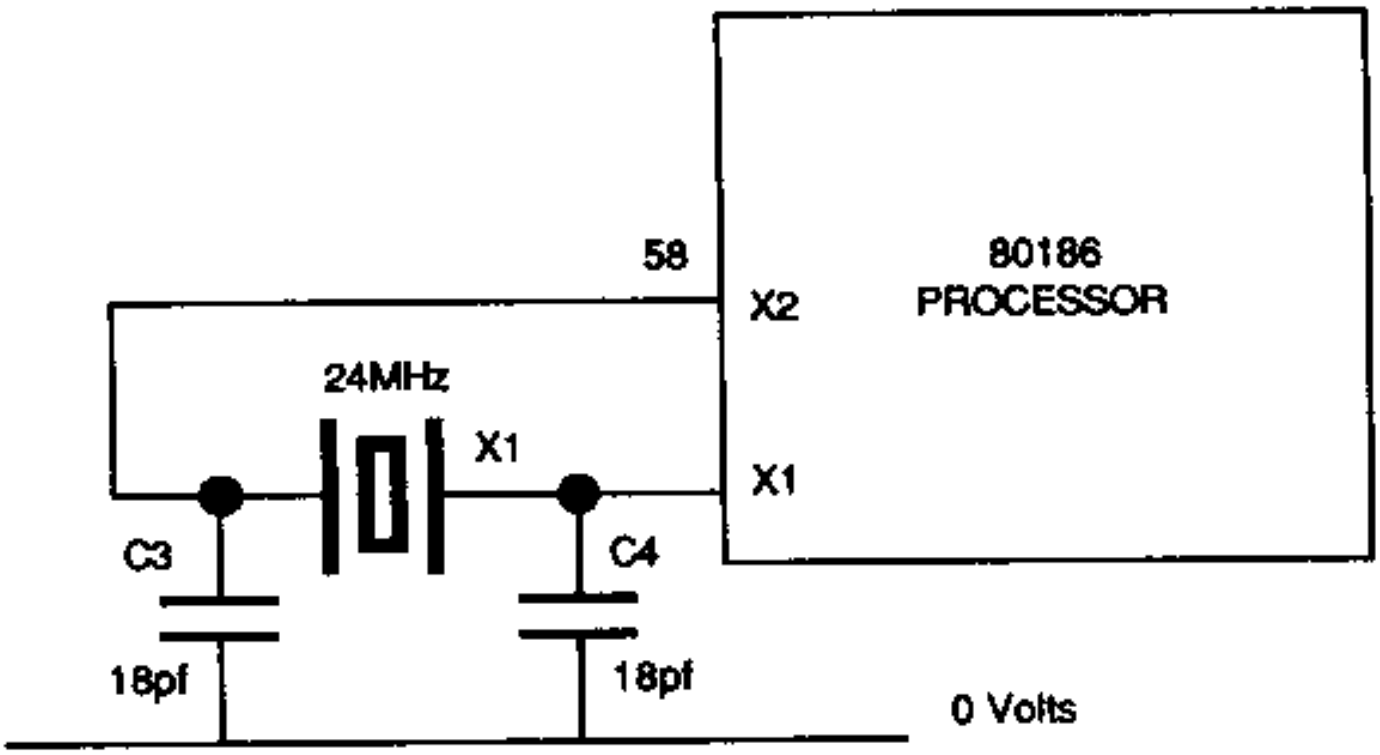


Figure 1 - Clock Circuit

The crystal is the silver metal-cased component located immediately at the side of the processor chip. It is soldered into the 512 board by two connections only.

By changing the crystal for a faster one you will be running the 80186 at 12MHz instead of 10Mhz. This works with some 80186 processors, but as it means running the processor 20% over its rated speed it is not reliable in all 512s. The only way to know is to change the crystal and test your 512 at 12Mhz. You will soon know if it doesn't like working out of specification, it won't boot correctly, or will crash out with random spontaneous errors within a few hours of being turned on. If this happens you might find that increased cooling (if possible) will relieve the problem. (Running the processor faster than its rated performance for a few hours is most unlikely to cause any damage, even if it ultimately does not succeed.)

In addition to the processor chip itself, another cause for the faster 512 failing may be the external co-processor box.

This can occur with either a Master or a BBC B/B+. The usual problem is timing, but it does work for some systems and Robin Burton has been running his 512 at 12 Mhz in an external co-pro attached to a model B for some time now without difficulty. If you do have problems, but really want to persist with this experiment the only other thing to try is replacing the 80186-processor chip itself.

Intel have now superseded the 80186 with the CMOS 80C186 which is a faster processor, but which is fully compatible and can be used as a direct replacement for the standard 80186. The 80C186 is available in 10 Mhz, 12 MHz and 16 Mhz versions.

You can replace the 80186 with a 12 MHz 80C186 (and this time you must change the crystal for a 24Mhz version) to get the 20% performance increase.

if you are really keen on modffications and hardware projects, you can try changing the 80186 for a 16Mhz 80C186 using a 32Mhz crystal, but be warned that you will certainly need to change both the two EPROMS and possibly the RAM too, since these would otherwise operate too slowly. This modification is clearly much more ambitious and costly than the change to 12Mhz, but it certainly would produce an extremely fast 512!

NB: It is *not* possible to change the 80186 for any vession of 80286 or 80386, because these processors are not pin compatible. It would therefore require extensive modifications to both the 512 board and to the software. You should also be aware that you would receive no benefit unless you used a fast 286 or 386, which cost very much more than a 186!

Master 512 RAM expansion board

This is a hardware project for those who would like more memory in their Master 512 and are handy with a soldering iron. The aim is to add a second bank of 512k of RAM to the 512 board, giving 1024k in a similar manner to the Solidisk PC+.

Publisher's note: You are advised not to attempt this project if you have no previous experience of circuit hoard construction and soldering, no matter how experienced you may be in other areas of using or programming the 512. Also note that incorrect assembly or fitting could feasibly damage the 512. This is not a beginner's project, nor are the costs trivial. As a rough guide, at the time of going to press raw materials costs are likely to be between £40 and £50 and RAM costs are currently increasing rapidly. If you do not feel completely confident about your abilities you are advised to refer to **Appendix H** for details of commercially produced 512 memory expansion boards. Dabs Press are unable to advise or assist in the building of this project.

Every effort has been made to ensure the accuracy of this design and successful prototypes have been built. However, the publisher, the author and the circuit designer accept no liability for any consequences arising from the use of this information, nor can they offer a consultancy service to anyone who encounters difficulty as a result of attempting the project. Use of the design is limited to the personal needs of the reader, this design may not be used for commercial purposes. Full acceptance of these terms and limitations is implicit in the use of this circuit design and its construction details.

Materials you will need:

```
Qty     item
16      256kx1 150ns DRAMs
```

```
1       74LS75
1       74LS139
1       74LS08
6       47 nf decoupling capacitors
1       10 nf electrolytic capacitor
3       2.2 Kohm resistors
2       Extended leg (15mm) 28 - sockets

16      l6 pin sockets (if you wish to socket your RAM)
1       Piece of vero board (wire wrap type) size 120 mm x 120 mm
1       Reel of 0.2 mm insulated wire
1       Reel of 0.5 mm insulated wire
or      Wirewrapping tools.
```

If you wish to socket the 74LS chips add two more 16 pin sockets and one 14-pin socket to the list.

The Master 512 uses 16 256k x 1 DRAMs. These are split into 2 banks of 256k bytes - the high byte and the low byte forming the 16-bit word used by the 80186. Each RAM bank holds one bit of each byte, hence 1 byte of each word. By adding a second bank of 512k wired to give the same format, you can have 1024k bytes of RAM.

The first task is to physically set out the RAM chips on the board. This is done using a similar layout to the RAM on the 512 board. Set out two banks of 8 chips, one as the high byte, the other as the low byte.

The next thing to consider is supplying the address and data lines to the RAM. On the 512 board the RAM, the processor and the ROMs have access to both the address and the data lines. To avoid too much soldering on the 512 board we can use the address and data lines from the EPROMs, especially as these are socketed. The address lines to the EPROMs are already split into high and low bytes, so this is highly convenient. The other signals that must be supplied are RAM read/write and refresh, the Row Address Strobe and Column Address Strobe. These are labelled RAS and CAS respectively on the diagrams.

The RAS and CAS are generated on the 512 board for its own RAM and we must patch into these signals for our extra RAM. We will need to supply address lines 18 and 19 (A18, A19) separately, as these are used by the EPROMs to decode the high bank.

The last thing the RAM requires is power. This is usually 5 volts on Vcc and ground (0 volts) on Vss. Power must be supplied to the addressing chips and can most conveniently be taken from the 512 board. As we are taking most of our signals from the EPROMs, we can conveniently take the power from there too.

Important: All chips and sockets have a notch at one end This is the 'top', also commonly called 'north' on circuit diagrams. Chips *must* be connected the correct way round. If they are not, not only will they certainly not work, but they may be damaged, and might well cause damage to other chips in the circuit when power is applied.

Pin 1 (for all chips) is always at the top left corner, next to the notch when the chip is viewed from above (i.e. the opposite side to where the legs point) with the notch at the top. Pin 2 is the second pin down on the left side and so on down to pin 8 for the RAM chips and the two 16 pin 74LS chips, 1 to 7 for the 74LS08 and pins 1 to 14 for the two EPROMs. Numbering then continues on the right side of the chip, but from the *bottom upwards* towards the notch this time. The pin assignments are also shown on the diagram.

Now to start laying out the board, beginning with the RAM (you can use sockets if you prefer not to solder chips directly). The first thing is to place the RAM chips (or their sockets) on the Veroboard. Lay them out as two banks of eight positioned about one hole apart and with pin 1 to the top. Solder each corner to hold them in. Your layout should be similar to this:

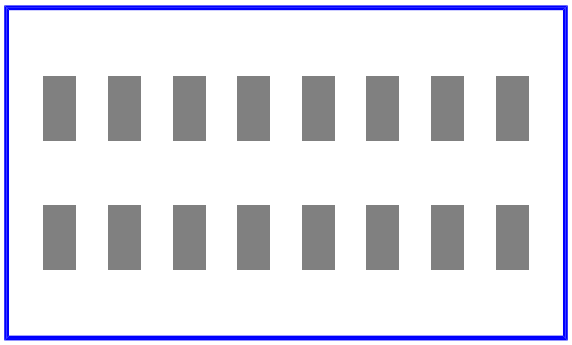
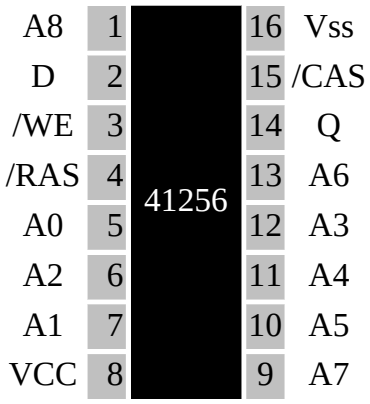


Figure 2 - RAM Bank Layout

The pin out of the RAMs is as follows. For each bank of RAM we need to join the address lines together along the banks, bringing the lines out of the left hand end of the banks.



A0 - A8	Address Inputs
/CAS	Column Address Strobe
D (Din)	Data In
Q (Dout)	Data Out
/RAS	Row Address Strobe
/WE	Write Enable Input (Active LOW)
Vcc	Power (5V DC)
Vss	Ground (0 volts)

Figure 3 - Pinout of DRAM

For each of the two banks, using the underside of the board, join A0 (pin 5) on chip 1 to A0 on chip 2 to A0 on chip 3 and so on right down to chip 8. Do the same with all the other address lines, ending with A8. Extend the address lines (A0 to A8) so that they come out of the left hand side of the two banks, but don't join the banks up yet.

Each bank should now look like this:

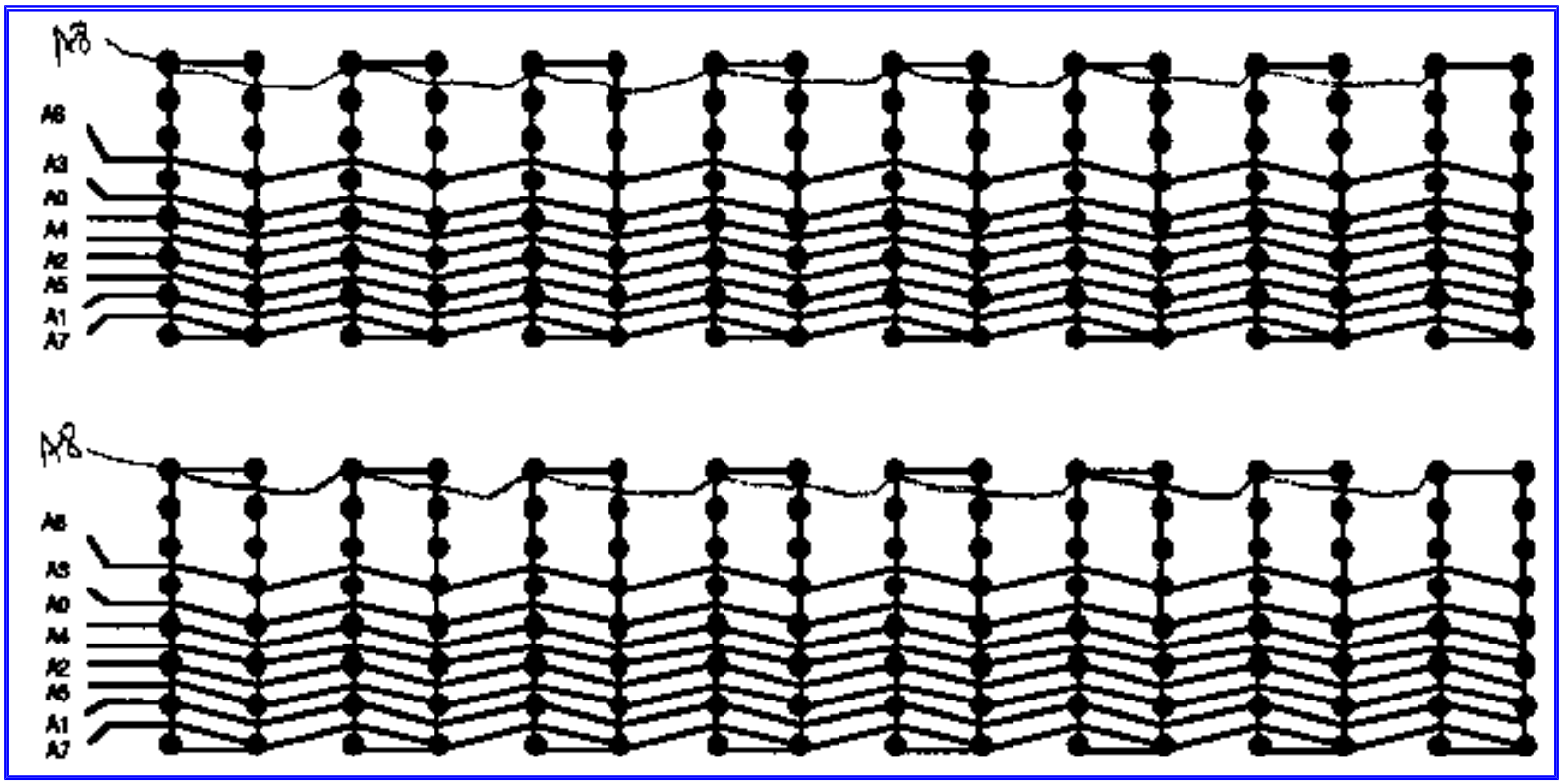


Figure 4 - RAM Address line Connections

The other lines, that is W, RAS and CAS should now be joined chip to chip in a sirnilar fashion. That is for each bank, join the appropriate pin of each chip to the sarne pin of the next chip, through all 8 chips. With these lines however, extend them out of the right hand side of the banks.

This just leaves power, Data in (D) and Data out (Q) unconnected.

On each chip join Data in to Data out. Now using the topside of the board, extend the data line for each RAM chip down between the chips, to the centre of the board.

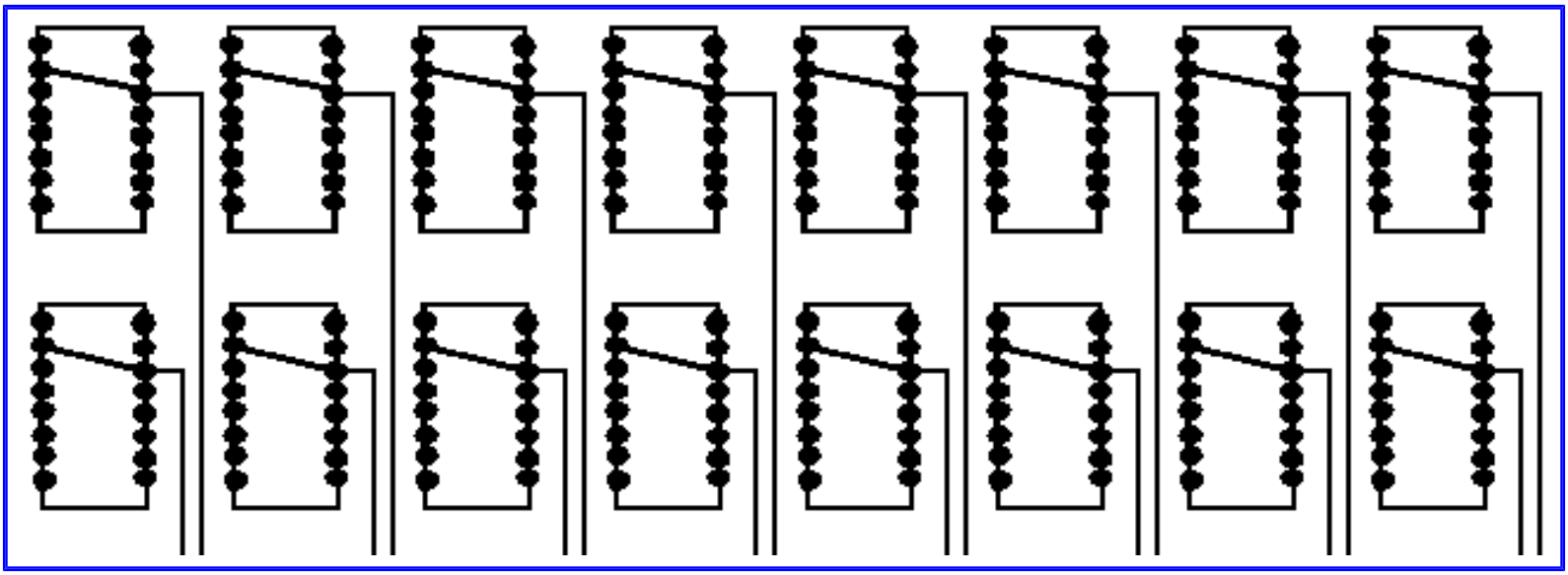
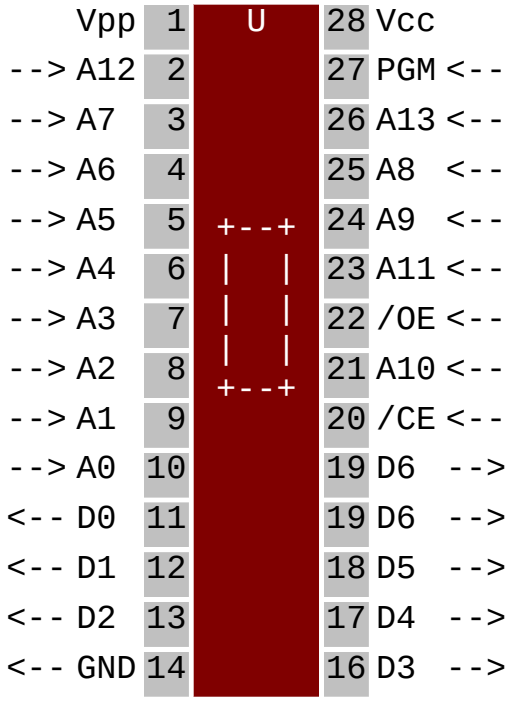


Figure 5 - Data Line Connections

Put the two EPROM holders onto the vero board with pin 1 to the right. Place them so that they plug directly into the EPROM sockets on the 512 board, leaving the RAM 5 mm clear of the processor. Next, solder the four corners of the sockets to the vero board. On the 512 board, IC31 is the low byte, and IC32 is the high byte. Using the top one of our two RAM banks as the high byte, join the data line of RAM 1 to D0 on the IC32 EPROM socket, then the data line of RAM 2 to D1 on the IC 32 EPROM socket, and so on until you have the data lines on each RAM joined to the relevant data line on the EPROM socket. The top RAMs should be joined to D0-D7 on IC 32 and the bottom RAMs should be joined to D0-D7 on IC 31.



Pin	Function
Vcc	5V
PGM*	Program
A0-13	Address lines
OE*	Output Enable, active low
CE*	Chip Enable, active low
A12	A12
Vpp	Programming voltage
D0-7	Data ouputs

Figure 6 - EPROM pinouts

Now on the underside of the board, do the same thing for the address lines. For the top bank join the address lines to the corresponding address line on IC 32. Then do the same for the bottom bank, joining the address lines to IC31.

Next we must add the control lines. The CAS is quite easy and is just a matter of connecting the CAS signal generated on IC 6, pin 8 on the 512 board to the CAS inputs of the RAMs. The RAS however must be patched in such a way that it does not interfere with the RAS signal on the 512 board.

As on the 512 board, we need separate strobes for the high and low bytes. These can be derived from the RAS signals provided by the 512 board. It is done using a 2 to 4 line decoder, in this case a 74LS139. This should be added onto the expansion board at the right hand end.

We also need two 2 input AND gates to combine the outputs of the 74LS139 and the 74LS113 on the main board. These are supplied by the 74LS08. The connections are described below and are also shown on the circuit diagram.

The best way to connect the RAS signals from pins 6 and 8 of IC 4 on the 512 board to our extension board is to desolder pins 2 and 4 of IC6 on the 512 and lift them from the board. Next solder the following:

1. A flylead from pin 2 of IC6 to pin 12 of the 74LS139.

2. A flyIead frorn pin 4 of IC6 to pin 4 of the 74LS139
3. A flylead from the vacated pad of pin 2 of IC6 to pin 14 of the 74LS139.
4. A flylead from the vacated pad of pin 4 of IC6 to pin 2 of the 74LS139.

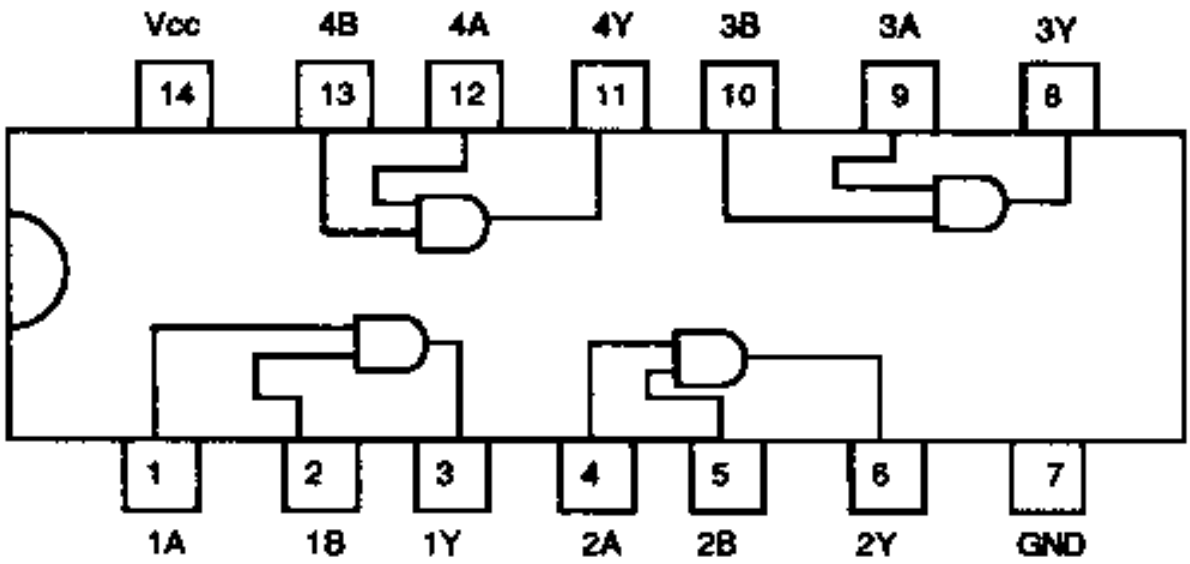
The next part is easier, as the 74LS08, IC 6 is next to the edge of the board and so is easier to work on. IC6 pins 2 and 4 are two of the inputs to the AND gates on the main board. On the 74LS139, join pin 13 to pin 3 so that the same signal can be used to supply both inputs. Also join together pins 1, 8 and 15, this grounds the enable pins for both decoders. Pin 8 will eventually also be joined to the earth rail.

We now need to fit the last two ICs to the expansion board. These are a 74LS08 positive AND gate with totem-pole output, and a 74LS75 4-bit bi-stable latch. These should be fitted near the 74LS139.

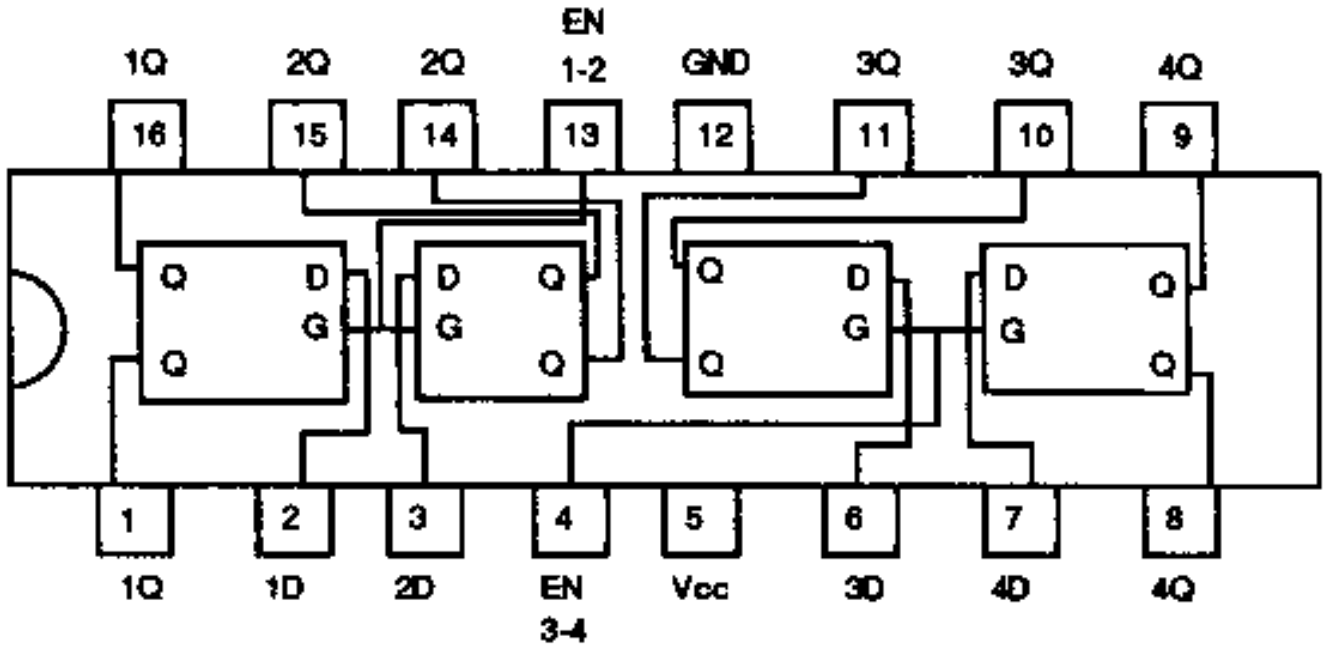
Join pin 3 of the 74LS139 to pin 15 of the 74LS75. This allows the Q output of the 74LS75 to drive the B inputs on the 74LS139. Now to add a few more fly leads. The first needs to go from IC3 pin 4 on the 512 board to pin 4 on the 74LS75, then also join pin 4 to pin 13 on the 74LS75. This supplies the enable signals to the 74LS75.

Now to fit the fly leads from the processor. You may alternatively remove the processor from its socket and solder onto the socket, or you can solder directy to the processor. Both methods work, but for safety you are strongly advised to carefully remove the processor chip and solder to the socket. Small cut outs are provided at opposite corners of the 80186's socket to allow insertion of fine screwdrivers, probes etc. to allow removal of the chip. Insert your chosen tools and lever the chip out evenly. Take great care, the 80186 processor is expensive to replace if damaged.

74LS08



74LS75



74LS139

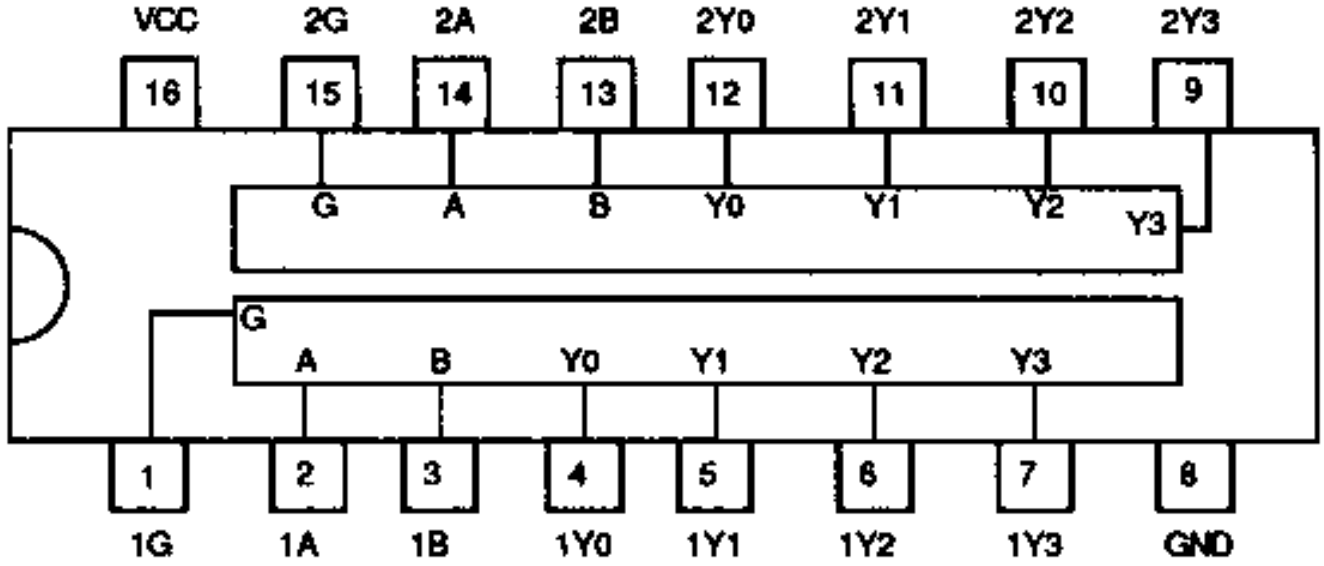


Figure 7 - Pinouts for 74LS08, 74LS75, 74LS139

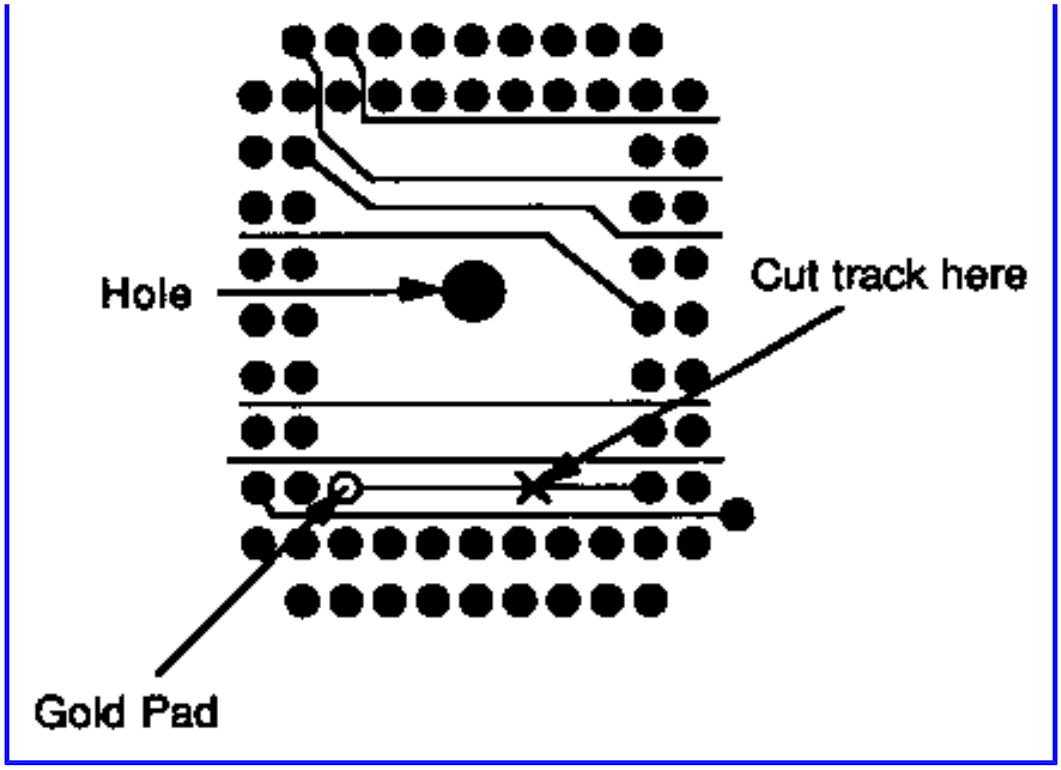


Figure 8 - Cutting the Master 512 tracks

The first signal required is the "not lower chip select" from pin 33 of the processor. We also need to cut the track between pin 33 of the processor and the through-plating, effectively disconnecting this pin from the main board. Turn the 512 board over, and place it so that the main board connectors are to the left and right, and the sticker showing the serial number is near the top of the board (away from you). Locate the processor, a square of connections, two pins deep all around. You will see a gold pad at the bottom left, inside the processor connection. A track leads from this pad, horizontally across to a processor pin. This should be cut through with a sharp knife *taking great care not to damage any other track in the area.* Now connect a fly-lead from pin 16 of the 74LS75 to the through plating on the main board so that we are supplying not-LCS instead of the processor. The other two signals required are A18 and A19. These require a fly-lead from pin 2 of the 74LS75 to pin 65 of the processor and a fly-lead from pin 3 of the 74LS75 to pin 66 of the processor.

Now for the 74LS08. On this we are only going to use two of the four AND gates; pins 1, 2 and 3 are the first and pins 4, 5 and 6 are the second. The first two inputs come from the 74LS139 (1Y2 and 2Y2).

Wire pin 10 of the 74LS139 to pin 2 of the 74LS08, then pin 6 of the 74LS139 to pin 5 of the 74LS08 to obtain these signals.

The signals that these are ANDed with come from IC6 pin 5 on the 512 board. They are obtained by wiring a fly lead from pin 5 of IC6 on the 512 board to pin 1 of the 74LS08, then joining pin 1 of the 74LS08 to pin 4.

The outputs of the 74LS08 are the RAS signals for the RAM. These are on pins 3 and 6, but as the signal is a NOT RAS signal the lines need to be pulled high to stop them floating. This is done by pulling these pins up to 5 volts through two 2.2K ohm resistors.

The resistors should be fitted near pins 3 and 6 of the 74LS08 with room between the resistor and the pin to take off another lead. The pin should be joined to one end of the resistor and the other end of the resistor should be joined to

the +5V rail. The RAS line that you wired in for the low byte on the RAM should then lie joined between pin 3 and the resistor, and the RAS for the high byte should be joined between pin 6 and its resistor. This gives us the row address strobe for both banks of RAM.

For the CAS we need to take this from the CAS signal on the 512 board. This is done sirnply by wiring both the high byte and low byte CAS lines on the RAM board together and fixing them to one point, then taking a fly lead from this point to resistor R9 on the 512 board.

Attach the fly lead to the end of the resistor nearest to the edge of the 512 board (to the left of IC6). Again this is an active low signal, so needs to be held high. This is done by wiring a 2.2K ohm resistor from the point where the fly lead and CAS lines join to the + 5V rail.

Next we will join in the NOT write signal by yet another fly lead. Again we are going to join the two W signals from the RAM together at a point (or pad) on the RAM board.

From this point we can then take a single fly lead to the 512 board and pick up the W signal from the resistor R11 on the main 512 board. Take the signal from the side opposite IC3.

Last, but equally important, is the power for the RAM chips. This is best provided by adding two power rails that run around the board, one from the +5 volt pin of one EPROM to the +5 volt rail of the other EPROM, and a rail from the 0 volt line of one EPROM to the 0 volt rail of the other EPROM.

Use thicker wire for the power supply rail and try to go as near to the power pins of all the ICs as possible. Join all of the Vcc pins to the +5 volt rail and all of the Vss pins to the 0 volt rail (this pin is sometimes labelled Vdd). Next, to each of the +5 volt connections to the RAM, solder on a 47 nf decoupling capacitor to the +5 volt rail, and solder the other end to the 0 volt rail. Repeat this for all 16 RAM chips.

Finally about halfway around the power rails, solder a 10.0 µF electrolytic capacitor across the two power rails, joining them together.

Ensure that you get the capacitor the right way around otherwise it may well go bang. It should have a +' mark on it. This is the end that should be joined to the +5 volt rail.

Now plug in The EPROMs, the RAM and the ICs (if you used sockets). Plug the RAM board into the 512 board, check all the connections, and then test the unit.

If you have built it correctly, booting up the system using DOS+ 2.1 should give you 768k of free memory.

Building Your Own Hard Disc

It is not the intention of this book to teach electronics, in spite of the 512K memory expansion elsewhere. That was prompted by termination of supplies of the Solidisk PC+. However, there is one other area worth outlining, since it will surely be of interest to many 512 users. We have included it because being aware that something is possible is often half the battle.

A desirable addition to any microcomputer is a Winchester, often called a hard disc. The Master 512, with its much

bigger memory, potentially much larger data files and usually much larger application programs makes this an even more attractive proposition. Floppy discs of 640K may be adequate for the BBC micro, even if sometimes inconvenient, but for serious users of the 512 disc capacity, even with the 800K format available in DOS plus, can be a serious restriction on the use of large applications such as a database or a spreadsheet.

512 users, more than most, are also aware of the fact that hard discs for PC clones are very much less expensive than similar capacity devices for the BBC micro, sometimes as little as half as much. This disparity in cost between ostensibly the same type of device intended to serve the same purpose has led some users to seek an alternative to purchasing a ready made BBC micro hard disc drive.

The differences in cost between a complete hard disc system and the individual components required to make one has induced some enthusiastic users to build their own units from individual boards, though not usually from individual IC components, although this is also possible!

This approach no longer offers such savings as it used to, primarily because the disc cartridge, which used to be the most expensive component, has vastly reduced in price over recent years. This is of course largely due to the increasing popularity of this type of add-on which, perversely for the 512 owner, has occurred mainly because of the IBM PC and its various lookalikes. For example, hard discs of 20 to 30 megabytes which now can be purchased 'ready to go' for about four hundred pounds for the BBC micro and two hundred pounds or so for a PC, could have cost well over £1,000 only two or three years ago.

However, the impetus to save money is still driven by the price of PC hard discs. These are available at under £200 for a 20Mb drive for a PC (and very little more for 30Mb) and understandably BBC Micro users feel hard done by that they must pay double this or more.

The criticism is not entirely fair, as there is one particularly expensive component in the BBC system that is not needed by a PC. The unfortunate fact is that this component remains expensive, because outside of the BBC micro market its use is fairly limited. As you'll see in the following list, this particular component now costs more than the bare disc and drive. Accepting that prices have fallen, even for BBC micro hard disc systems, you can still probably save about £100 by putting the system together yourself.

This is not a difficult job. It certainly requires no electronic skills, and probably (unless a wire falls off) requires little or no soldering either. There is also a certain satisfaction in knowing that you did the job yourself, but as with most 'good ideas' there are cons as well as pros. and this one is no exception. In this case you must remember that there aren't any guarantees on a home-built system. Should it fail, either because of bad workmanship or because you have used cheap or faulty components, you're probably on your own.

The equipment required to run a hard disc on a BBC micro is:

A SCSI interface card £175-225
A SCSI interface card £175-225
A host adaptor card £35-40
A power supply £10-20
A case (optional) £11-20
Cables £10-15

These prices are naturally approximate, but were typical in the summer of 1989. It is assumed that you already have

an Acorn compatible ADFS system. If not then this cost must be added.

Many people start their search for the other components after acquiring for little or no cost the hard disc unit itself. Real bargain hunters can probably substantially improve on the prices for the case, power supply and so on, but the two cards required, the host adaptor and the SCSI are going to remain costly items. In fact one of these, the SCSI, virtually on its own accounts for the difference in price between PC and BBC dives. PCs don't need an SCSI interface, BBCs do. You may consider avoiding the SCSI and designing your own direct hard disc connection to the 1 MHz connector, but remember that if you do this Acorn's ADFS will not support your hard disc system and you'll also have to write the driver software.

The SCSI Interface

The SCSI (Small Computer Systems rntefface) is a standard interface used to transfer data at high speed between micros and various peripheral devices. It is most commonly used (outside Acorn systems) for interfacing laser printers, plotters and scanners, as well as for one-off specialist applications. The 'intelligent' control required at the computer end is limited to issuing high-level commands, usually followed by parameters. For example the ADFS sector read and write commands use codes &08 and &0A, which are in fact the SCSI command codes for reading and writing data.

All SCSI hard disc controller cards are functionally the same, and all of them should have the same connectors on board. The SCSI standard was formerly known as SASI. This was originally developed by Shugart, the pioneer hard disc manufacturer, initially for their own products. Subsequently the standard was adopted as an ANSI (American National Standards Institute) standard, and reissued with slight modification as SCSI. SASI was the acronym for 'Shugart Associates Systems Interface' but this name was unacceptable to ANSI as it included and effectively advertised a trade name, hence the change.

Most SCSI cards will work correctly with the BBC Master. The one manufactured by Adaptec can be specifically recommend as Dabs have had one of these device running on one of their own BBC Masters for over two years without problems. You need not assume that other cards are not as good, but firm recomendations can only be given from our own experiences

The Host Adaptor

The 'host adaptor' card is a simple interface which converts Acorn's 1 MHz bus signals to or from the correct format for the SCSI interface. It has connectors for both the SCSI bus and the BBC 1 MHz bus. The card is specifically for BBC micros, and therefore normally must be obtained direct from a BBC rnicro dealer. The cards are made by Acorn and a few other suppliers.

Other Components

The remaining components can be obtained either from your local Acorn dealer, a local electronic component dealer or by mail order from one of the many suppliers who advertise from time to time in the BBC micro press. PC magazines are also a source for such dealers, as are some of the electronics hobbyists magazines. Indeed some dealers specialise in hard disc kits or components, and the better ones can usually supply reasonable instructions or diagrams about how to put the components together, as well as supplying sundry items like screws for fixing the finished drive into a case.

If you are able to find a single dealer who can supply most or all of the requisite components you will to some extent rninimise the Iikelihood of incompatibilty, and will certainly be better placed to request assistance if you have trouble and suspect faulty components

Raving put your hard disc together, the next step is to connect it and test it.

Formatting

There are two stages to preparing a Winchester for use in DOS Plus, the first step being that the Winchester must be formatted in ADFS. 'Ready to go' Winchesters for Acorn systems are often supplied ready formatted, but depending on where you obtained your drive, you may not be so lucky.

The format program needed for Winchesters is not the one used for floppy drives, but it is specific to Acorn machines and so is not a standard DOS program either. The formatting utility should be supplied as part of the the package when you buy a ready made drive for the BBC micro, which of course is of no help whatever if you have built your own, unless components were bought from a dealer specialising in Acorn systems.

The simplest solution is to find someone with a complete drive and to persuade them by fair means or foul to format yours for you, or to allow you to do it on their micro. This is necessary because it would be an infringement of copyright if you were to acquire a free copy of the program yourself from another user. However, some suppliers may be prepared to supply a format program directly.

The Acorn version of the Winchester formatting program is called Superform, which is a BASIC program contained in a directory called FORMAT on the Winchester itself, since Acorn Winchesters are supplied ready formatted.

It is not the intention in this section to explain how Superform works, and in any case you may be using a different program, but there are naturally common elements to formatting a Winchester whatever sofiware you use.

Unless software which is used is written for only one size of drive the program will require that you supply parameters to indicate the number of tracks and sectors. You should be aware that in some programs default parameters are provided. For example the default Superform parameters will format a 10Mb drive, even if a 20Mb or 30Mb devke is being used, in other words there is no automatic sensing of the drive's capacity. To supply the correct values you may need the data sheet for your particular drive. This should accompany the drive when you purchase it.

There are major differences between the processes of formatting Winchesters and floppy discs. One you wlll certainly become aware of is that formatting a Winchester takes longer, and very much longer for the larger sizes like 40 or 60Mb. Also, depending on the format program used, you may see displayed messages indicating the number of bad sectors found on the disc. Although alarming at first sight, this is quite normal because, given the large number of sectors and the high data density of the disc, a limited number of errors is almost unavoidable.

To cater for the loss of storage which would otherwise result from bad sectors Winchesters have a reserved area. Bad sectors or tracks are mapped onto this reserved area during formattting in such a way fhat references to the bad adresses are re-directed to the reserved area. After formatting is complete your Winchester is ready for use as a large capacity ADFS disc.

To use the Winchester in DOS+ you must then carry out stage two. This is done by booting DOS normally, then

using HDISK, included on DOS+ issue disc 1, to create a DOS partition. This is an ADFS file which is recognised by DOS+ as drive C: and it is referred to as such when you access it. if you've previously been using the Winchester in ADFS, as with all disc operations, take sensible precautions. Before you start ensure that you have adequate, up to date backups of all your files.

The size of the partition to be set up is selected during the HDISK initialisation process. This creates a locked ADFS file of the size requested. You can also make the hard disc bootable by copying the system files to it during this process, although this is optional.

The partition size will depend on the size of your Winchester and the amount of space you want in DOS (and ADUS), but ensure that there is enough free space (in ADFS) to allow for the DOS partition, or the initialisation will fail. The remainder of the disc, *after* the partition is set up, is available for use in ADFS when operating in native BBC mode.

The DOS Partition is allocated as an ADFS file, and it's a good idea to force the DOS partition to the 'end' of the disc. This is done.by first `*COMPACT`ing any existing ADFS files then `*CREATE`ing a suitably sized dummy file, again in ADFS, to reserve the rest of the space required in ADFS as a single, contiguous physical area.

After `DRIVE_C` is set up delete the dummy file and ADFS will then be restricted to the first part of the drive, while DOS will use the second part. Adopting this technique results in better performance, as ADFS files won't have to 'jump over' the DOS partition when they extend, and so unnecessary head movement will be avoided. On a more unpleasant note, it also means less trouble if you are ever unfortunate enough to need to recover data from the Winchester the hard way by editing sectors, because at least the two file types will be physically separated. *(I think the last sentence should read either 'more pleasant' or 'more trouble')*

You can of course still boot DOS from floppies even if you use a Winchester, which you may need to do for example to run a different version of DOS for compatibility with particular packages. However, when you do boot from the Winchester there are two obvious differences. The first is that it's much quicker and the second is that the initial screen is in white, not green as it is with floppies, as a reminder of which action you have chosen.

If you later decide that your DOS partition size needs changing you can alter it by repeating the installation process, but if you do this take note that you must first secure all your DOS files. The vital point is that re-allocating the partition loses the previous contents almost as surely as formatting. A new, empty root directory is always created, because the partition will start on a different physical track on the disc. The physical data is not actually deleted, but all the files are inaccessible.

The final stage in setting up a DOS partition is to to create the directories and sub-directories you will need, copying the files and programs to them from floppy as necessary. After creation of your DOS parition you should proceed to copy your DOS utilities and applications to it, together with any other files you'll need. The DOS copy command is used m precisely the same way as it is for floppies, except that the target directory is on drive C:.

It's normal to create a directory called `DOS`, or `UTILS` in which all your DOS transients and utilities are stored, keeping them separate from your other files. You should then set 'path' to point to this directory by the command:

```
path C:\dos
```

This should also be included in the `AUTOEXEC.BAT` file in the root directory of drive C: so as to ensure that

subsequently the path is set up automatically every time the system is booted from the hard disc.

Remember too that the floating drives, N:, O: and P: are very much more useful with a hard disc than they are with floppies. When assigned to appropriate directories on a hard disc applications can 'see' the hard disc as several different volumes.

H: Third Party Products

The following information concerning 512 specific products or services is provided in alphabetical order. Unless stated to the contrary all products or services listed are believed or known to be of good quality and reliable. However, you must judge for themselves the suitability of any given product for specific tasks or applications.

Beebug Ltd.

Beebug deserve mention not only as a supplier of hardware and firmware products for the BBC micro, but because they are now the only publisher of a magazine which is exclusively dedicated to the 8 bit BBC micro family, including its various co-processors. while other magazines have effectively transferred their allegiance to the Archimedes, Beebug instead choose to produce a cornpletely separate Archimedes magazine while continuing to publish the BBC micro magazine in its original form rnaintaining its original objectives.

A regular feature of Beebug magazine is 512 Forurn written by Robin Burton. This column, which is exclusively devoted to the 512 and related matters has been a feature of the magazine every month since mid 1988. From time to time additional contributions of 512 material are also included, such as reviews of 512 books and products. Sorne 512 software has also been included on the monthly readers disc.

Since other Acorn magazines have now substantially or entirely transferred their interest to the Archimedes, including only a page or two of relevance to the 8-bit BBC micro and extremely rarely even mentioning the 512, you might do well to consider taking out membership of Beebug. The magazine is published in ten issues per year and is available only by subscription, delivered direct to your home or office. A sample copy can be obtained on request and back issues are available for most items you may have missed.

Beebug Ltd., 117, Hatfield Road, St. Albans, Herts. ALt 4J8. Telephone 072740303

Dabs Press

Dabs Press, publishers of this book, publish three other Master 512- specific items, which are two volumes of Master 512-compatible shareware, and the companion book to this one, *Master 512 User Guide*, by Chris Snee. Full details of these are given in Appendix L.

Essential Software

Essential Software is a recent addition to the extremely limited list of suppliers of 512 products. Started

in late 1989, Essential Software is run by Robin Burton and Mike Ginns (better known until now for Archimedes books and software). Essential Software's stated objecfive is to provide novel and useful 512 software at reasonable prices. The list of currently-available packages is extensive and is still growing.

All of the current software packages consist of several programs, including support utilities to load, configure, enable, disable or query the various modules from DOS, even for programs that actually run in the BBC host. For programs actioned by 'hot-keys' the suppon utilities allow user defined hot-key selection, Saving and re-loading of key definitions, module suspension or deletion and the interrogation of module status and current hot-key definitions.

Many of the packages, like the command line mouse driver or the screen print, use BBC RAM and thus do not reduce 512 memory even though they're permanently resident and instantly available. All products are compatible with all versions of host and all versions of 512 DOS Plus. Where configuration may be required (e.g., to suit differing sideways RAM boards) the necessary programs are also included as required. All programs are loaded and controlled entirely from DOS with the exception of GoBBC/512 (See below).

In an effort to contain costs no printed documentation is supplied with these programs. Instead, all discs contain copious notes instructions, hints, tips and examples in text files which are interactively driven from a menu by simple numeric selection. The current product range is outlined below.

Ramdisc utilities

This package includes a highly memory-efficient ramdisc which uses 89K less memory than the standard MEMDISK supplied with DOS+ 2.1+ Also, when a rarndisc of less than 64K is required the program reconfigures DOS's ramdrive to use 1K blocks and the directory size is reduced to 32 entries. This halves the average 'per file' losses and directory wastage compared with MEMDISK. The ramdisc may be user-configured to any size from 10K to 64K in 1K steps, and from 64K to the total memory size less 64K in 2K steps. It also can be configured as any available drive in (i.e. except L:) on initial load.

Two other programs accompany the rarndisc. The first of these, AMNESIA, deletes the ramdisc without the need to re-boot the system provided that the ramdisc contains no files. If files do exist in the rarndisc the program reports an error and leaves the ramdisc intact, preventing accidental loss of files. When deleted the rarndisc may be loaded again as a new drive, or with a new size, or both as many times as required.

The second additional prograrn, DISCID, permits *any* drive including the ramdisc, both floppies and the winchester to be changed to any other drive in. This program can also report on the current drive identities and has the ability to suspend any drive so that even DOS Plus can't access it. (It can, of course also re-instate the drive.) Rarndisc Utilities is supplied on disc, is available now and costs £14.95 complete.

Suprstar

SUPRSTAR is a replacement for the standard STAR utility. [t maintains a separate virtual screen for * operations and can be called at any time, on hot-keys. when you enter the routine you see a true BBC mode 7 screen which still shows any * commands you'd issued previously and DFS is automatically selected too. On exit, achieved by pressing ESCAPE, the contents of the DOS screen from which you called SUPRSTAR are preserved as though you had never left it.

Within the * screen the machine behaves almost like a normal BBC micro, providing native facilities like standard BBC cursor editing, ESCAPE processing and normal MOS error handling.

The two greatest advantages of SUPRSTAR are that, first, commands can't destroy your DOS screen (even four colour graphics). Conversely, subsequent DOS operations don't overwrite previous * commands either, both being major annoyances with the standard program in spite of the fact that it can be called only from the command line. Secondly, as SUPRSTAR operates on hot-keys (default SHIFT-CTRL-*), SUPRSTAR can be called at any time from any package, even in the middle of loading a file if you like.

SUPRSTAR is available now on disc and costs £14.95.

GOBBC/512

GOBBC is a complementary program to SUPRSTAR, which you *must* have to use GOBBC. Please note this facility is impossible to implement within the extreme limitations of the standard STAR utility.

Having entered SUPRSTAR (from within a live DOS application if you wish) simply type GOBBC and you can immediately 'drop into' a completely normal BBC environment. If you wish you can directly enter any resident language, for example you might be editing in your DOS wordprocessor when you need to edit a BBC document in View prior to importing it. Just enter SUPRSTAR and type GOBBC *WORD and you're in View. Equally you can enter GOBBC on its own which will select BBC BASIC as the default language. After this you might run a BASIC program in native mode.

On conclusion of BBC operations simply enter * 512 from any command line or menu to return to precisely what you were doing in DOS, exactly where you left it and as if you hadn't. Just as for SUPRSTAR, the DOS display is maintained in its existing state, nothing is lost. if you were using your DOS wordprocessor, you're still using it when you return. If you entered SUPRSTAR while a file was loading, it will automatically continue to load when you return to DOS.

GOBBC/512 is supplied on disc in BBC DFS format and is available now. The price is £14.95 when purchased with SUPRSTAR (i.e. £29.90 the pair) or £19.95 when purchased separately.

Printscreen Utilities

Printscreen Utilities is a package that fills a glaring gap in the 512's capabilities, one that is probably a worse omission than the absence of a mouse driver. All real PCs have a PRTSCRN key which allows a print of the screen display to be produced on demand. This is now possible in the 512, but in a much more sophisticated form than *in any PC.* Two screen print programs are included in the package, GRDUMP and PRTSCRN, both provide two functions, all of which actioned by hot keys and so can be used at any time from any package.

GRDUMP will print anything on your screen in any mode, though it is expected to be used mainly in pure graphics modes to give shaded printer dumps of two or four colour screen displays. The screen print is produced sideways on the printer so as to permit a large size irnage (approx. 11" x 5") with no detail lost. Obvious applications for GRDUMP are immediate hard copy recording of images in art or graphics design applications.

INVERT Is a second graphics option, permitting the screen colours to be inverted as they are sent to the printer so that a display which would ordinarily produce large quantities of a black or a very dark background (like GEM or Wordperfect) appears on paper as a negative of the normal image, with a white or pale background, saving both time and printer ribbons.

PRTSCRN is designed for use or in 40 or 80 column text screens to give a 'fullI speed' print of the screen when text or mainly text is displayed. if graphics characters are encountered in text displays, for example IBM characters, the program automatically drops into graphics mode to print the non-alphanumeric characters, ensuring that 'non-IBM' printers can handle all displays. This prograrn is invaluable for instant recording of transient screen information such as disc catalogues or hard copy recording of the help display from command line programs

The second text print facility in PRTSCRN is FAXPRINT. This is similar to PRTSCRN except that the printer's line spacing is adjusted to close up the gaps between lines which normally spoil the appearance of mixed prints of text and graphics. It is therefore just a press of the hot-keys to produce a rapid, precise hard-copy facsimile (hence the name) of mixed text and graphics displays, with box characters in particular appearing properly 'joined up'. FAXPRINT is ideal in applications for rapid recording of pull down menus which include text, virtually standard in most applications nowadays, or for printing menus or displays containing IBM box Characters. FAXPRINT is also very useful for producing a compact (approx 8" x 3.5") and rapid image of full-screen graphics when the need for large scale and fine detail is less important.

Both programs, GRDUMP and PRTSCRN, are configurable for both 9 and 24 pin printers, either on loading or after installation. They are supplied on disc, cost £14.95 complete and are available now.

Command Line Mouse

Essential Software's mouse driver, unlike a standard PC mouse driver works on the command line (or in fact anywhere that the keyboard can be used) hence its name. It therefore provides cursor control by mouse movement in programs which normally do not or cannot work with a mouse, as well as many that do.

Command Line Mouse is not an MS mouse driver, since it takes no 512 memory and doesn't require packages to be configured to use a mouse, or even to be able to be so configured. ES's approach has the advantage that any program, including strictly keyboard only types, can be used with a mouse. Also the SHIFT/CTRL keys work with the mouse in a similar manner to the way they usually work with cursor keys, giving instant and super-fast scrolling. The two mouse buttons can be user-assigned to any key function (RETURN and ESCAPE by default) and the sensitivity of mouse movement in both the X and Y planes can be independently adjusted. Your chosen settings can also be saved to disc by the SAVEKEYS utilitity. The Command Line Mouse driver is supplied on disc, costs £12.95 and is available now.

Screen Save

SCRNSAVE permits any DOS Plus text or graphics screen display to be saved directly from DOS, within an application or from the command line, to either DFS or ADFS formatted discs. Defaults are provided, but the filing system, drive and filename are all user definable. A sequence identifier antomatically increments after each save to permit multiple related images to be saved without the need to change filenames.

Screen saving is triggered by hot-keys, so that any diagram or text in any package can be saved without disturbing the screen image and without access to the command line. The filename, the drive or the filing system can be changed independently as often as required.

SCRNSAVE is supplied on disc at £9.95 and is available now.

Key Translation

TRNSLATE is a key translator which allows extra keys to be defined on the Master, B or B+, or any existing key function(s) to be assigned to other nominated key(s). Definitions can include the missing numeric pad keys for model B/B+ users (or Master users if they don't like the current locations) as well as other normally unavailable IBM keys.

The keys used and the actions to which they translate are entirely user definable to the extent that you can re-map the entire keyboard if you wish. If you don't like the current key assignments in any package you can now easily change them at will. TRNSLATE costs £14.95 and is available now.

Popup Notepad

512 memopad is a 'pop-up' notepad. Access to the pad is by hot-keys, and one bank of BBC sideways RAM is required for storage, as the pad uses no 512 rnernory. whatever you're doing, press the hot-keys, make a note or two for use later, then return to the main application.

Utilities are included allow the contents of the pad to be saved to (DOS) disc and subsequently reloaded. Built in commands allow the pad's contents to be cleared or printed, both requiring confirmation before execution. A simple wordprocessor like environment is provided for the entry of data in BBC screen mode 7. Facilities include character or line insertion and deletion, plus line or page scrolling in any direction. Approximately 12K bytes of storage is provided for text (approx. 4 to 6 A4 pages).

The 512 mernopad is supplied on disc at £9.35. Available now.

Miscellaneous Utilities

This is a disc containing various utilifles, including:-

SELECT - a batch file menu driver which provides an interactive link between the user and batch files, giving on-line selection and operation of the options presented in batch file menus. A cut-down version of this is used on all Essential Software discs to drive om own 'readme' batch files.

COLORDEF - a screen colour changer which replaces COLOUR, allowing the screen colours to be changed on hot-keys instantly and at any time. Ideal if you use GEM, WordPerfect or any package with a glaring background. With COLORDEF change colours whenever and as often as you like.

SUSPEND - stops the 512 in its tradcs. Both the suspend and resume actions are on hot-keys, so you can stop the machine while you answer the phone, get a coffee etc, and be sure that no-one has destroyed hours of work or your files when you return (tintess you tell them your hot-keys!).

LOCK - Similar in purpose to SUSPEND, but this is a command line program and is absolutely secure. You supply LOCK with a password when requested, anything you can enter through the keyboard, including control keys. The password therefore can be different every time you use it. On entry the first password is deleted from screen and after this, until the correct, and only the correct password is re entered the machine won't react to any further keyboard instructions, including CTRL-C. When the correct password is re-submitted the program reports if illegal entry has been attempted and if so, how many attempts were made.

SOUND - a program which allows full access to the BBC micro's sound facilities from DOS - just enter the parameters exactly as you would in the BBC micro.

ENVELOPE - a complimentary program to SOUND, allowing sound envelopes to be defined from DOS. Both SOUND and ENVELOPE operate exactly as they do in BBc BASIC in native BBC rnode.

(Enhance your batch files with these two by adding varying sound effects to menu choices made through SELECT) This disc contains 26 files including documentation and support utilities, so you'll have to buy it to find out about everything on it. The miscellaneous disc costs £11.95 complete and is available now.

Any of the last five programs, COLORDEF, SUSPEND, SOUND, ENVELOPE or WCK may be added to any other order at a cost of £2.50 for each program. SELEOr may be added to any other order for £5.95.

History

HISTORY is a DOS command archiver which gives options of both manual and automatic command recall. Manual recall is perforrned by cursor keys (or the command line mouse), allowing biAirectional scmlling through the command history. Automatic recafl is based on partial command matching, when entry of part of a command instructs the program to find the nearest previous command matching the information giver'. A recalled command can then be edited and a ~h& search initiated, orit can be used as the Stan point for hirtha manual history examination. Recalled commands can be edited and executed as new commands, or simply repeated without amendment. By scrolling backwards or forwards through the history this program also provides an audit frail of operations performed, as well as the ability to rapidly repeat any sequence of commands with the minimum of keyboard re~entry. Used in conjunction with PRTSCRN, HISTORY can be naed to provide a printed log of operations performed-

HISTORY costs £12.95 and is available now.

Additional Products

Other Essential Software products are currently planned or in development. Some are complete or nearly so, but were not packaged at the time of going to press. The list includes the following.

512 Memory Expansion

A 512K byte memory expansion board for the 512 is available. The board uses the latest chips to allow an extremely compact, low-power board with high inherent reliability. The price of upgrades supplied from the initial production batch, fitted and complete with modified EPROMS is £;99.00,

rt is intended that this price will be maintained for as long as possible. However, owing to the currently unstable prices for RAM (caused by the EEC's impending 'antlAumping' duty on chip imports) and the uncertainty of the total number of boards to be manufactured, long term prices cannot be guaranteed.

A further batch of boards may be manufactured if overall demand justifies it, hut if a subsequent batch is of a smaller number the price may unavoidably inaease to reflect this, as the cost of chips and of boards varies inversely with the number purchased.

If you have any doubts about your ability to construct the project in the previous chapter, you might like to contact Essential first about their ready-built solution.

Other Software

A function key definer which allows user defined commands to be assigned to the function keys. These can then be executed in DOS on a single key-press, just as they are in native BEC mode.

A file 'unerase utility which will help you to recover accidentally deleted files (provided the file's data hasn't already been overwritten).

A file find utility which can optionally search from one to four drives for a specified file, identifying either the first or any number of Ocairrences of the file. A further option is to switch to the appropriate disc/directory when a file is found, or to simply display its path, leaving the current path unchanged.

A 'Fast Root' to permit the 512 to be started up much more quickly than from the normal 640K ADFS format boot disc. Pan of the speed gam is because some of the code is loaded from ROM or sideways RAM, while DOS Plus itself is loaded dfrectly from the much faster WOK DOS Plus format 512 disc.

A hard disc panitioner, allowing any size in I Mb units to be allocated to the two partitions.

Essential Software, PO Box 5, Groby, lekestershire *LE6* OZB

Interactive Software Services

Interacter from ISS is a FORTRAN development system. Users of FORTRAN in Pcs or the 512 will be aware that its standard input/output capabilities are extremely Innited and are very much a DIY task in many implementations. Even in more sophisticated versions of FORTRAN, such as those found on larger systems, any ready to use I/o routines are likely to be unique to the compiler in use.

The result of this is that the machine spedfic areas of programs, particularly screen/hard copy input/ output, are not readily transportable to other systems or perhaps even between different peripherals (e.g. a plotter instead of a printer) since they must be re coded by the user.

Interacter is designed specifically to overcome the difficulties of writing presentable user interfaces in FORTRAN by providing numerous ready written routines which can be called from user code to do the job for you. The technique is that, after writing your program to carry out the main processing, the results are passed to Interacter in a pie-defined consistent format simply by calling the approprate supplied library routine.

This greatly simplifies the task of producing new programs, since all that is left to the user are the main

functions of the program. Peripheral interfacing such as console, printer or plotter output can be handled in a consistent manner by Interacter's ready written routines.

However, the benefits of divordng these functions from both the hardware and the compiler are more significant than simply easing the production of new programs, though this in itself is a welcome feature.

Interacter is available for a wide range of machines. These include quite large systems such as Hewlett Packard, DEC, VAX and SUN, as well as IBM Pcs nsmg variety of screen graphics display adaptors such as MDA, CCA, ECA, MCGA and VGA. Of interest to Acorn users, it is also produced in versions for both the 512 (using Microsoft v.4.1 or Prospero compilers) and the Archimedes (either Arthur or RISCOS using Acornsoft's release 2 compiler).

The consequence of such a development package being availabk for a wide range of hardware is that Source programs can be much more easily transported between different systems since the I/o interfaces remain fixed. while the core functions and syntAX of the language should already be either common or very sifliliar between different compilers, by employing Interactor the same compatibility is extended to input/output *facilities.*

Pricing of the package varies with the size of the system and the number of users, but 512 users will probably find the 'personal' version at £;99.00 plus VAT provides all that is needed. A demonstration disc can be obtained from Iss which gives a good overnew of the package's capabilities for prospective purchasers. The impression conveyed is that this is a professional and comprehensive package which should be of considerable beneflt to FORTXAN users.

For a demonstration disc, details and prices of the pa&age for the various implementations, contad Lawson Wakefield at:

Interactive Software Services, 25, St. Michael's Close, Penkridge,

Stafford STl9 5AD, Tel. 0785 715588

Margolis & Co.

COMM+ by Margolis & Co. is a communications processor which allows the Master 512 to be used as an intelligent terminal. Both Viewdata and ANSI standards are provided, allowing full asynchronous communication with other computers. COMM+ can also be used as a corms control package with most moderns, giving access to email and bulletin boards such as Prestel and Telecom Gold.

COMM+ allows the use of most modem features. It caters for split baud rates, auto dial, auto answer, and various other facilities. It also offers file transfer with full enor checking to and from other computers which are runrung either COMM+ or a compatible cOmms package, providing a very flexible

extension to the 512's capabflities As it is the only commercial communications paCkage which is known to work in the 512 it is particularly good news that virtually all the required functions are provided.

COMM+ also provides a built in programming language, allowing screens to be customised and foreign characters to be used. This language also offers simple database facilities and hilt access to the operating system for file management. The programming language is simple to use, allowing most communications links to be set up automatically and also catering for subsequent monitoring.

The documentation is supplied in an A5 ring binder that lies flat on the desk and allows pages to be removed and used separately. There is a good index to the various secilons, split into the "off line tutorial" and the "cookbook". The tutorial is very good, catering for both the regular hacker using the 512 and the beginner who is new to comrns. The "COMM+ cookbook" delves into the intricacies of COMM+ from staffing to troubleshooting. The "reference guide" documents such goodies as the editor. This gives the ability to edit and format text before transmission. You can also Save text and load documents prepared into a word processor. The editor automatically removes control codes from the text ready for transnusswn.

14 different terminal ernulation modes are also documented in the reference guide, covering alternatives such as VT52 and ANSI standards. Kermit support is one of the many additional facilities provided by COMM+ and is also documented in this section.

The "Language Manual" documents the built in language. This is the bit for the hacker to experiment with. It offers the ability to automate and monitor communications using a logical structure. Most should benefit in some way from this faclity. The software is easy to use and flexible. The manual is comprehensive and the tutorial is very good

Overall this package should prove to be flexible and capable of almost anything required. rt is well worth the normal asking price, but 512 users can obtain spedal terms. The 512 package consists of both the 512 specific version and the normal PC version for about 20% less than the normal price of the PC version alone.

For the current price and further details conbct:

Margolis & Co., 105 Foundling Court, Brunswick Centre,

Marclmont St, London, WCIN IAN. Tel. Ol 278 3032

Permanent Memory Systems

Permanent Memory Systems are well known for their BBC micro products, notably 'Genie' and 'Genie]

unior'. However, one item In the Genie family, the 'Genie Watch' is of particular interest to 512 users who employ a mode' B or B+ host.

The PMS Genie Watch is an EPROM mounted on a carrier which contains an additional chip and Some rather clever circuitry such that it is of a size that if plugs in to any ordinary EPROM socket. when so fined the Genie Watch provides a permanent, battery backed real time dock of the day, date and time to H and R+ miacs, giving wirdiar facilities to those of a Master 128's real nine clock atTC). PMS estimate the battery life of the Genie Watch is ten years, a vast improvement over the batteries provided in the Master 128+

In addition to providing additional date and time related functions in BBC native mode such as the ~nME w~d or 'rUNT TIMES' in BASIC, automatic date insertion in wordprocessor documents and so on, the clock also supports OSWORDS 14 (&E) and 15 (&F). The first of these is the call by which DOS reads the current system date and time during 512 start-up.

Owners of model B or B+ hosts nonnallally either have suitable 'DATE' and 'TIME' commands in their AUTOEXEC.BAT file requestin$_8$ tedious and error prone mannal entry of these, or forgo entirely the very real advantages of DOS file date/time stamping. while file date/time stamps are not strictly necessary for the general operation of the system, considerably helpful, sometimes vital file information is lost if accurate dates and times are not available to reflect the most recent updates or backups of unponant files.

A very simple, quick modification to DOSPLUS.SYS using EDBIN is all that is required to allow automatic reading of the date and time variables from the Genie Watch during system-boot in non-Master hosts. This allows completely automatic and error free booting of the 512 without the need for manual intervention. Before starting the modification you can check full details of how to use EDBIN in chapter 11 if you are not familiar with this utility.

The modification steps for DOS Plus 2.1 are as follow:-

1. Create anew floppy boot disc using the DISK command.

2. At the end of formaffin& when prompted, confirm that the disc should be made bootable. This copies the system files to the new disc.

3- on completion, exit the DISK program back to DOS flus.

4. Set the file DOSPLUS.SYS on the new disc to read/write by the command

FSET B:DOSPLUS.sYs [RW].

ahese e'camples all assume that your system drive is A: and that the newboot discisindnve B:).

5. Load the new copy of the operating system into EDBIN by the

command

EDBTN B:DOSPLUS .sys.

'This should produce a display similar to this--
size ot tile = 014780 bytes Nnr~}er of bytes read = 014180

6. At the '-' prompt, after EDBDJ has confirmed the number of bytes in the me, enter z 7 9sF. The display should now appear as:-

-. 79~ 1693.79SF 75 03 ES 03 CO E9 90 FS C6 Os D2 £;A FT BO CE

7. Enter the number 90, followed by cursor right and another entry of 90. The display should now be amended to:-

90 ~O ES 03 CO ES 90 F6 Ce 06 fl2 'A Fr SO Os Ba

8. Press CTRL-C to leave edit mode. Enter 'W' to write the file ba& to disc.

10- Enter 'Q, to leave EDBIN.

Note that the segment value shown in these examples may be different in your own system. This is irrelevant, as the segment number is defaulted in these commands. The amendment changes a 'jyrz OS' command to two 'NOP'S. The jump is normally taken at this point in the stanup procedure if the host machine is not a Master, removing this jump therefore permits reading of the RTC date and time in all host models.

If all has been done correctly you should now be able to use the new disc to boot the 512, when you will find fhat the date and time are automatically set by the OSWORD &E call to the Genie Watch. If the boot fails you have made a rnistake. Restart the 512 from your original boot disc and repeat the operation. You need not restart from the beginning by re-formatting the disc. Instead you can set the new DOSPtuS.SYS copy to read-write, copy your original unchanged~ version to it and repeat the changes from step 4 alove.

On successful testing the last task is make the changes permanent. On your new boot disc reset the amended file to read-only, then set your 'real' boot disc's copy of DOSPLUS.SYS to read/write. Copy the new DOSPtUS.SYS file to the original one and the job is almost complete. (Copying the read~nly file to the read-write version also changes the attributes without further use ofFSET). The only item

remaining is to delete the now redundant 'date' and 'time' commands from your 'AUTOEXEC.BAT' file.

The Genie Watch runs m any Model B or B+ micro whidi has a free ROM socket on the mother board, or on most ROM expansion boards. It will not work plugged directly into an STL 32K or 256K board, though if a mother board socket is, or can be made available this can be ueed successfully.

The Genie Watch normally costs £;29.90 plus £;1.00 P&P, but as a special offer to readers, VMS will supply the Genie Watch at a price of £;24.90 plus P&P. You should mention the Technical Guide when you order to obtain this discount of over 15%.

Permanent Memory Systems are at 38 Mount Cameron Drive North, St. Leonards, East jaibride G74 2ES Tel. 03552 32796

Shibumi Soft

Shibumi Soft's sole 512 product to date is a disc package called Problem Solver, the purpose of which is to remedy or reduce Sorne of the incompatibility problems inherent in the 512 and its versions of DOS plus.

In early versions of Problem Solver, that is, up to the third quarter of 1989, although the program worked in a Master 128 it had proved extrernely or enfirely unreliable in other hosts, especially using version 2.1 of DOS Plus. As a result of a visit to the UK from Portugal by the proprietor these difficulties were ironed ont and since September 1989 the program has exhibited a much higher and acceptable level of reliability in all machines.

It can now be said that, within th; range of its capabilites, Problem Solver is a remarkable product. Since it operates effectively by patching DOS Plus, it convincingly demonstrates that with very little more effort and a little more unagination Acorn could have produced a much more 'PC compatible' implementation of DOS Plus.

Problem Solver operates by filtering program code as it loads from disc (or optionally by filtering after loading but before problems are encountered) in such a way that many of the instructions that would hang or crash the 512 are amended to a different 'special' invalid operation code.

An 'invalid opcode' error occurs when these amended instructions are executed, but Problem Solver intercepts the invalid opcode vector and so takes control when such an interrupt occurs. The invalid instruction, if known to Problem Solver, is then executed by means of kgal calls. Following this the results are passed back to the calling appikation in such a way that it behaves as if its original (invalid) call had been successful.

The malor areas in which this type of fix can assist are limited to screen display and certain types of

keyboard access. That said$_1$ there is a substantial number of software titles which will not run in the 512 normally, but which will run with Problem Solvers assistance. It is fair to say that, because of the nature of the problems this package addresses and can compensate for, the majority of the listed software is games. However, it should be noted that Some 'serious' packages are also included in the list, particularly those where a program's original problem concerns reading key presses rather than reading input ASCII characters.

A few additional *facflities* are included in the program, such as the ability to slow down the 512, presumably *as an* aid to playing *games* which run too quickly *in* the 512. Also there *is the* ability to change screen mode on hot-keys, permitting *manual* assistance for some padages that don't change (512) mode correctly.

For Model B/B+ owners there is also emulation of the master's numeric keypad keys, but these are mapped onto the function keys at fixed locations which cannot be changed. This facility can be toggled on or off, but of course normal function key operations are lost when re-mapping is switched on.
A list of problem packages which this program can assist is not included here, frankly because it would occupy two or three pages. That fact in itself demonstrates the effectiveness of this product within the confines of the type of incompatibility that it tackles. Problem Solver is supplied on disc, accompanied by a limited implementation of a DOS mouse driver and a program called vJsr which assists with some displays. A DOS screen save is also included. The price is ~s5 and Problem Solver is supplied by:
Shiburni Soft, It. Prof. Camara Sinval 138, 4100 Porto, Portugal

Solidisk Technology Ltd.

The pc+ is a 512K byte memory expansion board which was manufactured and sold by STE who were for some years active in the Acorn marketplace. They have however, since early 1989, entirely moved their operations into the PC market and no longer produce or sell Acorn ad-ons. The Pc+ was the first commercial 512 memory' expansion board and during its life was produced in two versions.
Note that the information here relates only to the later and more reliable version 2 and is included for those who may wish to consider the purchase of a second hand Pc+.
The PC+ attaches to the 512 by means of fly leads and by plugging into the 512's EPROM sockets; the EPROMS then plug into the sockets provided on the PC+
The issue 2 rc+ is wen built and does not consume excess power (which it takes from the 512 board) hence it generates little extra heat. A Master with adequate ventilation or a fan is unlikely to have problems of overheating. If fitted to an external co-processor adaptor this should present no difficulties either. The spedrnen board which was tested ran in a Master 128 for two months without problems, indicating that this expansion should be reliable.

The pc+ sits neatly on top of the 512 board, adding about 2 cm to its height. The board fits easily into a Master case when the internal tube is used, or into an external co-processor box. The tu' shaped board is supported by the EPROM sockets into which it plugs, and extends across the 512 board to the base of the 'U' where the fly leads run off to their soldered connections on the main 512 board. The cutout

portion in the centre of the 'U' is to allow the 80186 to dissipate heat.

The board is sturdy and well made but it could be fitted more elegantly. Solidisk chose to cut two of the pins of a chip on the 512 board and solder to the stubs, rather than lifting the pins from the main board. This makes subsequent removal of the pc+ much more difficult, for example for repair, as the modified chip on the 512 board must be removed and replaced to return the board to normal operation for testing.

When using the Pc+ a rnodffied version of DOS Plus 1.2 can be used which takes into account all of the extra memory. This version *was* supplied by STL with the Pc+, but was never issued by Acorn. (An umnodified version of DOS Plus I .2 ignores the extra memory) DOS Plus 2.1 has a modified memory map which recognises the extra memory giving 704K of user RAM with the pc+, but 2.1 will not recognise the top 256x of the expansion. This is only significant if you have a package that requires 700K pIus and ix}s+ 2.1. Since PCs and clones are limited to 640K of main RAM (i.e. excluding EMS) this is unlikely to present a problem. Most packages will not use anything more than 640K even if it's available. In fact Lotus 1-2-3 is the only
package known to use the memory beyond 640K as standard.

The original disadvantage of the pc+ was that, since it required modification of the Master 512 board any Acorn warranty was void. However, as all purchases will now be of second-hand units it is likely the Pc+ wili already be fitted to the 512, which will be out of warranty anyway. So long as you see the board working reliably this point should not unduly deter you. However, prospective purchasers should note that the issue two PC+ is much preferred over the issue one version, which was neither so well made nor so reliable.

In suatmary, if you are able to find one the Pc+ offers increased memory at reasonable cost and is a wonhwhile 'add-on' if you (want to) use larger memory packages. However, bear in rnind that you will almost certainly have to buy it attached to a 512, therefore the cost will effectively be doubled. Cuser installation is not recommended.)

The final point to consider is that STL no longer supply or support Acorn add-ons, therefore the servicing of a faulty 512 fitted with a PC+ might prove difficult.

Tull Computer Services

Tull Computer Services is another recent and welcome addition to the all too limited numter of snppliers of 512 products. At the time of writing Tuil have just released a second 512 product which has not been evaluated$_1$ therefore only brief details are included. Their lint product has however$_1$ and it is one which has been long awaited by some 512 users, an 'Ms-type' mouse driver.

As mentioned in the hardware introduction in this book, for reasons best known to themselves Acorn and Digital Research jointly or severally chose not to produce a generic mouse driver for the 512, instead supplying a version which only works with the bundled CEM collection.

Cliff Bowman, the proprietor of TCS was one of the many users who felt the need for a 512 mouse driver for those DOS packages which cannot be used without one (Generally art, CAD and graphics design packages) and therefore be set about producing it. The resulting program has been commercially available to 512 users since the latter part of 1989.

The Tull Mouse Driver is supplied on disc with a comprehensive manual. Although the use of a mouse driver is ljznited to loading the mouse program, loading an application configured for a mouse and using it, the manual supplied with the mouse driver includes extra detail whidi should be of interest to 512 users.

The manual begins with a short but interesting overview of the development of the mouse as a standard user interface. 'This is followed by details of the files on the issue disc, notes on program

compatibility and a few minor limitations of the 512 irnplementation as compared with a mouse driver running in a true PC. One of the disc files is a README file containing the latest information on compatibility and enhancements to the program.

The bulk of the manual, the programmers gtaide follows. 'This section provides function by function details of the various calls to INT $33h_1$ the mouse internipt. INT 33h is not supported by the 512's version of DOS Plus as standard, hence it is not detailed in the appendices of this book, but the TulI manual provides all the necessary information for those who wish to write their own applications code to interface with the inonse driver.

The remaining sections of the manual lists known compatible and incompatible programs plus 'Plug corner, containing brief details of other sources for 512 products or information.

The Tull Mouse Driver requfres 5K of memory in the version tested.

Although Tull actively encourage user suggestions and ate keen to bnprove the program where possible, it is extremely unlikely it will grow beyond 6K, so even in unexpanded 512s memory usage is minima'.

At the time of writing the mouse driver does not itself provide a mouse pointer in four colour modes, partially because this would maease the program's size substantially, and partially becanse many

graphics packages irnplement their own mouse pointer. However, this ndght limit the use of the program with some appikations. The current compatibility list as supplied by Tull is shown below.

Title Publisher
Autoroute Nenbase Ltd.
Autosketd~ Autode~

DeLuxepaint II Eledroric Arts
Dr. Halo JIr Unknown
Elite Firebird Software
flight Simulator 3 Microsoft
GEM 3.0 and 3.11 Digital Research
Interactor Iss (See above)
Mouse Cursor Designer Shareware
Norton Editor 1.SC Peter Norton
Optiks Shareware
PC Paintbrush 7-Soft
Shareware
Reflex Borland
Turbocad 1.5 and 1.52 Borland
Ventura Publisher Xerox
Word 4.0 Microsoft

Tull's Mouse Driver has been tested with DOS Plus versions 1.2 and 2.1 using XIOS 1.01 and 1.03. It[1]is also happy to run with Problem Solver, which, it should be noted may be required to run some of the above programs successfully in the 512. Also, sorne of the packages shown will not run in an unexpanded 512.

Tull hope for feedback from users to expand the list of known compatible software, therefore readers are advised to contact Tull directly for latest information if there are doubts about any particular package. rt is requested that telephone enquifles are restricted to evenings only at present, though this may change in due course.

The Tull Mouse Driver is a thoroughly competent product which, given the limitations imposed by the peculiarities of the 512 (compared with true Pcs) provides an excellent implementation of a standard DOS mouse driver. At the time of going to prsss the price is £;30.00 but readers should again check with Tull before ordering.Tull Computer Services, 115 Ganirnons Lane, North Watford, Herts. WU2 5JD. Tel. 0923662240

I: Glossary

Not all of the following terms are explicitly referenced or used within this book. However, as readers further explore DOS, '86 series programming and the 512 it is inevitable that many of these words or abbreviations will be encountered, therefore they are included. That many of the entries are abbreviations or acronyms is unavoidable, as the computer industry continually is forced to invent new terms by which to define or refer to previously non-existent hardware or software features or functions for which no name exists.

ANSI
: American National Standards Institution - A body which decides on certain agreed standards and conventions for the computer industry.

ASCII
: American Standard Code for Information Interchange - an agreed standard for representing character values by two hexadecimal digits (i.e. a byte)

ASCII (string)
: An ASCII character string which is terminated by a null (hex. zero) byte. Such strings are used by many of the DOS interrupts concerned with files and directory paths.

ASCII (string)
: A character string which in DOS, by default is terminated by a dollar ($) character. The terminator byte can, if required, be amended to another character. Such strings are used by DOS in string output functions.

Attribute (file)
: One of the bytes stored in the directory entry for a file or subdirectory which specifies the actions which can, or cannot be performed on the file. Each bit of the byte indicates the status of each of the possible conditions which may apply, such as 'read only', system and soon.

BAT
: The file extension used by DOS to recognise an executable file containing command line commands as text strings, each terminated by a carriage return.

BDOS
: The Basic Disc Operating System. That part of the core of DOS which is normally invariant and concerned with memory and program management and device independent input/output.

BIOS
: The Basic Input Output System. The part of the DOS operating system which is normally customised by the hardware manufacturer to cater for the specific needs of the type of device controller employed in a particular type of PC.

Block device

An input/output device which normally operates with or on complete blocks of data rather than individual characters. For example, discs are block devices, since a minimum of one cluster will be read by DOS, even though a program may request a specific larger or smaller quantity of information to be passed to it.

Boot record

In PCs, the boot sector on a bootable disc. The first sector of such a disc contains information which enables the machine to load the operating system from a cold start.

CCP

The Console Command processor. that part of DOS which is responsible for translating user commands into DOS function calls.

CHCB

CHaracter Control Block. An area of memory set aside in DOS to describe the attributes and capabilities of of a character device.

Character device

An input output device which operates at individual character level. For example, the keyboard is an input character device, while the screen display is an output character device, as is a printer. A communications port is an input-output character device. In DOS certain functions may appear to read or write such devices in blocks, such as writing a complete text string to the console, but in fact such operations are performed by repeated calls to character routines.

Cluster

The minimum size of the allocation unit when a disc file is either newly created or extended. Clusters may vary in size from disc to disc, but are always 1, 2, 4 or 8 sectors in length. Sector sizes also vary, therefore there is no standard cluster size.

Cold-start

Also called a cold-boot. The process of loading a machine when it is first switched on. See also Warm start.

CMD

The file extension used by CP/M, CCP/M and DOS Plus to recognise an executable file which contains a machine code program. Essentially similar to a COM files, CMD files differ in the way they are loaded and executed at run time.

COM

The file extension used by MSDOS, PCDOS and DOS Plus to recognise an executable file which contains a machine code program. Essentially similar to CMD files, COM files differ in the way they are loaded and executed at run tirne.

Console

The name given to the normal user interface to a PC. It is commonly used to refer to both the input and output devices as though they were a single entity, though in DOS interrupts quite separate functions are provided for input and output, with the exception of INT 21h function 6.

Console mode

The status of the console device. This may be altered to allow or preclude certain automatic processes. For example, in the default mode provided at power-up a tab key depression at the keyboard is automatically expanded on output into eight spaces, while CTRL-S will stop a scrolling screen display. These functions can be enabled or suspended by changing the console mode.

Cooked mode

The term used to describe input or output access to a character device when it is treated as a block device. DOS permits all devices to be treated as files, including the keyboard and the screen. When a character device is opened as a file, subsequent input-output access is referred to as 'cooked mode'.

CCP/M

Literally, Concurrent Control Program for Microprocessors. CCP/M is a Digital Research operating system developed for 16-bit machines from the original 8-bit CP/M. CCP/M allows the user to run several different tasks from the same console simultaneously.

CP/M

Literally, Control Program for Microprocessors. CP/M is the highly successful pre-cursor to DOS and CCP/M developed by Digital Research for 8 bit Intel 8080 and Zilog Z80-based machines in the 1970s.

CP/M86

Another development of CP/M. it was intended to be the direct growth path for users migrating from 8-bit CP/M to 16-bit machines.

Device driver

An extra or separate part of DOS which supplements the BIOS. Loaded as a program after system boot, they are used to provide device control for input\output functions for hardware or software interfaces not catered for by normal, standard DOS interrupt services. For example to drive graphics plotters, or to read data from an IEEE interface, a specffic device driver must be provided, since these devices are not provided for in standard DOS.

DMA

Direct Memory Access. The process of transferring input or output data directly to or from areas of memory and peripheral devices such as discs. Depending upon the type of processor, bulk peripheral transfers are carried cut either by DMA, as in the 80186, or under NMI control, as in the 80286.

DPB

Disc Parameter Block. An area within DOS which is used to describe the characteristics of the particular disc format in a drive at the time. As different formats of disc are used from time to time the contents of the disc parameter block are amended accordingly. These need not be unique and more than one drive may be using a particular DPB at any time if they are both using the same format of disc.

DPH

Disc Parameter Header. An area within DOS used to describe the characteristics of each disc

drive. Part of the information in each DPH, for example the pointer to the DPB, will change as different formats of disc are used.

DR-DOS

Digital Research DOS. The successor to DOS plus and the current version of DOS supplied by Digital Research.

DTA

Disc Transfer Area. The area within a program which is set aside to hold data which is to be transferred to or from disc. The DOS interrupts which action such requests require that processor registers are set up to point to the DTA before the call is issued.

Edited mode (Console)

The default mode in which the console operates. Tabs are expanded, CTRL-C abandons input or output, CTRL-S pauses screen display, CTRL-Q resumes it, CTRL-H performs a backspace and so on.

EMS

Extended Memory System. The system employed by DOS and later versions to manage memory outside the normal limit of 640K. Introduced by Intel and the Microsoft Corporation in 1985 to provide a standard under which MSDOS Windows would operate to provide a type of multi-tasking, previously unavailable to MSDOS. In the correct hardware/software system one segment of memory, known as the page frame, is set aside to allow 16k byte blocks to be swapped with the rnemory beyond 640K, in effect giving overlaid memory facilities. This facility requires at least an 80286 processor, since page frame changes must be carried out under a hardware NMI.

EXE

The file extension used by MSDOS, PCDOS and DOS Plus to recognise an executable file which contains a machine code program. Different in concept to CMD or COM files, EXE files permit values to be pre-assigned to various processor registers at load time, hence they are not, like COM or CMD files, limited to a maximum size of 64K at load. EXE files may be produced from a variety of source language modules, but always consist of a number of object modules which are linked to produce the finished program.

FAT

File Allocation Table. An area of disc which contains a table with one entry per cluster for the entire disc. Each directory entry on the disc points into the FAT to give the start cluster for the file or sub-directory. Each of these in turn point to the next (or last) cluster in the chain which makes up the corriplete file or directory. The FAT is used by DOS to permit files to use numerous small, separate areas of disc, rather than having to be contained in one large contiguous area. Over time files may therefore become fragmented, when ideally the disc should be re-organised.

FCB

File Control block. An area of memory within a program which must be set aside and identified to DOS before a file or device may be opened for input or output. On a successful open DOS supplies various information about the file by entering data into the FCB so that it can then be accessed by the program.

File handle

A number returned by DOS interrupts which open a file or device for input and/or output. After such a call the file handle is one of the variables which must be passed to DOS with every call to access the file or device. After opening, the handle therefore uniquely identifies to DOS the device or file to be accessed.

Fragmented (files)

The description given to files which largely or even entirely consist of single or very small numbers of separate clusters scattered, effectively at random, all over a disc. This is an unavoidable consequence of the file management system used by DOS to permit all areas of a disc to be used no matter how small each one rnay be. The minlinurn fragment is one cluster, but these can be as little as 512 bytes depending on the disc format used. Fragmented files usually make themselves known by extended access times, since DOS and the disc drives have to work harder to retrieve or write data. Files can be 'unfragmented' by copying them one at a time to a newly formatted disc.

Intel

The manufacturer of the 80 and '86 series of micoprocessors, including the 512's 80186.

Interrupt

iterally. an interrupt to the norrnal execution of a program. In DOS three types of interrupt exist, internal hardware interrupts, external hardware interrupts and software interrupts. The last of these are generated by applications prograrns by a machine code instruction when requesting one of the services provided in DOS.

MCB

Memory Control Block. This is an area within DOS which is used to allocate a block of memory to a program which is currently executing. If the program is a normal external program the memory block recorded in the memory control block will be freed when the program terminates. Programs which remain resident (TSRs) do not release their memory blocks, hence the number of MCBs determines how many TSR and external programs can run sirnultaneously in DOS Plus this is 32.

MSDOS

Microsoft-DOS. The version of DOS produced by the Microsoft Corporation for use by manufacturers of 16 bit personal computers. (See also PCDOS)

Nibble

Half a byte, or four bits. A nibble is the rninimum amount of data which can express all the hexadecimal digits from 0 to F. When concerned with memory addressing a nibble is also the amount of storage required to fully address one paragraph.

NMI

Non Maskable Interrupt . Literally, an interupt which cannot be deferred. All NMIs are generated by hardware events and consist of two types in 86 series nachines. Internal hardware interrupts are the highest priority of NMI and are generated by the processor because of a catastrophic event such as a store parity error. The second level of NMIs are generated by peripheral hardware requiring attention, such as a key being pressed on the keyboard, a sector being transferred from disc or, in the 512, a tube data transfer pending. NMIs immediately stop all application program execution and non maskable interrupts until the NMI is fully processed.

Offset

The address, expressed in numbers from zero to 65,535 inclusive, which defines a spedfic byte of memory from a given start-point. In '86 series machines the offset can be combined with a segment address to specify any single byte of memory from address zero to address 1,048,575.

Page

An area of memory 256 bytes long. A page is 16 paragraphs, hence a page can be addressed by a single byte, or two nibbles.

Page Frame

The segment of memory designated to be used to swap with extended memory in EMS systems. 16k bytes may be changed on each call, hence four calls can give access to a complete extra segment. Page frame transfers are carried out under NMI and therefore require at least an 80286 processor, since the 80186 and earlier versions do not support NMI memory transfer.

Paragraph

A contiguous 16 byte area of memory. A paragraph can be addressed by a nibble and in cenain DOS interrupts concerned with memory allocation the amount of memory must be specified in paragraphs.

PCDOS

The version of DOS provided with IBM PCs. If is substantially slinilar to, in terms of developments and facilities, MSDOS, with which it usually shares version numbers. PCDOS differs internally to take account of certain functions which are provided in hardware in IBM machines.

PSP

Program Segment Prefix. A one page area of memory which is created to immediately precede a COM file in memory at load time. It contains the contents of the vectors for INTS 20 and 22h to

24h at the time of the load, the far return address to the general function despatcher (INT 21h) pointers to the environment string and the command tail of up to 127 characters given when the program was called.

Raw mode

A term applied to console input/output when no automafic actions are to be performed on the data. This type of I/O may be obtained by changing the console mode, but is much more conveniently available, regardless of the current console mode, through INT 21h function 6. This may be used by programs which wish to process data with no interference from the operating system. In other words, CTRL-C is ignored, tabs are not expanded and so on, but instead the characters are passed directly to the caller as they occur.

ROM BIOS

A mis-nomer outside IBM machines, this is the term applied to a range of interrupts and functions provided by firmware in IBM PCs, but which is otherwise provided by software. ROM BIOS emulation is provided in the 512 to quite a high degree of conipatibility. Functions provided include pixel plotting, cursor positioning and much faster screen output than is possible through the general function despatcher, which execute by repeated calls to ROM BIOS functions. Most commercial applications almost exclusively use ROM BIOS functions for rapid screen handling.

RSX

Resident System Extension. A special type of program only available Digital Research operating systems, prepared originally from a CMD file, RSXs 'attach' themselves to the operating system automatically when loaded to provide permanent extra functions.

Segment

A contiguous area of memory 65,535 bytes long. A segment must always starts on a paragraph boundary. One segment is the largest unit of memory which can be addressed by sixteen bits, two bytes, or a single register in '86 series processors.

System data segment

The area within DOS Plus which holds the operating system variables used to control all aspects of the system. No standard calls are provided to directly alter the contents of this area, though systems programmers may need to access the data to provide special or added functions not normally available.

XFCB

Extended File Control Block. The file control block which must be employed if access to the attributes of a file is required, rather than merely input/output. An XFCB must also be used if the

file to be processed has the read only, system or hidden bits set.

J: The Programs Disc

The program disc has been specialy put together to complement the 512 Technical Guide, while at the same time helping you make the most of your 512. Its contents include:

Demonstration Software

Demonstrations of many of the third party programs detailed in appendix H are included on the disc for you to run in your own 512. All the programs provided are functional and can be tested in your own applications in the way you choose.

512 BBC BASIC

This is one of the many programs Acorn should have supplied with the 512+ Written by Richard Russell, author of the BBC BASIC which Acorn did supply with the Z80, it is a very dose emulation of the BBC rnicro's inbujit BASIC, but it runs in the 512 under DOS Plus.

The commercial PC version of this package would cost you £100.00, but this special 512 version which is freeware (provided that you know where to get it!) gives facilities much closer to those of a real BBC micro. It suffers far fewer of the limitations normally imposed by DOS in PCs because some of the BBCs native facilities can be used. One not available in the PC version of the package, for example, is true BBC screen displays. Two versions of the program are includel, a more efficient small version for programs of 64K or less, while a large memory version permits programs of a size you could only dream of in the RISC micro, even with a 6502 second processor.

The package contains dozens of ready written examples and demonstrations to ensme that you can have your first BBCBASIC program running within moments of loading the language.

A commercial version of the Z80 version of BBC BASIC which includes a 500+ page manual is available from M-Tec Computer Services, telephone 0603 - 870620.

A86/D86

Normally available in shareware (at almost the cost of this disc on its own) A86/D86 is *the* assembler & debugger for the 86 series of Intel microprocessors. It is so well written that it is 100% compatible with the 512. Supplied in archived format to conserve disc space, this package provides the complete range of tools needed to produce professional quality software for the 312.

The package includes extensive documentation on the disc which, together with the introduction to the

80186 and the interrupt call information in this book and the example programs provided, should allow anyone to begin to explore 80186 assembler programs in a short time. The assembler is particularly easy to use, being unusually flexible in its requirements for source code layout and organisation. The debugger allows programs to be single stepped, with the ability to display several different areas of the 512's memory, while at the sarne time continuously showing all the 80186 registers' contents.

Example Programs

The 512's version of DOS Plus officially claims only compatibility for DOS interrupts 20h to 27h, but in fact many others an implemented, some to quite surprising degrees of completeness. Source code programs ready for immediate assembly by A86 are provided to show how 80186 assembler code programs are constructed. In addition they demonstrate many of the functions which you will most frequently need to use in your own programs.

Examples of the general function despatcher (INT 21h) include string and character input and ontput, disc file handling, the highly efficient techniques available for comparing areas of memory (e.g. text strings) as well as examples of the ROM BIOS calls (INT 10h) which an needed if you wish to control screen effects such as inverse and bold, the cursor position and formatted output to the screen.

K: Bibliography

In the following list where no author or ISBN is included the volume is not a commercially published product and can be obtained only from the manufacturers or their agents.

Advanced Disc Filing System, Acorn Computers Ltd.

Advanced DOS, Michael Hyman, MIS Press ISBN 0 9435188 83 0

Advanced MSDOS, Ray Duncan, Microsoft Press ISBN 0 914845 772

DOS Plus User's *Guide,* Digital Research

DOS Plus Programmer's Guide, Digital Research

DOS Plus System Guide, Digital Research

Master Operating System, David Atherton, Dabs Press ISBN 1 870336 01 1

Master 512 User Guide, Chris Snee, Dabs Press ISBN 1 870336 14 3

MSDOS Functions, Ray Duncan, Microsoft Press ISBN 1 55615 128 4

The Advanced User Guide, Bray, Dickens & Holmes; Cambridge Microcomputer Centre ISBN 0 946827 00 1

The DOS Plus Reference Guide, Digital Research, Glentop Press Ltd ISBN 1 85181 147 8

L: Other Dabs Press Products

Other Dabs Press Books

Dabs Press publishes a wide range of books on computer topics. There follows a list of some of our recent and forthcorning titles.

If you are interested in any of these books, details of how to obtain them an given at the end of the list.

BBC Micro & Master

Master 512: A Dabhand Guide by Chris Snee

ISBN 1-870336-14-3. Price £9.95. Programs Disc £7.95 inc.VAT.

Available NOW.

This is a comprehensive reference guide for all users of the Master 512, Acorn's PC-compatible add-on for the Master 128 and BBC Micros, and the companion volume to this book.

Highly practical in approach, the book provides detailed information on all DOS Plus commands, and explains how they differ from MS- DOS. It shows 'step-by-step' how to install and run PC applications on the Master 512, including useful techniques such as the creation of batch files.

In addition, the use and operation of the utilities provided with the machine are explained, many of which are previously undocumented.

Features of the book include.

- Summary of DOS Plus commands and reserved words
- Transient utility programs
- Differences between DOS Plus and MSDOS
- How to check if PC software will run
- The Master 512 disc set
- Use of hard discs

Chris Snee is a consultant in the fields of personal computers and mechanical engineering. His considerable expertise with the BBC Micro and PCs has been derivied from writing practical applications software, and troubleshooting. This is his second book, his previous one *Mastering the Disc*

Drive receiving much acclaim.

This book has the pleasing and informative style that Dabs Press seem to encourage. It is neither over-technical nor over-simplistic in approach, but deals with the subject in a logical and understandable manner that reveals the author as a master of *the machine"* (Micronet 800 *March* 1989).

BBC Micro Assembler Bundle by Bruce Smith

ISBN 1-870336-08-9. Price £4.95 (inc.VAT). Available NOW.

This is a five part package of materials for anyone starting out learning assembly language/machine code progranuning on the BBC Micro/Master Series.

BBC Micro Assembly Language is a 204-page introdnction to programmmg the machine in 6502 assembly language/machine code. It assumes no prior knowledge whatsoever, and takes you to a reasonable level of proficiency in the subject.

BBC Micro Assembler Workshop starts where the previous book leaves off, progressing further into the subject, with a host of useful type-in utilities, which are also informative in machine code technique.

The third and fourth parts of the package are two discs, one to accompany each book, containing the programs from the book. Over 90 programs are included on these discs. Finally, an extra booklet has been produced covering the further opcodes and features on the Master Series, bringing the books bang up to date. The whole package is available exclusively from Dabs Press for £4.95 *whilst stocks last.*

Master Operating System: A flabhand Guide by David Atherton

ISBN 1-870336-01-1. Price £12.95. Program Disc £7.95 inc. VAT. Available NOW.

Now in its second edition, this is the definitive reference work for programmers of the BBC Model B+, Master 128, and Master Compact computers. It also contains much material of interest to BBC Model B and Electron users. The book covers all the features of the Acorn machine operating system CMOS) including:

- All 'star' commands on all models
- 65C12 opcodes (including Rockwell additions)
- An new OSBYTE/OSWORD and other system calls
- Sideways and Shadow RAM programming
- ROM service calls (complete) and header code
- Driving the Tube in both directions

Also included is a complete list of differences between the various Acorn computers, and in one convenient place, all those vital tables that you need when programming your BBC computer.

The Shadow and Sideways RAM and Tube chapters are expanded to provide application ideas, and the book is liberally spnnkled with program listings.

"Senous users shouldn't be without their copy of this invaluable book" A&B Computing (November 1987).

Mastering Interpreters and Compilers by Bruce Smith

ISBN 0-563-21283-7. Price £14.95 incl. programs disc (incl.VAT). Available NOW.

This clear and comprehensive introduction to the often misunderstood topic of computer language interpreters and compilers emphasise the practical side of the art.

It moves gradually from the idea of a 'wedge' in the BBC computer's operating system, to a simple interpreter, a simple graphics language, threaded interpretive languages (incinding FORTH), and finally, a stand-alone compiler.

Listings of all the implementahons are given. To save typing time, these listings are also supplied as a disc

This book will give anyone with a good knowledge of assembly language the foundation upon which to build an interpreter or compiler of their own.

Mini Office II: A Dabhand Guide by Bruce Smith and Robin Burton

ISBN 1-870336-55-0. Price £935. Program Disc £7.95 inc.VAT Available NOW.

Bruce Smith and Robin Burton have joined forces to write this offidal tutorial and reference guide to the award-winning and revolutionary Mini Office II software. This book covers the BBC Micro and Master versions of the program.

New and existing users will find the book to be a veritable mine of information, covering the everyday use of all the modules, and providing much data never before published.

The approach is a practical one throughout, using worked examples for you to try yourself. Divided into logical easy-to-read sections, it deals with each of the Mini Office II modules, providing many hints and tips en route. You'fl get so much more out of your Mini Office II software after reading this book. The many features of the book include:

- File Management
- The Wordprocessor
- Mailmerging
- The Label Printer
- The Database
- The Spreadsheet
- Graphics
- Communications

VIEW: A Dabhand Guide by Bruce Smith

ISBN 1-870336-00-3. Price £12.95. Program Disc £7.95 inc.VAT. Available NOW.

Now in it's second edition, this is the most comprehensive tutorial and reference guide ever written about the Acornsoft VIEW wordprocessor, for the BBC Micro, and issued as standard (but without a manual!) on the BBC Master 128 and Compact computers.

No stone has been left unturned, and all aspects of wordprocessing are covered. In addition a suite of VIEW utility programs are provided for you to type in, including View Manager, an easily extendable menu-driven system for managing your documents.

Thorny subjects such as rnacros, page layout and printer drivers are revealed in Bruce Smith's well-known relaxed style.

"It's *very good... I liked it very much"* Radio London. *"much* more *to offer the* competent *VIEW* user... practical *and down-to-earth... for those who want a complete, thorough and readable guide to VIEW, then Bruce Smith is your man.* Beebug magazine (June 1987). *"This* is *the first cornputer book I've read in bed for pleasure, rather than* to *cure insomnia" Acorn User* (September 1987). "*Smith brings a depth of understanding to View which should appeal* to *both novice and regular User"* Micro User (November 1987)

ViewSheet and ViewStore: A Dabhand Guide by Graham Bell

ISBN 1-870336-04-6. Price £12.95. Program Disc £7.95 inc. VAT. Available NOW.

This is a complete tutorial and reference guide for the ViewSheet spreadsheet and Viewstore database manager for the BBC Micro model B/B+, Master 128 and Compact computers. Whether you wish to check your bank statement or run a million-pound business, this book is for you.

Every aspect of setting up and using a database and spreadsheet is covered, and numerous examples are

provided to guide you.

There are also a number of utility programs to help you get more out of the VIEW family, including programs which join two databases together, and help transfer spreadsheets into a wordprocessor. Overview and Viewplot are also extracted and explained.

The many features of the book include:

- Usable with DFS, ADFS or network
- Simple spreadsheets and databases
- Absolute and relative replication
- Building an invoice system
- Database design
- Use of SELECT and REPORT
- Using a printer
- Hints and Tips
- Overview and Viewplot

"If you are one of the people for whom the normal ViewSheet and ViewStore manuals may just have well have been the Rosetta Stone, then this book is definitely/br you...This guide is the sort of immutable reference tool that all serious users of the View business suite need...Having read two of the previous Dabhand Guides and found them both to be irreplaceable works, I for one am eagerly awaiting Dabs Press's next attempt to cut away more swathes of complexity from the software and hardware world." Electron User (June 1988). *You certainly feel that the manual has been put together by someone who has explored the facility thoroughly. The author seems aware of all* its *pitfalls and knows how to warn others to avoid them and get the best out of* it". Oldharn Evening Chronicle.

Z88

Z88 Dabhand Guide by Trinity Concepts

ISBN 1-87033640~7. Price £14.95. Available NOW

This is the rnost comprehensive guide for all users of the Z88 portable computer and is indispensable for anyone wanting to get the most out of their machine.

An of the standard built-in application programs, including (but by no means limited to) PipeDrearn, are covered and clearly explained using easy-to-follow examples, and many hints and tips are included en route.

In addition, the book also shows you how to transfer files between madilnes, using the optional link produds. No previous knowledge is required or assumed in the book, which includes much previously

unpublished information.

The many topics covered include:

- PipeDream
- The Filer
- Printing, and printer drivers
- EPROM and RAM cartridges
- Machine expansion
- The Diary, Calendar, clock, and Alarm
- File transfer to/from PCs, Macintoshes, BBCs
- a Modem communications
- Introduction to BBC BASIC

This book is from Trinity Concepts, the partnership who designed the Carnbridge Z88 Operating System software, and many of the standard applications. Thek understanding of the way the Z88 works is second to none, as you will quickly appreciate from the pages of this book. No serious user of the Z88 should be without this guide.

Z88 PipeDream. A Dabhand Guide by John Allen

ISBN 1-870336~1-5. Price £14.95. Available NOW

In this detailed and authoritative book, John Allen explains how to get the most out of PipeDr~am, the standard business *software* supplied with the Cambridge ma portable computer.

The book starts by covering the essentials required to write documents, produce spreadsheets, and keep database information. Thorny topics such as print formatting, the RAM filing system and the printer driver editor are covered in great detaii,with clear and concise practical guidance which cuts a swathe through the complexities.

John Allen is a professional broadcaster and writer, and well-known to magazine readers for his many informative articles on the

Psion Organiser

Psion Organiser Lz: A Dabhand Guide by Ian Sinclair

ISBN 1-870336-92-5. Price £;14.95. Available NOW

In this exciting book, Ian Sinclair, the UK's premier computer author delves into the new tz Organiser

from Psion, explaining how to use the various utilities and the built-i programming language.

Ian Sinclair has written over a hundred books on computers and other technology-related subjects.

IBM PC Compatibles

GW-BASIC: A Dabhand Guide by Geoff Cox

Price£;14.95 approx. Available Summer 1990.

In this book, Geoff Cox provides a comprehensive tutorial and reference to the programinlng language provided free with most IBM- compatible nadilnes. As well as a friendly and helpful tutorial in BASIC programming, the book contains a complete command reference, detailing every comnmnd in CW-BASIC with examples of it's use.

The book is also suitable for programming with Microsoft QUIckBASIC and Borland TurbonAsIC.

Ability and Ability Plus: A Dabitand Cnide by Geoff Cox BBN 1~70336-51~. Price £;14.95. Available Spring 1990.

In this book, Geoff Cox provides a no-nonsense comprehensive tutorial and reference to this popular integrated package for IBM compatible computers including the Ainstrad range.

Al] aspects of all the modules are covered, and by the use of examples, you are shown how to perform a range of business tasks and how to use the programs in conjunction with each other, including transferring of data

Supercalc3: A Dabhand Guide by Dr A Berk

ISBN 1-87033645-8. Price £14.95 Available NOW.

This is a complete tutorial and reference guide for one of the most popular pieces of software of all ttrn~the Supercalc spreadsheet, and in particular, versions 3.1 and 3.21 for the Anstrad Pc1512, "ciMo, and other IBM PC compatibles. The book is also applicable to certain degree to Superflalc 2 for CP/M computers.

Dr Berk specifically writes to appeal to both the beginner and niore experienced user, and packs the book with examples which should spark *off* many new ideas, based on your own work or home situation. Even those completely new to using computers will find the book perfectly comprehensible, yet the book omits nothing in its coverage.

Every aspect of setting up, using and applying the spreadsheet is described in detail. Commands, formulae, graphics, and files, to name but a few, are all explained in depth, and by frequent example.

Windows: A Dabhand Guide by Ian Sindair

ISBN: 1-870336-63-1. Price: £;14.95. Available NOW.

Windows gives the MSDOS user a view into the future, the way that the high power machines of the 90's, and their users, will operate.

You can take ~ advantage of this power with this Dabliand Guide to one of the most sophisticated operating environments yet written for the IBM PC and its compatibles.

In this book Ian Sinclair, the UK's premier computer author, provides
you with the definitive introduction and reference work for Windows, including the 286 and 386 versions.

The book gives simple sterby-step examples which help you install Windows on your computer...get up and nmr~g...use the numerous utilities supplied with the software to best effect...and gradually progwssto more advanced use of Windows.

It is packed with hints and tips that show even experienced Windows users how to get the best performance from their software, and also how to fine-tune it to get the very best from existing software.

The book shows you how to get your favourite non-Windows programs up and running under Windows. Step-by~tep and with nunerous hints and tips you need never see the M~flOS prompt again.

Ian Sindair also shows you how other Windows-based software can be run to best effect and take advantage of a common working environment to exchange data.

Samples of use include Excel, Arni, PageMaker and many more are given. And$_1$ of com$e, the book provides full details on all the Windows applications supplied with the system

Windows; A Dablicind Guide helps you to realise the full potential of Windows to become a true Windows power user.

WordStar 1512:A Dabband Guide by Bruce Smith

ISBN 1-87033&17-8. Price £;14.95. Available NOW.

This is the most comprehensive tutorial and reference guide ever written about the WordStar 1512 and

WordStar Express wordprocessors on the IBM/Arristrad PC and compatibles.

Both beginner and advanced user will find the book to be a valuable companion whether merely writing a simple letter or undertaking a thesis.
No prior knowledge of computers or worciprocessing is required, yet no stone has been left unturned, and all aspects of using the program are covered in Brace Smith's own injrnitable style.

The book is applicable to both versions of the wordprocessor and to the Amstrad 1512 and 1640 models, as well as other IBM compadbles.

Features covered include:

- Rulers and Margins
- Copy, Move and Delete
- Find and Replace
- Format and Justify
- Dot Commands
- Page Layout
- Using BDOS
- Hints and Tips
- How to use the Spelling Checker
- Step-by-step guide to mailmerge
- Complete Unique Reference Sections

Brace Smith grew up in the East End of london, and first became interested in computers whilst working as a technician in a hospital operating theatre. His hobby developed into a career and now he is now one of Britain's most prolific computer writers, with over 20 books published to date, and couniless magazine articles.

Acorn Archimedes

Archimedes Assembly language. A Dabhand Guide by Mike Ciruw

ISBN 1-870336-20-8 Price £14.95. Programs Disc £9.95. Available NOW.

Learn how to get the most from the remarkable Archimedes micro by *programming* directly in the machine's own language, ARM machine code. This is the only book that covers all aspects of machine code/assembler programning specifically for the entire Archimedes range.

For those new to assembler programming, this book contains sections which take you step-by-step through flew and exciting areas of Archimedes programming, including many examples using the

features of the RISC OS Operating System, including the co-operative multitasking environment.

- Practical tutorial approach with example programs
- Descriptions of all the processor instructions
- Using the Operating System, WIMPs and Vectors
- Co-operative multitasking explained
- Assembler equivalents of BASIC commands
- Sound and graphics in machine code

"The contents make the book a welcome addition to the manual provided with the computer. and will. no doubt, be an invaluable source of information for many owners of an Archimedes" Everyday Electronics (December 1988)

Archimedes First Steps: A Dabhand Guide by Anne Rooney

ISBN 1-87o33&~9. price £9.95. Available NOW.

This book is the ideal starting point for first-time users of the Archimedes, taking you through the first few days and months of owning and using the machine.

There is an abundance of software provided with the Archimedes, and Anne goes through the programs, telling you how to get them started, and how to get the most out of them.

Many hints and shortcuts for using the RISC Os Desktop are also discussed, as are many third-party commercial software packages in such fields as art, music and so on.

Archirnedes Operating System: A Dabhand Guide by Alex & Nic Van Someren

ISBN 1-870336-48-8. Price £14.95. Programs disc £9.95 inc.VAT. Available NOW.

For Archimedes users who take their computing seriously, this guide to the Operating System gives you a real insight into the micro's inner workings. flils book is applicable to any model of Archimedes.

The Relocatable Module system is one of the many areas covered. Its format is explained, and the information necessary for you to write your Own modules and applications is provided. This tutorial approach is a common theme running throughout the book.

The sound system is explained and the text includes much information never before published. The discerning use will revel in the wealth of information covering many aspects of RISC OS such as:

- The ARM instruction set
- Writing relocatable modules
- VIDC, MEMC and IOC
- Sound
- The voice generator
- SWIs
- Vectors and Events
- Command Line Interpreter
- The FilCSwitch Module
- Floating Point Model

Throughout the book, programs are used to provide practical examples to use side-by-side with the text, which go to make this publication the ideal tabIe~de companion for all Archimedes users.

A programs disc is also available containing all the listings from the book$_1$ and Sorne extra useful programs as well.

"Here is *an essential book for Archimedes programmers"* Micronet 800 (April 1989) *"A jolly good read. Lots of really useful information presented in an accessible and readable* manner..~this is a *clearly written, welt presented book. It is up to the usual high standards we hare come to expect* from *Dabs Press, and I wholeheartedly reconimend it* to *all who want to know more about their machine's operating system."* Archive magazine March 1989.

BASIC V: A Dabhand MiniGuide by Mike Williams

ISBN 1-870336-7W5. Price £;9.95. Available NOW

This is a practical guide to programnang in flASIC V on the Acorn Ardilmedes. Assuming a familiarity with the BBC BASW language in generaL it describes the many new commands offered by BASIC V, already acclaimed as one of the best and most structured versions of the language on any micro.

The book is illustrated with a wealth of easy4~follow examples.

An essential aid for all Archimedes users, the book will also appeal to existing BBC BASIC users who wish to be conversant with the new features of BASIC V. Major topics covered include:

- Using the colour palette
- WIRLE, 'p and CASE
- Use of mouse and pointer
- Local error handling
- Operators & string handling
- The Assembler

- Control structures
- Matrix operations
- Functions and procedures
- Sound
- Extended graphics commands
- Hints and tips

Mike Williams has been working with computers for over twenty years. For the past five, he has been editor of Beebug and USC User magazines, the 'after being the Largest arailation magazine devoted to the Archimedes.

Amstrad PCW

A Dabhand Guide by K John

lSBN 1-870336-50-X. Price £;14.95. Available Summer 1990.

The Arnatrad PCW9512 personal computer word processor and it's accompanying software, the LocoScript 2 system has revolutionised low-cost wordprocessing, and introduced a whole generation of people to compater-based word processing for the first time.

In this easy-to-follow guide, John explains how to use the program starting from first principles, with no prior knowledge assumed, either of the Ainstrad rcw system, the LocoSaipt program or even computers in general.

You are shown in pradical detail how to set the system up to yom own preferences. and how to produce neatly laid out letters, reports, essays and so on.

Difficult subjects are not avoided, instead they are introduced in a painless and straightforward way. After you have read this book, you will without knowing it, become a perceptive and sagadous word processor user'

P. John Atherton has used an Ainstrad rcw machine for many years, and has trained dozens of beginners on the machine. He has used the most common questions and problems as the basis for many of the topics in this book.

Commodore Amiga

AmigaDOS: A Dabhand Guide by Mark Burgess

ISBN 1-870336-47-X. Price £;14.95. Available NOW.

This is a comprehensive guide to The Commodore Amiga, and it's disc operating system, covering releases 1.2 and 1.3 of AmigaDos Workbench. It provides a unique perspective on this poweritil system in a way which will be welcomed by the beginner and experienced user alike.

Rather than simply reiterating the Amiga manual, this book takes a genuinely different approach to understanding and using the Amiga and contains a wealth of practical hand-on advice and hints and tips.

Among the many features in this book are:

- PUn coverage of ArnigaDos functions
- Filing with and without the Worklkndi
- The Amiga's hierarchical filing system
- Paflinarnes and device names
- The Amiga's multitasking capabilities
- The AmigaDos screen editor
- AmigaDoscommands
- Batch processing
- Amiga Error code desaiphons
- Use of the RAM discs
- Using AmigaDos with C

Amiga 500 First Steps: A Dabhand MiniGuide by Clive Grace

ISBN 1-870336~64). Price £;tba. Available summer 1990.

This book is the perfect introductory guide to the Commodore Amiga 500. Its sole aim is to guide you through those first few months of ownership as an easy-to-read supplement to the Amiga User Guide and assurrung absolutely no prior knowledge.

Its practical easy going approach introduces the various software and hardware components of the Amiga 500 and describes in detail how to put the madyifle to best use.

The Introductory Discs contain a wide range of useful programs that are also fully covered. But this book goes beyond this and also describes the many software and hardware additions available to the Amiga owner, and how to choose and install them.

AmigaBASIC: A Dabhmd Guide by Pani Fellows

ISBN 1-870336-87-9. Price £14.95. Available spring 1990.

AmigaBASIC: *A Dabhand Guide* provides a fufly structured tutorial to using AmiganAsic on the whole

range of Commodore Amiga computers.

Practical application is one of the many themes running through the pages and *as* such the many varied programs contained in its pages are both useful, and informative in programming technique. You are assumed to have a grounding of the way in which your Amiga works but no prior knowledge of BAS[C itself is necessary. A general theme of graphics is applied to the many examples throughout the book *so* that the techniques described are visually reinforced.

General

C: A Dabhand Guide by Mark Burgess

ISBN 1-870336-16-X. Price £14.95. Discs £7.95-9.95 inc. VAT.

Available NOW.

This is the most comprehensive introductory guide to C yet written, giving clear, comprehensive explanations of this important programming language.

The book is packed with example programs, making use of all C's facilities. Unique diagrams and illustrations help you visualise programs and to think in C.

Assembling only a rudimentary knowledge of computing in a language such as C or Pascal, you are provided with a grounding in how to build up programs in a clear and efficient way.

The differences between various compilers are acknowledged and sections on the popular compilers for the Amsfrad/[BM PC, Acorn machines including BBC and Archimedes, Atari ST and Commodore

Amiga are included, with notes concerning the ANSI and Kernighan and Ritchie standard.

Features of the book include:

- Compatible with all popular ANSI and K&R compilers
- Sections for rcs, Atari, Amiga and Acorn
- Diagrams to help you thiflic inC
- Arrays and string handling
- Data structures
- Mathematical programming
- Recursion
- fliscs available for many machines

Mark Burgess write computer programs in many languages of which C is his favourite. He is an honours graduate in Theoretical Physics.

"I wish this book had been available when I was learning C" *Personal Computer World. "...will give even relatively inexperienced programmers a clear* understanding *of* programming in *C." Elektor Magazine (December 1988).*

Software

Dabs *Press* publish a range of software for the Acorn Archimedes and BBC Micro, including products as diverse as language compilers for BASIC and Pascal, and computer games. For a free catalogue of software, please contact us.

Master 512 Shareware Collections

Price £29.95 per collection (ind. VAT and p&p)

Quite a lot of PcCornpatible software does not run on the Master 512, but Dabs Press have scoured through vast quantities of shueware products to produce these two collections of five 800K discs (equivalent to eleven or twelve 360K discs), all of which has been tried and tested on the Master 512.

Volume I includes MindReader, a powerful word processesor with the amazing 'word-antidpate' feature, As Easy As, a full-function LoRis compatible spreadsheet, with rnicros and graphics~ floflraw, a fimt class flowchart design utility, Fo~ind, a writing style anylisen a bridg~p1aying game, a colour chess game, an aerial pIan~flying action game, and many other arcade and board games. For Epson compatible printers there is a letter quality text printer, a downloadable font designer, and a utility to send printer codes directly from DOS. Your children will love the easy to use 'Kids Wordprocessor' with a simple command menu, and text characters an inch high. The world map program allows you to select any part of the globe and zoom in, with latitude and longitude co-ordinates, or by capital city.

As a result of the tremendous success of the first volume, *Volume* 2 has been released contabdng more tried and tested shareware and public domain software for the 512. This colection includes ExpressPile, a full function database manager with input form designer, report generator, and label printer; ExpressCalc - a powerful spreadsheet linking to ExpressFile, EZFL an exellent forms designer allowing simple aeation of forms using all the lines and boxes available in the IBM character set~ EQ an equation calculator wich works out mathematical, scientific and financial equations.

The packages also includes a manual which contains full details of any patticular points on usmg the software with a Master 512, and also plenty of general hints about the differences between a 512 and other Pc<ompatibles.

HyperDriver

Hyperdriver is an easy to use printer driver ROM that is independent of the application environment meaning that it can be used regardless of the language, program or word processor the you an using. It is hilly compatible with Epson compatible, matrix and laser printers, and all BBC and Master micros including 6502/Turbo second processors. Hyperdriver also works with the VIEW family (via a unique inbuilt printer driver). The pack contains a 16K EPROM for permanent internal fitting into the miac and a sideways RAM image on disc.

MiniDriver

MiniDriver provides real printer power for users of Mini Office II. Its many features include 50 easy * commands, on screen preview of effects, test print option, NLQ printer driver, and Teletext screen dump The pack contains a 16K EPROM for pennentant internal fitting into the micro or as a sideways RAM image on disc, additional hints and tips for use,and a 40 page manual are also included.

MOS Plus

This is a ROM that improves and extends the Master 128. It is not a utility ROM and its not meant for programmers, however it has been designed for the day to day user of the Master 128, providing useful commands which fall into the following five categories:

- Some of the bugs in the Master are fixed
- Extra commands introduced by Acorn with the Compact are included
- Some completely new commands are provided
- The Master alarm clock is implemented
- An on line help facility giving information on View, BASIC, Terminal and MOS Plus

A major problem with the Master has always been that the DFS will not close files properly. Many software packages don't work with DFS because of this, including several major and important items for which there is no substitute. MOS Plus axes the bug, possibly freeing a socket in the machine.

Compact improvements. Although the Master Compact has inferior hardware to the Master 128, it has a superior operating system, because it has benefited from some of the further time spent on development. MOS Plus brings your Master 128 operating system up to Compact standard. *FORMAT, *VERIFY, and *BACKUP are in ROM,

*SHOW now shows the contents of all function keys, *sRLOAD allows you to use '1' suffix to initialise a ROM image after loading it.

*BUILD/*APPEND take top-bit-set characters. *CONFIGURE and *STATUS now list in alphabetical order. *ROMS now shows RAM where sideways RAM is present. Also *BACKUP and *COMPACT

now use sideways RAM meaning less swaps for single drive users, and of course you will now be able to *BACKUP without corrupting your current program or data.

Obtaining Dabs Press Books and Software

You can obtain Dabs Press books and software from any good bookshop or computer dealer, or in case of difficulty directly from us, post free.

Orders can be sent by post and payment can be made by cheque (drawn on a UK bank), postal order, credit card (quote number and expiry date), or official order (education/public sector/PLCs only).

Telephone or fax orders can be made with a credit card - this is the simplest and most popular method.

Our address, telephone number and fax number are on part 2 of this book.

Index

It is the authors firm conviction that a good index does not compensate for a poor contents list. In this book therefore, the contents list is very detailed, with the intention that references to particular functions or operations can be made directly from it.

In the following index, therefore, references are limited to significant occurrences of words. For example no general references will be found for words such as 1)08, 512, MOS and so on since these words probably occur on half of the pages in the book. Likewise, words which are not specific, such as disc-sector, disc drive etc. are ornitted.

Notes