MasterDOS

For The SAM Coupé

User Manual

SAM Computers Limited

********** MasterDOS FOR THE SAM COUPE **********

- CONTENTS -

If you have any suggestions, or problems with this program, you can contact SAMCO, or alternatively, write to:

Dr. Andrew Wright, 24 Wyche Ave, Kings Heath, BIRMINGHAM B14 6LQ

This program took me a lot of time and effort to write, and the price is very reasonable. Please let your friends buy their own copy, so that I (and SAMCO) can continue to work on new products!

STARTING TO USE MasterDOS

To be on the safe side, ensure your MasterDOS disk is write-protected so you do not accidentally erase it. You should be able to see through the little hole in one corner; if you can't, move the small plastic slider over.

To load MasterDOS, turn on the computer and place the MasterDOS disk in drive 1, which is the left-hand slot. The label should be uppermost and the metal part should be on the side away from you. Then press the F9 key and the light on the disk drive will go on and the Disk Operating System should load. The computer will now understand a range of new commands that deal with disk operations.

You may be aware that the Coupe normally LOADS and SAVES to tape if DOS has not been loaded. Pressing the F9 key loads the DOS using the BOOT keyword, and this selects DEVICE dl, which means that SAVE, LOAD, MERGE and VERIFY now automatically use disk drive 1 rather than a tape recorder. This means that Basic programs that worked with tape usually do not need to be altered at all when transferred to a disk. To go back to using tape, you can include "t:" in the file name. For example, SAVE "t:test" or SAVE "t45:faster" or LOAD "t:test". There is a special case when you do not have to do this: LOAD "" ALWAYS uses tape, because loading a file with no name makes no sense to a disk system. Keys F7 and F8 are programmed to be LOAD "" and LOAD ""CODE and they will always work with tape. You can also type: DEVICE t to go back to using tape for all SAVES and LOADS.

Copying your MasterDOS disk

The first thing you should do is make a copy of your Master DOS disk. To do this, press the button on the disk drive and remove the MasterDOS disk, and replace it with a blank disk. Then type: FORMAT "d1" and press the <RETURN> key. (From now on whenever you have a command to type in I will assume that you will press <RETURN> afterwards.) The computer will format all the tracks, and then verify that they are O.K. If this does not happen, try again, perhaps with a different disk. When the FORMAT is complete, take out the disk and put the MasterDOS disk back in. Type: BACKUP "d1" TO "d1". All the files will be read from the disk. When "Insert target disk press a key" appears, insert the disk you formatted earlier and press a key. If the "0 OK, 0:1" message appears, the copy is complete.

(This will be what happens, unless many extra files have been placed on the disk. Otherwise, you will be prompted to "Insert source disk" again. Follow the prompts until the copy is finished.)

Now put the original MasterDOS disk away in a safe place and use only the copy.

You must FORMAT any new disks before you use them. It is not necessary for each disk to have a copy of MasterDOS on it, but it is convenient. The DOS must be the first file on the disk, or it will not be found, so it is convenient to copy it on just after doing a FORMAT. You can use COPY "mdos1" TO "*" to copy just the DOS from a disk with it on, to another disk - just follow the prompts.

RAM DISKS - Why you have at least six disk drives.

A RAM Disk is a section of computer memory that acts like a disk drive. RAM Disks are explained in detail in a later section, but I mention them here so that you know that you have at least six "drives" on your Coupe, even if you thought you only had one! You can usually use two-drive forms of commands explained later on in this manual - just note that a RAM disk loses its contents when you switch the computer off!

EXPLAINING THE DIRECTORY

Put your new MasterDOS disk into drive 1 and type: DIR. DIR displays on the screen the DIRectory of the disk drive which is your current DEVICE. You'll see a screen similar to this:

```
MasterDOS 1:

MDOS1     memuse    progl
prog2     prog3     rafprog

Number of Free K-Bytes = 758
6 Files, 74 Free Slots
```

This is a simple directory which gives only the names of files on the disk and some information about the disk itself. The file names are sorted into alphabetical order. (More exactly, sorted according to the CODE of each character. "1" comes before "A" which comes before "a".) For more detailed information, type: DIR 1 (or DIR 2 for drive 2). You will see something like this:

```
MasterDOS 1:

   1 MDOS131 C 65536,15700
   2 progl2 BASIC
   3 prog23 BASIC
   4 prog33 BASIC
   5 rafprog 2 BASIC
   6 memuse  3 BASIC1

Number of Free K-Bytes = 758
6 Files, 74 Free Slots
```

Don't worry if the screen doesn't look exactly like this.

The disk name is printed at the top left of the directory. Here it is "MasterDOS". To the right of the disk name is "1:" which tells us that the directory is for drive 1. After this follows a long list of names. Each named item is a file. Files can be BASIC programs, machine code programs, screen pictures or array files. At the end of the list some information about the disk is given: the space still available, the number of files present and the number of "Free Slots" for extra file names in the directory. In the example the disk can hold as many as 80 files, provided there's still space left on the disk. Later you'll find out how to prepare disks able to hold hundreds of files. (See FORMAT.)

In the list of files, each entry starts with a file number. Whenever you SAVE a file, the Coupe will give it the first available file number. So if you have the DIRectory shown in the example, the next file you SAVE will automatically become program number 7. This program number will stay the same until the file is ERASEd. But if you ERASE, say, program 9, then the next file you SAVE to disk will become the new file number 9.

The second column shows the file name, which can be up to 10 characters long. Names can contain almost any characters, although it is probably best to start the name with a letter and continue with letters, numbers and perhaps spaces. Names to AVOID are "d1", "d2", "d3" etc. and names starting with numbers - these can be ambiguous.

Full stops have a special role in separating the main part of the name from an "extension", and the characters "?", "*", "/" and "\" also have a special purpose. Avoid using these characters in file names for now - their use will be discussed in detail later on.

The third column in a DIRectory listing shows the number of disk sectors used. Each sector holds 512 bytes (= 0.5 Kilobyte) so to find out the number of Kilobytes of disk space used for any file, divide the number in the third column by 2.

The fourth column is for the file type. These are the different types of file you are most likely to use:

```
BASIC     = BASIC program
C         = Code file
SCREEN$   = Screen file
D.ARRAY   = Numeric data array
$.ARRAY   = Character array
SNP 48K   = Snapshot file (a Spectrum memory copy)
OPENTYPE = Serial or random access file
DIR       = Subdirectory
```

You probably will not understand the significance of all these types of file yet, but don't worry, they will be explained later.

Files that are of CODE type will be followed by the start address and length of the code; advanced programmers will find this

handy. If a number follows the type BASIC, this is the line number that the program will auto-run from on LOADing.

If your directory is longer than the screen, you will be prompted to "Scroll?". When you have read the directory, press <RETURN> to see the next screen of directory entries. That's enough about directories for now - details of other forms will be given later.

Simple SAVING and LOADing

Place a formatted disk in your disk drive, then enter this short program which we'll use to illustrate the various disk operations:

```
10 REM circles
20 FOR r=1 TO 255 STEP 2
30 CIRCLE 128,77,r
40 NEXT r
```

To SAVE the program to drive 1 with the name "circles", type: SAVE "circles".

The maximum number of characters in a file name is 10. Capitals are equivalent to lower-case letters, so "circles" is the same as "CIRcles". Spaces within the name matter, so "file 1" is different from "file1".

Now check that the file has been correctly SAVEd by typing: VERIFY "circles" and you should get an OK message. (VERIFY compares the program in the computer with the program on the disk - so if you edit the program and then VERIFY again you will get "Verify failed".)

Now, clear the program in the computer by typing NEW, then LOAD the program back from the disk with: LOAD "circles". When the OK message appears, the program has been loaded. Press <RETURN> to list it and confirm this.

It is also possible to make the program automatically run when it is LOADed. Let's do that now with: SAVE "circles" LINE 10

But of course there's already a program called "circles" on the disk. The Coupe tells you this and asks you whether you wish to overwrite the existing file. Press the Y key for Yes or the N key for No. There is no need to press <RETURN> afterwards. Any key apart from Y is accepted as No by the computer, to minimise accidents.

Type DIR 1 to see "circles" in the directory. Notice the auto-running line number after "BASIC". (Actually, it is an auto-GO TO line number, but everyone calls it "auto-run"!)

To SAVE and LOAD from drive 2, you can type: DEVICE d2 and then type exactly what you did for drive 1, or you can use file names preceded by the drive number, for example: "d2:circles". The

"d2:" is lopped off the file name before it is used to SAVE or LOAD the file, so it is not counted as part of the 10 characters you are allowed for the file name proper.

SHORT-CUTS TO LOADING

After getting a detailed directory, you can LOAD a program simply by using its number from the left-hand column - e.g. LOAD 3. You do not even need a space after LOAD. This works with all file types except array files. The drive the file loads from will be the last one used, so: DIR 2: LOAD 12 will load the twelfth file in drive 2's directory.

This form of LOAD is particularly convenient if you want to load SCREEN$ or CODE files - you still use just a number.

AUTO Files and BOOT

When you press the F9 key, a special keyword, BOOT, is entered and this causes the DOS to load. BOOT also looks for any file starting with "auto" and loads it, if found. If you SAVE a Basic program using: SAVE "auto" LINE 1 the program will be automatically loaded and run, immediately after DOS has loaded.

If you press F9 again, or use the BOOT keyword, AFTER the DOS has loaded, the "auto" file will be loaded again, but the DOS-loading step is omitted. If you type: BOOT 1, the DOS will be loaded without the "auto" file being loaded.

Other forms of SAVE and LOAD

You will find quite a lot of information about other forms of SAVE and LOAD in the Coupe's manual, but I will give a brief summary of them here.

```
SAVE "name" LINE N      - SAVE BASIC file "name" and GOTO line N
SAVE "name" SCREEN$     - SAVE the  currentscreen image including
                          its PALETTE colours.
SAVE "name" CODE a,b    - SAVE CODE file "name" from address A,
                          with length B.
SAVE "name" CODE a,b,c  - As above, but run the machine  code
                          from address C on reLOADing.
SAVE "name" DATA xyz$   - SAVE string or string array xyz$ as
                          "name".
SAVE "name" DATA xyz()  - SAVE number array xyz() as "name".
```

All forms of SAVE can be of the form SAVE OVER "name". In that case, any existing file with the same name will be overwritten without asking. This applies even if the file is PROTECTed.

```
LOAD "name" LINE N      - LOAD BASIC file "name" and GOTO line N
                          Ignore any auto-running line.
```

```
  LOAD "name" SCREEN$     - LOAD a screen image file, or LOAD a
                            CODE file to the screen. A SCREEN$
                            file will automatically set the
                            correct screen MODE.
  LOAD "name" CODE        - LOAD CODE file "name" to the address
                            it was SAVEd from (see the directory).
                            Also works with SCREEN$ type files.
  LOAD "name" CODE a      - As above, but LOAD to address A.

  LOAD "name" DATA xyz$   - LOAD string orstring array file "name"
                            and call it xyz$.
  LOAD "name" DATA xyz()  - LOAD number array "name" and call it
                            xyz().
```

VERIFY and MERGE

Disks are quite reliable and VERIFY should almost always work.

```
  VERIFY "name"           - Compare BASIC file with program in the
                            computer.
  VERIFY "name" CODE      - Compare CODE file with the same area in
                            memory.
  VERIFY "name" CODE a    - Compare CODE file with the memory area
                            at address A.
```

VERIFY also works with array files and SCREEN$ files, although for the latter you will have to use the command within a program, because your typing will alter the screen!

```
  MERGE "name"            - MERGE BASIC file "name" and its
                            variables with the current program. It
                            may take some time if either program is
                            large.
  MERGE "name" CODE       - Like LOAD "name" CODE but stops any
                            auto-running.
```

You cannot use MERGE with other file types.

Using the NETWORK

You can SAVE or LOAD over the Network by including "n:" before the file name, for example: LOAD "n:" on the receiving computer and then type SAVE "n:testing" on the sending computer. The file name "testing" will appear on the screen as with a tape LOAD, but transfer is much quicker. You can easily pass messages to another computer by saving and loading string array files.

BACKUP

The BACKUP command allows you to copy an entire disk at once. The disk you copy to (the "target" disk) must be already formatted. Any files on that disk will be lost. You can copy from drive 1 to drive 2, drive 2 to drive 1, drive 1 to 3 etc. or you can use a single drive. For example:

```
BACKUP "dl" TO "d2"     - copy from drive 1 to drive 2
BACKUP "dl" TO "dl"     - single-drive copy
```

Reading from the source disk starts at once. Used areas of the disk are read into the computer's free memory until it is full, or everything on the disk has been read. Then you are prompted to insert the target disk, if you are using a single drive, and the information is written to the target disk. Often BACKUP will finish at this point, especially if you have a 512K computer, but if necessary you will be prompted to insert the source disk again and reading and writing will continue until all the information has been copied. If anything goes wrong before BACKUP finishes, do not use the copied disk even if DIR seems to show all the files are present.

You can minimise the number of disk swaps required for a single-drive BACKUP by freeing as much memory as possible in the computer. You could try: CLEAR 32767: OPEN TO 1 (this frees 48K normally used by BASIC) or, if you have used more than one screen: SCREEN 1: FOR s=2 TO 16: CLOSE SCREEN s: NEXT s. This will free 32K for each screen that was open. RAM disks can be deleted by using the FORMAT command - e.g. FORMAT "d3",0

Naming the target disk

The names used in BACKUP can be longer than "dl". For example:

```
BACKUP "dl:ignored" TO "d1:FILES 5"
```

Apart from determining the source drive, the first name is ignored, but the second name is used to name the target disk. In the example, the disk would be called "FILES 5". The name would appear at the top of a directory listing. If you use just "dl" or "d2", the default name of "MasterDOS" is used. RENAME can also be used to name a disk.

BACKUP marks each target disk with a different random number, so that MASTER DOS can tell when you change a disk - even if you change to a BACKUP of the disk you are using.

COPY

The COPY command can be used to copy one or many files. To make a copy of a file called "Test", which will be called "Testcopy", for example, you would type:

```
COPY "Test" TO "Testcopy"
```

The file "Test" will be loaded into a temporary storage area in the computer, and you will briefly see the message "LOADING Test". Then the message "Insert target disk press a key" will appear. To make a new copy of the file on the same disk, simply press a key now. Or, if you want your copy on a different disk, insert that now and press a key. (Don't forget that your disk must be formatted.)

If by any chance there is already a file called "Testcopy" on the target disk, you'll be asked whether you wish to overwrite it. Press Y for Yes or N to cancel the copy.

As the copy is saved, you will see: "SAVING Testcopy", and then you'll see: "Insert source disk press a key". The Coupe is checking whether there are more files to be copied. (You'll see why when you come to "Wild Cards" in a moment.) So re-insert your original disk and press a key. If there are no other files to copy, you'll see an OK message.

Computer abuse is a growing social problem. A common cause of violence is when the computer deliberately corrupts or loses a vital file. Don't believe that it can't happen to you. It will! Even the SAM Coupe can be temperamental. So make it a habit to back-up key files. Use a different disk but keep the same filename to make life simpler.

Large files may need to be copied in several sections, since only free memory pages can be used. Follow the prompts, if required. If you are using two drives, you can also copy a file from one drive to the other. Here are some examples:

```
COPY "Test" TO "d2:Test"
COPY "dl:name" TO "dl:name"
COPY "dl:xyz" TO "d3:arghhhh"
COPY "name" TO "d2"
```

The last example used "d2" to mean "copy to drive 2 and use the original name" which is certainly convenient, and explains why you shouldn't try to call a file "dl" or "d2"!

With disk commands in general, if you use "di" or "di:" or "di:*" after the TO keyword, the meaning is "drive 1, use the original name", and similarly for the other drives.

If your machine has just a single real disk drive you may find COPY "dl" TO "d1" entails an irritating number of disk swaps, and the BACKUP command may be easier to use. (You can always selectively ERASE after a BACKUP.) However, you can also set up a RAM disk as a temporary store:

```
10 FORMAT "d3",1,24: REM enough for 20 files, 115K
20 COPY "dl" TO "d3"
30 PRINT "Insert target disk and press any key"
40 PAUSE
50 COPY "d3" TO "dl"
60 FORMAT "d3",0
```

If you have a 512K machine or an extension RAM you will be able to set up a larger RAM disk. Also, if you don't mind doing a little programming, the routine above could be extended to cope with large

numbers of files in several "swaps". The new functions DIR$, FSTAT and DSTAT may be useful here.

At this point we need to explain a number of features that apply to many of the disk commands - these are Wild Cards, file-name extensions, and the "ASK ME" option.

WILD CARDS

These are special symbols in file names that can be equivalent to any character or series of characters. (They are called after special cards in some games that can be anything you want them to be.) They apply not just to COPY but also to DIR, ERASE, RENAME, HIDE, etc. The asterisk (*) will match any series of characters, so: COPY "*" TO "d2" will accept any and all files on the current drive for copying (with the same name) to drive 2. "dl:*" or "d4:*" mean "everything on this drive" For convenience, "dl" or "d4" could have been used instead, but the asterisk is more flexible; if you want to copy all files starting with "n", use: COPY "n*" TO "d2".

Whereas "*" stands for any character or sequence of characters, "?" stands for any individual character, so: COPY "a??cs" TO "d3" will copy all files called a--cs. The second and third characters are irrelevant. You can also use wild cards in the name after TO. An asterisk means "take all following characters from the source name", a question mark means "take this character from the source name" and anything else means "use mel ignore the source name". So: COPY "m*" TO "X???two" will copy "mrt" to "Xrt two" and "mrt2" to "Xrt2two". All this is a bit mind-boggling until you are used to it - and even afterwards) But it can be very useful on occasion.

FILE-NAME EXTENSIONS

You already know that file names can be up to 10 characters long, including spaces, but excluding any device specifier like "d2:" or "t50:". But to help you organise your files, the Coupe allows you to use file-name extensions, in much the same way as a business computer.

For example, you might tag all your letter files with the extension ".let", all your database files with ".dat", and all your graphics files with ".gra". Your directory might show:

```
bank   .dat   bank   .gra   bank   .let
bank2  .dat   invite .let   lettrl .let
lettr2 .let   stamps .dat   stamps .gra
```

File names must still be less than 10 characters, including the decimal point. Notice the way the computer lines up all the extensions neatly. This is neat, but it does mean that a file called "x.12345678" will appear as "x .123", although its name is still "x.12345678" and you need to LOAD it as such. Remember, the name is what you entered, not what is displayed. It is possible to turn off the extension-alignment feature if you want - see DOS Variables.

The wild card "*" will work separately for the first part of an extended name, and the extension itself, so that you can use for example: COPY "dl:*.dat" TO "d2" to copy all data files, or ERASE "bank.*" to erase "bank.dat", "bank.let" and "bank.gra". DIR "*.let" will show just letter files.

The "ASK ME" Option

Another way to select some but not all files is to get the computer to ask you before each operation. This option works with ERASE, RENAME, PROTECT and HIDE, as well as COPY. You just follow the command with a question mark to tell DOS that you want it to ask you about each file. (This is distinct from the use of the question. mark WITHIN file names as a wild card character.) For example:

```
COPY "dl" TO "di"?
```

will ask:

```
COPY "file name " (y/n/a/e)
```

for each file. You can reply by pressing "y" if you want to COPY the file or "n" to skip it and proceed to the next one. Pressing "a" will COPY the file and All the rest, without asking further. Pressing "e" will Exit from the COPY command. Any other keypress is equivalent to "n".

MORE ABOUT DIR

The DIR command has many different forms. So far you have encountered DIR, which gives a simple, sorted directory of the current drive, and DIR 1, DIR 2 etc. which give a detailed directory of a specific drive. Only the latter form does a CLS first. Here are examples of some other forms:

```
DIR *           - detailed directory of the current drive.
DIR 11          - simple directory of a specific drive.
DIR "*.let"     - detailed directory using wild cards.
DIR "asd??"!    - simple directory using wild cards.
DIR "d2:*.dat"  - detailed directory of a specific drive, using
                  wild cards.
DIR 2)"*.dat"   - another form of the above.
DIR DATE        - detailed directory with date of SAVE, if
                  applicable. Perhaps best viewed in MODE 3. See
                  CLOCK commands.
DIR DATE "k*"   - as above but selecting drive and/or files.

DIR "subl/*"    - directory of a named subdirectory. See SUB
                  DIRECTORIES.
DIR ?           - simple directory showing files whatever their
                  subdirectory. DIR 1?, DIR "name"? and DIR
                  "name"!? are also allowed.
```

You can control whether simple directories are alphabetically sorted or not, and how many columns are used across the screen. (Normally the number of columns is the maximum that will fit into the current screen WINDOW in the current MODE.) See DOS VARIABLES.

ERASE

To ERASE a file, type: ERASE "file name". As usual, you can include a drive number if you want to specify the drive, for example: ERASE "d2:fred". Also as usual, "d1" on its own means "everything on drive 1" and "*" means "everything on the current drive" so be VERY careful you mean ERASE "dl" if you type it!

ERASE with the "ASK ME" Option

suppose you have a disk with 30 files on it, 25 of which you want to erase. It would be rather a chore to type in the 25 names, but ERASE "*" or ERASE "dl" will delete ALL the files, so what can you do? The answer is to use:

```
ERASE "*"?
```

The queston mark tells DOS to ask you what to do before each ERASE. If the first file is called "letter2" you will see:

```
ERASE "letter2" (y/n/a/e)
```

at the bottom of the screen. You can reply by pressing "y" if you want to ERASE the file or "n" to keep the file and proceed to the next one. Pressing "a" will ERASE the file and All the rest, without asking further. Pressing "e" will Exit from the ERASE command. Any other keypress is equivalent to "n".

You can also use wild Cards, as in: ERASE "*.bas"?
A special form of ERASE is available that works only on subdirectories - see SUBDIRECTORIES.

PROTECT

You can protect an individual file from being ERASEd by typing:

```
PROTECT "file name"
```

A PROTECTed file will appear in a detailed catalogue with an asterisk in the left-hand column, instead of a file number.

Protection can be removed by:

```
PROTECT OFF "file name"
```

You can also use PROTECT and PROTECT OFF with wild cards and the "ASK ME" option, e.g.: PROTECT "d2:gr*"?

If you use the normal ERASE command on a protected file, or answer "y" to the "Overwrite?" prompt during SAVE, you will get a BEEP and a "PROTECTed File" message. If you are erasing multiple files using wild cards, you will get a BEEP for each protected file, and ERASE will go on to the next file. The final message will be "OK" if at least one unprotected file was found and erased - otherwise it will be "PROTECTed file" again.

When a file is protected, it is still possible to SAVE or ERASE over it using:

```
SAVE OVER "file name"
ERASE OVER "file name"
```

Also you can COPY the contents of a file (say "new file") OVER the contents of another file (say "old file") even if "old file" is protected, using:

```
COPY OVER "new file" TO "old file"
```

This will copy the contents of "new file" to "old file"; "old file" will keep its original name.

HIDE

You can hide files too. Simply type:

```
HIDE "file name"
```

and it will no longer appear in the directory. HIDE OFF "file name" does just what you'd expect. You can use wild cards and the "ASK ME" option, as usual. When you HIDE a file, it becomes both hidden and protected. Watch out! It can still be overwritten using SAVE OVER, ERASE OVER or COPY OVER. HIDE OFF will then leave the file still protected, but visible.

RENAME

The RENAME command can be used to change the name of a file or the name of a disk. Files can also be moved from one subdirectory to another by including a directory name as part of the file name. (See SUBDIRECTORIES.)

To RENAME a file, we use the form:

```
RENAME "old name" TO "new name"
```

Let's RENAME the "Circles" file as "Example 1":

```
RENAME "Circles" TO "Example 1"
```

Look at the DIRectory again, to confirm that the change has been made. If you had already had a file called Example 1, you would have seen a "File name used" report and the RENAME would not have been allowed.

To RENAME the disk in the current drive, we use the form:

```
RENAME TO "name"
```

Try this and then do a DIR. You will see that "name" appears at the top of the DIRectory, in the space reserved for the disk name. Naming disks is useful for reminding you of the sort of files you have put on the disk, and for matching DIRectory listings sent to a printer with the disks they came from. It is a good idea to write the disk name on the label, too.

RENAME with multiple files

You can use wild cards to RENAME many files at once. We can use, say, "*" as the first name to RENAME all files, or "SNAP*" to just RENAME Snapshot files. The second name needs a little thought, because obviously we cannot use e.g. "testing" or the DOS will try to name all the files with the same name! (Don't worry - it will tell you "File name used" if you do this by accident.) A more sensible example would be:

```
RENAME "SNAP*" TO "GAME???"
```

This would RENAME all Snapshot files, but the fifth, sixth and seventh characters (represented by the question marks in the second name) would be kept as they were.

You can also use the "ASK ME" option so you can choose which files are RENAMEd - just follow the second name with a question mark.

RENAME can be very useful when used with subdirectories - see SUBDIRECTORIES.

FORMAT

The FORMAT command prepares a disk by writing blank sectors to it and verifying that they can be read back. (If FORMAT fails repeatedly, the disk surface is probably damaged.) The simplest form of FORMAT is: FORMAT "d". This formats the disk in the default drive. FORMAT "dl" or "d2" format the disk in drive 1 or drive 2. You can specify how many tracks are to be used to store the disk directory, and thus how many files the disk will be able to hold. If you do not specify a value, a 4-track directory holding up to 80 files will be assumed. Here are some examples:

```
FORMAT "d1"
FORMAT "d1",4    Both allow up to 80 files and give 780K of disk
                 space free for files on an 800K disk.

FORMAT "d1",5    Allows up to 98 files and gives 775K free.
FORMAT "dl ",    10 Allows up to 198 files and gives 750K free.
FORMAT "d1",39   Allows up to 778 files and gives 605K free.
```

Each directory track hold 20 file entries, except for the fifth track which holds only 18 because part of it is reserved by the DOS. Thirty-nine tracks is the maximum you can allocate to the directory, and four tracks is the minimum, except for RAM disks, which allow you to use as little as 1 track. The disk format used by MasterDOS is slightly different from SAM DOS format; it allows data to be transferred to and from the disk 10% faster. (Discs formatted by MasterDOS can still be used by SAM DOS, provided the directory takes up 4 tracks.)

As well as writing blank sectors to the disk, MasterDOS writes some information about the disk into the first sector. This includes a 2-byte random number which allows disks to be told apart, and a disk name which will appear at the top of directory listings. If you do not specify a name, "MASTER DOS" will be used. To use another name, use for example:

```
FORMAT "NUMBER 6" or FORMAT "d2:ALAN"S".
```

RAM DISKS

Introduction

A RAM disk is a section of computer memory (RAM) that acts like a disk drive. (If it was up to me, I'd call it a RAM drive, but never mind.) Its main advantage is that it is much faster than a real disk drive. This compensates for the disadvantage that all files are lost when you turn the computer off! The usual method of working is to copy the files you are going to use most frequently from a real disk to the RAM disk. There they can be used just as though they were on a very fast real disk. At the end of a programming session, any files that have been altered are transferred back to a real disk before you turn the power off.

Another advantage of a RA14 disk is that you magically have an extra disk drive when you need one! If your machine only has a single disk drive many file-copying operations can involve tedious disk swapping, which can often be greatly reduced by setting up a RAM disk as a temporary store. In fact, you can have up to 5 RAM disks, each of which holds from 5K to 780K. The only limit is available memory; if you have an external memory add-on, this will be automatically used for RAM disks before any of the main memory is used.

RAM disks are numbered from 3 to 7, and in general they are used just like real drives, for example: DIR 3 or SAVE "d3:test" or: DEVICE d5: LOAD "xyz" or OPEN #4;"d3:serial". However, before use they have to be FORMATed, and this command does differ slightly when used for RAM disks.

Setting Up a RAM Disk

To FORMAT a RAM disk, use the form:

```
FORMAT "d3",D,T
```

D is the number of tracks you want to be dedicated to the DIRectory, and T is the total number of tracks you want on the disk. As with a real disk, each DIRectory track can store 20 file entries. The other tracks can store 5K of data per track. You must have at least 1 directory track and 1 additional track, so the smallest RAM disk is obtained by: FORMAT "d3",1,2. Actually the DOS allocates whole 16K pages to a RAM disk as required, so you might as well use FORMAT "d3",1,3 because both require just one page. Each page can hold 3.1 tracks.

The FPAGES Function

Often, you will want a RAM disk to be as big as possible. How big is that? A 256K Coupe normally has 9 free 16K pages, while a 512K machine has 25. Each external memory module adds 64 more pages. (Of the reserved pages, the normal allocation is 4 for Basic, 2 for the screen, and 1 for DOS.) However, the number of free pages will change as SCREENS or RAM disks are set up, or utilities are loaded. Available pages will increase if Basic's allocation is reduced by e.g. CLEAR 32767: OPEN TO 1. The easiest way to check is to use the function FPAGES. Try PRINT FPAGES now. It is not a good idea to use every page for RAM disks, because COPY and BACKUP need at least 1 page temporarily when working. To use all except 1 page for RAM disk 3, you could use:

```
10 LET tks=INT((FPAGES-1)*3.1)
20 FORMAT "d3",1,tks
```

This will usually give a 24-track RAM disk on a 256K machine, or 74 tracks on a 512K. This will hold 115K or 365K of data respectively. By reducing Basic's memory use you can stretch this to 34 tracks (165K) or 83 tracks (410k). With an external memory unit you will need to ensure the number of tracks is 160 or less, because the capacity of each RAM disk can be no greater than that of a real 80 track double-sided disk. Also, the number of data tracks must be 156 or less. It is convenient to use FORMAT "d3",4,160 because that is the usual real disk format.

You can change the size of a RAM disk by using FORMAT again. Any files on the RAM disk will be lost. To release all the memory used, use FORMAT "d3",0.

Files can be transferred to a RAM disk simply by LOADing them from a real disk and then using SAVE "d3:name", but COPY is probably easier: COPY "dl" TO "d3" will copy all files to RAM disk 3, if there is room. Wild cards can be used so that only a selection of files is transferred. Another method is to use BACKUP "dl" TO "d3", which will work provided all used sectors in the source disk are available on the RAM disk; otherwise, you will get an error message.

Once you have a set of files on your RAM disk, you can try to use them as though they were on drive 1. You could DIR 3 and then LOAD a file by number, or use LOAD "d3:name", or DEVICE d3: LOAD "name". A problem may well occur, though) the program you load may include a line like DIR 1 or LOAD "dl:udgs" CODE which goes back to using slow old drive 11 What we would really like is to tell the DOS that it should use disk 3 whenever disk 1 is mentioned - and we can do just that.

Master DOS keeps a table of which drive it should actually use when a drive called for. Normally this holds the numbers 1 to 7, so that e.g. DIR 1 looks at table position 1, reads 1 and uses drive 1. But if the table is altered to hold 3 at that position, drive 3 will be used instead. (Or you can "swap" drives 1 and 2 by making the table hold 2,1,3,4,5,6,71) The table is stored at DVAR 111 to 117, so transfer your files to disk 3, then POKE DVAR 111,3. Now drive 3 acts as though it is drive 1. Even the F9 key loads AUTO files from drive 3. To return to normal, POKE DVAR 111,1.

SUHDIRECTORIES

Once you have been using a disk-based computer for a while, you will probably find that you tend to accumulate disks with a large number of fairly short programs on them. Even the 80 files allowed in the smallest directory size take some looking through, and if there are hundreds of files in one directory you have a problem. of course you can use lots of disks, but this is wasteful and means you are always looking for the right disk. The solution is to separate the files into subdirectories. Creating a subdirectory is almost like creating a new disk within a disk. To see what I mean, take a new disk and save some files onto it. Then create a subdirectory like this:

```
OPEN DIR "subl"
```

Of course the name can be anything you like, as with a file name. If you do a DIR 1 now you will see "subs" in the list of files, with the type DIR showing that it is the name of a subdirectory. We can make "subl" the current directory by typing:

```
DIR-"subi"
```

Now do a DIR and you will see something like this at the top of the screen: MasterDOS 1:\subl. The "subl" shows the subdirectory name. You will see that there are no files in the directory yet, so save a few, in the normal way. The free slots and free disk space will decrease, but note that this subdirectory can expand to fill all the free slots and space on the disk, if need be. Now make a sub-subdirectory by, say, OPEN DIR "games". Then use DIR="games" to make it the current directory. DIR now shows the subdirectory name as 1:\subl\games, meaning that "games" is a subdirectory of "subl", which is itself a subdirectory of the main, normal directory. We

can go on creating directories inside directories until all the free slots are used up, or until 254

subdirectories have been created - quite a while! Any directory can have multiple subdirectories in it.

The main directory is usually called the "root", because one can think of the arrangement of subdirectories as like a tree, with the root directory branching into subdirectories which branch further until you end up with "twigs" with no further branches - just "leaves" or files. (One problem with the analogy is that we tend to talk about "going deeper" as one goes into subdirectories further from the root - so the tree must be growing upside down!)

Let's go back to the root now. There are several ways to do this. We can go back up the way we came, using DIR="" (SYMBOL SHIFTH), which will take us up one level to "subl", and then the same again to get back to the root. Or, we could do it in one go using DIR="\", which means "make the root the current directory" and which will take us back from wherever we are in the "tree". These two forms are used quite often, and to make typing easy, you can omit the quotes: DIR=/ will work. You might have noticed that the direction of the slash altered; MasterDOS is happy to accept either form.

Once back in the root, we can experiment with some different forms of SAVE and LOAD. Try: SAVE "subl/testfilel". If you go back to directory "subl" (DIR="subl") and do a DIR, you will find that the file has been saved here. We could also have SAVEd to a sub-subdirectory using e.g.: SAVE "subs/games/file-name". The string that tells SAVE where to put the file and what to call it has a special name - it is called a "path". Only the last part of the string is the actual file name: the rest determines the path along the directory "tree", and does not contribute to the 10character file name length limit.

Normally, a path starts from the current directory, but you can force it to start at the root directory if you like by making the first character of the path name a slash. For example: SAVE "\subdr2\fred" will save a file called "fred" in subdirectory "subdr2" which is a subdirectory of the root directory, whatever directory is our current directory. Without the first slash, "subdr2" would have to be a subdirectory of our current directory.

You can also use "" as the first character, which will go back up to the previous directory level before using the rest of the path name. This allows you to access a subdirectory which is a branch of the same directory that your current directory branches from, if you are so minded. (Usually, I'm not.)

DIR and Subdirectories

You can use e.g.: DIR "subs/*" to display all the files in a subdirectory. DIR ? will list all files in any subdirectory, as will DIR 1? The file numbers in the left hand column can always be used to LOAD the files, whatever directory you are in.

ERASE and Subdirectories

You can use ERASE as you might expect. For example: ERASE "subs/*" will erase all files in the subdirectory "subl", and ERASE "games\bad game" will erase "bad game" in the "games" subdirectory.

The normal form of ERASE ignores DIR-type files, but you can erase such a file using e.g.: ERASE DIR "subl". This will only work if the subdirectory is completely empty; if not, you will get a "Directory not empty" message.

COPY, PROTECT, HIDE and Subdirectories

By now you should have got the general idea, but here are some examples:

```
COPY "games/lotto" TO "d2:games/*"
PROTECT "/subl/imp.let"
HIDE "subl/st*"
```

RENAME and Subdirectories

RENAME is a quick way of moving files from one subdirectory to another. If you have a disk with lots of files on it, and you want to place some of them in a subdirectory, try this:

```
OPEN DIR "some name"
RENAME "*" TO "some name\*"?
```

If you press "y" when prompted, the file will be RENAMEd to the SAME name - but it will be in subdirectory "some name", and will vanish from the main directory. Try: DIR "some name*". You can use more complex path names, such as:

```
RENAME "\sub2\games\frogger" TO "\junk\hop"
```

The normal form of RENAME automatically avoids RENAMEing any subdirectory names it finds. Should you want to RENAME a subdirectory, there is a special form of the command which only deals with subdirectories. This is:

```
RENAME DIR "old name" TO "new name"
```

You can also use RENAME DIR with a path name. RENAME DIR "old name" TO "subdl/*" is equivalent, in our directory "tree" analogy, to cutting off a branch, complete with sub-branches and leaves, and grafting it back on somewhere else on the tree. The DIR file "old name" and all its own files will now be accessible only via subdirectory "subdl". Avoid using e.g. RENAME "fred" TO "fred/xyz" - this makes DIR file "fred" only available via subdirectory "fred".... This is like grafting a branch into an endless loop, not connected to the tree!

The PATH$ Function

This function returns the name of the current subdirectory, as given after the disk name in a directory. For example, if you are in the root directory: PRINT PATH$ will give "1:" or perhaps "2:" or "3:". If you type: DIR="subl" PATH$ will become "1:\subl", and complex examples like "2:\subl\games\chess\chdata" are possible. The maximum length attainable is 38 characters; anything more is chopped off (and not printed by DIR) although you can be at a directory level deeper than this might suggest.

USING THE CLOCK/CALENDAR

A Clock/Calendar with battery-backup is provided on SAMCO's multi-slot motherboard. The commands DATE and TIME and the functions DATE$ and TIME$ allow you to set and read it. Files are "date and time stamped" as they are saved.

DATE and DATE$

The DATE command has two purposes: it allows you to set the date on the Coupe's add-on clock, and it prints it for you. Even if you do not own an add-on clock the command can be useful. To set the date, follow the PATE command with a string containing six digits, for example: DATE "010391" will set the date to 1st. March 1991. You can include non-digit characters in the date string to make it more readable if you want - so "01/03/91" would be allowed. The day of the month must be 1-31, the month must be 1-12, and the year must be 0-99, or you will get an "Integer out of range" error report.

Typing DATE by itself will print the current date in the form dd/mm/yy, followed by a carriage return. For a more flexible method of using the current date in a program, you can use the DATE$ function, which returns an 8-character string containing the date. For example:

```
PRINT DATE$
PRINT "Month=";DATE$(4 TO 5)
```

You might enjoy writing a program to generate something like "3rd. July 1991" from a DATE$ of "03/07/91".

If you own the clock add-on, all the files that you create will be "date-stamped" and "time-stamped" with the current date and time, as held by the clock. This information can be displayed using the DIR DATE command (see DIR for more details). If you do not own the clock add-on, the date will start at "00/00/00" unless you set it. This odd value tells the DOS not to bother date-stamping or time-stamping files it creates. However, if you want you can set the date yourself every time you turn the computer on, in order to date-stamp your files. First, POKE DVAR 150,0 to tell DOS not to try to use the non-existent hardware. (The normal value of this DVAR is 239, the clock "port" address.)

TIME and TIME$

The TIME command has two purposes: it allows you to set the time on the Coupe's add-on clock, and it prints it for you. To set the time, follow the TIME command with a string containing up to six digits, for example: TIME "11:30". If you supply less than six digits, then zeros are assumed. As with DATE, separators can be used in the string, but they are not required. The hour must be 0-23 and minutes and seconds must be 0-59, or you will get an "Integer out of range" error report. A 24-hour clock is used, so e.g. 2:00 p.m. is entered as "14:00".

Typing TIME by itself will print the current time in the form hh:mm:ss, followed by a carriage return. For a more flexible method of using the current time in a program, you can use the TIME$ function, which returns an 8-character string containing the time. For example:

```
DO: PRINT AT 10,10)TIME$: LOOP
```

(It won't change unless you buy the motherboard and clock!)

DIRECTORY AND FILE FUNCTIONS

The DIR$ Function

This function returns all the file names in the current directory as a string. Each name takes up 10 characters in the string - short names are "padded" with spaces. The names occur in the order you would see in a detailed directory listing. PRINT DIR$ will give you a directory, but it is not as neat as typing DIR, so what is it for? The function is designed to be used within a program to aid file handling. Each file name can be obtained in turn by some thing like this:

```
10 LET list$=DIR$
20 FOR p=1 TO LEN list$ STEP 10
30  LET nm$=list$(p TO p+9)
40  REM...now do something with nm$
50 NEXT p
```

Making the screen window 10 characters wide can be handy:

```
10 WINDOW 0,9,0,18: CLS 1
20 PRINT DIR$: WINDOW
```

You can also use wild cards with DIR$. For example:

```
PRINT DIR$("let*")
```

The DSTAT(drive,N) Function

DSTAT is short for Disk STATus. This function can be used to tell you lots of useful things about the disk in any drive. For example:

PRINT DSTAT(1,1) gives the free space on the disk in drive 1.

You can specify a drive number between 1 and 7, because the function works with RAM disks as well as ordinary disks. You can also use an asterisk to mean the current drive, so that:

PRINT DSTAT(*,1) gives the free space on the current drive.

Here is a complete list of the information provided for different values of the second number (the N parameter):

1. Returns the amount of USABLE space on the disk in bytes. (Specifically, 510 times the number. of free sectors, less 9 byte; that might be required for a header when saving a file.) If there if: no space in the directory (i.e. if Free Slots--0) or the disk is write-protected you will get a value of zero, even if the disk has free space on it. Returns -1 if there is no disk in the drive, or a RAM disk has not been FORMATed, or the drive is not fitted. This is true of all the options.

2. Returns 1 if the disk is write-protected, and zero if the disk is present and not write-protected. Note: RAM disks are never write-protected.

3. Returns the amount of free space on the disk in bytes, like option 1, but without worrying about whether the disk is write-protected or has free space in the directory.

4. Returns the number of free slots for file names in the directory. 5. Returns the total number of files on the disk.

6. Returns the number of files in the current directory. This will be the same as option 5 unless you have created a sub directory (see SUBDIRECTORIES).

7. Returns the number of tracks used by the directory.

8. Returns the current drive number.

The FSTAT("file name",N) Function

FSTAT is short for File STATus. This function provides useful information on specified files. For example:

PRINT FSTAT("test",2) returns the length of the file "test".

Here is a complete list of the information provided for different values of the number N:

1. Returns the number of the specified file in the directory. This is the number you would see alongside the file in a detailed directory. If the file does not exist, zero is returned. This is useful for checking if a file exists before SAVEing or LOADing. All options return -1 if there is no disk in the drive.

2. Returns the file length in bytes. Some files have a 9-byte "header" which is not included in this length.

3. Returns the file type using the following codes:
   1=ZX BASIC
   2=ZX Numeric Array
   3=ZX String Array
   4=ZX CODE
   5=2X SNP 128K
   6=ZX Microdrive File
   7=2X SCREEN$ 8=SPECIAL
   9=ZX SNP 128K
   10=OPENTYPE
   11=2X EXECUTE
   16=SAM BASIC
   17=SAM Numeric Array
   18=SAM String Array
   19=SAM CODE
   20=SAM SCREEN$
   21=SAM SUHDIRECTORY

4. Returns the file type as above, but also adds 64 if the file is protected, and 128 if the file is hidden.

SERIAL FILES

Creating a Serial File

To create a serial file, you must first of all OPEN the file for output, specifing a stream number and a file name, for example:

```
10 OPEN #4;"testfile"OUT
```

The disk operating system will check to see if "testfile" already exists on the current drive, and ask you whether you want to overwrite the old copy if it finds one. You can specify a different drive as usual by using a name starting with "dl:", "d2:", "d3:" etc. Stream 4 will be assigned to the file, unless stream 4 is already in use. (You can use any stream number from 4 to 15.) From now on, PRINT commands qualified by "#4" will PRINT to the disk file, rather than the screen. For example:

```
20 FOR n=1 TO 50
30 PRINT n;" abcdefghi"
40 PRINT #4;n;" abcdefghi"
50 NEXT n
```

You can see what is being sent to the disk file because line 30 prints a copy to the screen. The file will contain 50 strings, from "1 abcdefghi" to "50 abcdefghi". In many applications these strings may be referred to as RECORDS.

If every single character had to be placed immediately onto the disk, writing to a file would be very slow. Instead, characters are accumulated in a special buffer in memory until there are enough of them to fill a whole disk sector. Then the disk drive is started up, if need be, and the whole sector is written in one go. Similar buffers are used when files are read. The Coupe stores 510 data characters in each sector. Two more bytes are used by the DOS, giving a total of 512 bytes per sector. The use of a buffer means that the program above is incomplete; when the FOR-NEXT loop finishes, the buffer will be only part-full, and if nothing is done the information in it will be lost forever. Therefore we need to CLOSE the file:

```
60 CLOSE #4
70 STOP
```

This writes the last buffer to the disk, and also creates an entry in the disk catalogue. The type is shown as "OPENTYPE".

READING A SERIAL FILE

Having created a serial file as described above, we will now read it. Again the file must be OPENed, but this time with the IN keyword. The stream number you specify in the OPEN statement can later be used to read data from the file using either INPUT # or INKEY$ # or INP$

```
100 OPEN #41"testfile"IN
110 INPUT #4;a$
120 PRINT a$
130 GO TO 110
```

The example above will show the file contents and then stop with an "End of file" report. To close the file, you should type:

```
CLOSE #4
```

Although closing an input file is not as vital as closing an output file, input files use memory for a buffer just as output files do, and this cannot be reused without a CLOSE. Besides, the stream needs to be closed if it is to be used again.

If you have opened a file with one disk in the drive, it is usually not a good idea to change to a new disk. INPUT from the file will be rubbish, and PRINTing to it is likely to corrupt

files on the new disk. It is possible to insert a different disk temporarily and use DIR or LOAD. Provided you use disks FORMATed using Master DOS, the Coupe will know that the disk has been changed and will keep reminding you by a BEEP and an "OPEN file" message at the bottom of the screen, every time you DIR, LOAD or SAVE with a different disk than the original. If you SAVE to a different disk, the use of available disk space will not be very efficient for either the SAVEd file or (after you have put the original disk in again!) later PRINTS to the OPENTYPE file.

You can abandon any OPENTYPE files (with possible data loss if you are writing to the files) using CLEAR #. MOVE (see later) uses temporary OPENTYPE files and you may need to use CLEAR # if a MOVE is interrupted part-way through. PRINT PEEK DVAR 20 will show you the number of open files if you are interested.

Each INPUT reads one of the strings originally PRINTed to the file, and you may wonder how this is done - in other words, how does the DOS know where a string ends? The answer is that each string is "terminated" by a special character, CHR$ 13, which is called "carriage return", a name dating from the days of teletypes. When you enter something like:

```
PRINT "one": PRINT "two"
```

the Coupe actually sends "o", "n", "e", CHR$ 13, "t","o", CHR$ 13 to a ROM routine that puts the normal letters on the screen but RESPONDS TO the CHR$ 13s by printing on the next line. Printing to a disk file is similar, but the CHR$ 13S are actually stored on the disk, instead of being responded to. When you come to INPUT from the file, the DOS reads characters from whatever point it has got to in the file until it finds a CHR$ 13. It then assigns these characters to the variablespecified in the INPUT command. (The CHR$ 13 itself is thrown away.)

The INKEY$ Function

INKEY$ can be used to read a disk file one character at a time. Unlike INKEY$ from the keyboard, INKEY$ from disk always gets a character. Try changing line 110 above to:

```
110 LET a$=INKEY$#4
```

I suggest you also add a new line:

```
90 CLOSE #4
```

This prevents "Stream used" errors and is often convenient. Now RUN 90 to try the program with the existing line 120, then with these variants:

```
 120 PRINT a$;
or 120 PRINT a$;" ";CODE a$
```

The last version will show up the CHR$ 13s explicitly. You can create other files that contain "control codes" (which cause actions) other than CHR$ 13. For example, the print comma which tabulates screen output sends CHR$ 6s to a disk file, and PEN, PAPER, TAB etc. send special character sequences.

INPUT LINE

If some of the strings being INPUTed contain quotes - for example, "Bide-a-Wee" as a quoted house name in an address (ugh!) you will need to use INPUT LINE, just as you would if the INPUT was from the keyboard. For example:

```
INPUT #47 LINE a$
```

More About Opening and Closing Files

It is possible to OPEN a file without specifying IN or OUT; in that case, the DOS assumes you mean OUT if the file does not already exist in the catalogue, and assumes you mean IN if it does. If a file is PROTECTed, the DOS will prevent you writing to the file by using IN, whatever you actually specify.

Although it is usual to use OPEN and IN or RND on OPENTYPE files, you can in fact OPEN any file type in this way, although the things you can use this for are fairly exotic. (Altering snapshots without loading them?) Some file types (Basic, CODE, array and SCREEN$) have a 9-byte header before the main part of the file.

As well as its use for closing a particular stream, CLOSE lets you to close ALL open files using the form:

```
CLOSE *
```

CLEAR # clears all open files, but without doing a CLOSE. This makes no difference to IN files, but OUT files will be lost. RND files will be lost if they are new files, or may be part-written otherwise.

MOVE

The MOVE command reads a file, a character at a time, and writes it to another file or a stream. It actually uses something very like successive INKEY$ #s and PRINT #s to do this, but the process is invisible to the user, and the streams and buffers associated with the disk files are OPENed and CLOSEd automatically by the DOS. For example:

```
MOVE "testfile" TO "copyfile"
```

"Testfile" must exist and "copyfile" should not, unless you want to overwrite it. This is a fairly slow method of copying a file and COPY "file" TO "file" is much faster for long files. However, MOVE is more flexible. For example, if you had two files called

"first" and "second" you could create a single longer file that contained copies of them both like this:

```
10 OPEN f5;"combined"OUT
20 MOVE "first" TO #5
30 MOVE "second" TO #5
40 CLOSE #5
```

You can also MOVE files to stream 2, which is the main part of the screen. As well as being an easy way of looking at an OPENTYPE file, this allows you to look at Basic programs without loading them, although they will not be quite as neatly listed as usual. It is also a good way of looking for text or data included in a CODE program. See DOS Variable 24 for details. You can also MOVE files to stream 3, which is the printer, and randomaccess files (see later).

RANDOM ACCESS FILES

Although serial files can be very useful they have the major disadvantage that to look at a particular item in a file you need to read all the previous items. To alter an item you have to read and rewrite the entire file. For some applications, this can be very inconvenient, and the problem gets worse as the file gets bigger. Master DOS provides RANDOM ACCESS filing. This means you can examine or change any part of a file without having to load the whole thing. The normal OPENTYPE files are used, but they are OPENed with the RND extension; e.g.:

```
OPEN #5;dl"testx"RND
OPEN #strm;d2"file name"RND
```

This allows you to write to the file, using PRINT #. as you could if you had used OUT, and read it using INPUT # or INKEY$ # or INP$, as you could if you had used IN. In fact you can change all the INS and OUTS in the examples for serial files to RNDS and the programs will work as before.

Creating a Simple Random-Access File

Although you can OPEN any existing OPENTYPE file using RND, explanations will be more straightforward if we create a simple test file, as shown below. Those who hate typing will be pleased to learn that the examples are included on their Master DOS disk. LOAD "progl" now.

```
10 CLOSE #4
20 OPEN #4;"test"RND
30 LET a$="abcdefghi"
40 FOR n=1 to 200
50 LET a$( TO 3)=STR$ n
60 PRINT AT 10,10;a$
```

```
70 PRINT #4;a$
80 NEXT n
90 CLOSE #4
100 STOP
```

Just RUN to create the file. I have used the first 3characters of each string to store a record number so that we can easily check which one we are looking at in our experiments. The file will consist of 200 strings or records, from "1 defghi" to "200defghi". Although the text of each string is 9 characters long, each one will take up 10 characters of disk space because they are terminated by carriage return characters (CHR$ 13s). These strings are an example of FIXED-LENGTH records, and the advantages of using this system will become apparent later.

In our example, it was easy to count the characters in a$ and know how long the records would be, but when you uselonger strings a better way is to use DIM. For example:

```
25 DIM a$(99)
```

would ensure each string was 99 characters long and took 100 bytes of disk space. All this should be plain sailing if you followed the discussion of serial files; if line 20 had ended in OUT everything would have worked exactly the same way. The disk file you have created is a normal OPENTYPE file.

Reading a Simple Random-Access File

Assuming you have an existing OPENTYPE file, you can OPEN it with the RND qualifier and handle the file just as if you had used IN, reading one record after another. As you do this, an internal FILE POINTER advances through the file as INPUT or INKEY$ are used. This points to the next character to be read. For example, if you INPUT a 10-character string, the pointer advances by 11, because of the CHR$ 13 terminator. (OUT files have a similar pointer which points to the next position to write to, which is always at the end of the file.)

The file pointer used by MasterDOS starts with a value of zero before the first character is read, so we can say that the first character is at position zero in the file. Just before we read the last character in the file, the pointer will have a value 1 less than the file length. The file can be thought of as a sequence of characters numbered from zero to (file length 1). When the last character has been read, the pointervalue will equal the file length. Any further reads will give an end-of-file error.

POINT

Use the form:

```
POINT #stream,position
```

The file pointer for the file associated with the specified stream is immediately moved to the specified position. (This will often cause a new sector to be loaded from disk.) Now try this, by using RUN 200:

```
200 CLOSE #4
210 OPEN #4;°test"RND
220 DO
230 INPUT "Record? ";rec
240 EXIT IF rec=0
250 POINT #4,(rec-1)*10
260 INPUT #4;a$
270 PRINT a$
280 LOOP
290 CLOSE #4
```

You should be able to see the advantages of having fixed-length records - the pointer value for any record can be easily obtained from the record number, allowing you to INPUT from any record in the file very quickly. If the record you want is in the current sector, it will be obtained particularly quickly, but even if it is at the other end of a 700K file, and the disk is stationary, the record can be obtained in 1 or 2 seconds. RAM disks will be almost instantaneous.

Note: POINT with a value less than zero will give an "integer out of range" error, and POINT with a value greater than the file length will give an "End of file" error.

Altering a Random-Access File

An OPENTYPE file OPENed in random-access mode can be written to as flexibly as it can be read. The file pointer indicates the position that data will be written to, as well as read from, so POINT allows any record to be altered. The program below demonstrates random reading and writing of our trusty "test" file. It leaves the record number intact to provide reassurance, although we always know which record we are dealing with simply from the file pointer value we set using POINT: record=file pointer/record length+1. (Later you will see how to read the value of the file pointer directly.) Now LOAD "prog2".

```
10 CLOSE #4
20 OPEN #4;"test"RND
30 DIM a$(9)
40 PRINT "Read, Write or Exit? (R/W/E)"
50 LET c$=INKEY$
60 IF c$="W" OR c$="w" THEN GO SUB 100
70 IF c$="R" OR c$="r" THEN GO SUB 300
80 IF c$<>"E" AND c$-"e" THEN GO TO 50
90 CLOSE #4: STOP
100 INPUT "Record to write? ";r 110 IF r=0 THE14 RETURN
120 POINT #4,(r-I)*10 130 INPUT #4;a$
140 PRINT "Old text:";a$ 150 PRINT "New text?"
160 INPUT n$
170 IF n$="" THEN GO TO 100
```

```
180 LET a$(4 TO )=n$
190 POINT #4,(r-1)*10
200 PRINT #4;a$
210 GO TO 100
300 INPUT "Record to read? ";r
310 IF r=0 THEN RETURN
320 POINT #4,(r-1)*10
330 INPUT #4;a$
340 PRINT a$ 350 GO TO 300
```

I suggest you play with the program, reading and writing records all over the file. (You might want to add a check to prevent the use of record numbers greater than the number of records in the file. And you might like to rewrite it completely, since I wrote it first in nasty Spectrum Basics) Enter a "record number" of 0 to stop writing or reading. (In fact the write subroutine at line 100 can serve for reading as well, since if you press RETURN when prompted for "New text?" the record will not be altered.) Notice that POINT is used at line 120 to set the file pointer for an INPUT, and then again at line 190 for a PRINT to the same record, because the INPUT will have moved the pointer. When you are finished press "E" to exit and CLOSE the file. The disk may or may not run, depending on whether you have altered the current sector or not.

OPENTYPE FILE FUNCTIONS

PTR - Reading the File Pointer

As well as being able to move the file pointer using POINT, Master DOS is able to read the current pointer position using the PTR function. (LOAD "rafprog" now.) PTR (stream) will tell
you the pointer position for an OPENTYPE file associated with the specified stream. For example:

```
10 OPEN #4;"test"RND
20 PRINT PTR 4;" ";
30 INPUT #4;a$: PRINT a$
40 GO TO 20
```

This function is most useful when a file contains variable length strings and the file pointer moves by different amounts with each INPUT. You can find out the position of a particular string and get back to it later using POINT.

The LENGTH Function - Finding out a file's length
Extending a file

Knowing a file's exact length is often useful. For example, you can extend an OPENTYPE file easily by setting the file pointer to the end of the file using POINT before you use PRINT # to add new data. To do this you need to know the length of the file, and MasterDOS lets you do this using the LENGTH function.

For example:

```
100 CLOSE #4
110 OPEN #4;"test"RND
120 POINT #4;LENGTH#4
130 FOR n=1 TO 10
140 PRINT #4;"extended "
150 PRINT LENGTH#4
160 NEXT n
170 CLOSE #4: STOP
```

This will add some data to the end of our much-used "test" file, printing the current file length as each string is added. If line 120 had been omitted, the first part of the file would have been overwritten by 10 "extended "s, because the file pointer would have started at zero. The file length would have stayed the same.

The file length is the maximum value you can use with POINT, so if you want to extend a file, even with blank records, you must POINT to the end and use PRINT # to increase the file size.

Now let's read the entire file to check that all is as we expect:

```
200 CLOSE #4
210 OPEN #4;"test"
220 DO
230 INPUT #4;a$
240 PRINT a$
250 LOOP
```

This finishes with an "End of file" report, which can be a problem when you are writing a real program. It is possible to end the file with a "rogue value" like "zzz" and use a line like: 250 LOOP UNTIL a$="zzz" but this is not very convenient. MasterDOS provides the function EOF to tell you when the End Of the File has been reached so that you can avoid further INPUTs - see below.

The EOF Function - End-Of-File detection

The function returns 0 if the last character in the specified stream has not been read yet, or 1 if it has. The example above could make use of a modified line 250:

```
250 LOOP UNTIL EOF 4
```

The INP$ Function - Reading a fixed number of characters

The INP$ function reads a specified number of characters from a stream. For example:

```
LET a$=INP$(#4,10)
```

Rather like INKEY$, INP$ does not care what characters it reads, and it will include any carriage returns it finds. The number of characters can be up to 16384, so you can often read a whole file at once, using e.g.:

```
LET a$=INP$(#4,LENGTH#4): PRINT a$
```

This is much faster than repeated use of INPUT or INKEY$.

DATA PACKING IN RANDOM-ACCESS FILES

Our "test" file is fairly small and simple. A real example would probably use longer records, with different parts of the records dedicated to particular purposes. (These areas are called "fields".) For example, if each record is created by PRINTing a 100-character string to the file, you might place data in a record as shown below. (This partial program is not on your disk.)

```
100 DIM d$(100)
110 INPUT "Author? ";a$
120 LET d$( TO 40)=a$
130 INPUT "Title? ";t$
140 LET d$(41 TO 96)=t$
150 INPUT "Year of publication? ";y$
160 LET d$(97 TO 100)=y$
170 PRINT #something;d$
```

Having read such a record from a file, you could display the information like this:

```
500 PRINT "Author:";d$( TO 40)
510 PRINT "Title:";d$(41 TO 96)
520 PRINT "Year:";d$(97 TO 100)
```

Some data can have an annoying number of fields - anaddress, for example, can include street, district, town, county and postcode. If you reserve enough space for the maximum field length assumed possible, you waste LOTS of disk space. An alternative approach is to stay with a fixed-length for each complete record, but handle the contents a bit more flexibly. For example, we could store a number of variable-length items per record as shown in the incomplete program section below:

```
600 DIM d$(100)
610 LET t$=""
615 DO
620 INPUT a$
630 EXIT IF a$=""
640 LET t$=t$+a$+CHR$ 13
650 LOOP
660 LET d$=t$
670 PRINT #4;d$
680 REM rest of program
```

The items are separated by carriage returns, which means that we need multiple INPUTS to read each complete record, as shown by the incomplete input routine below:

```
900 LET poi=PTR 4
910 DO
920 INPUT #4;a$
930 PRINT a$
940 LOOP WHILE PTR 4<ptr+100
```

The use of PTR as shown allows a variable number of substrings to be stored in each record. An alternative to INPUT is the INP$ function, which will read a fixed number of characters from a stream. All the lines above could be replaced by:

```
900 LET a$=INP$(#4,100) 910 PRINT a$
```

This requires you to search for the carriage returns in a$ if you want to chop it up into individual items - INSTR will be useful here (see Coupe manual).

STORING NUMBERS

Let's assume you want to store numeric data in a file. Sometimes you can simply PRINT the number, as in: PRINT #5;x. However you will have more control over exactly what part of the file is used if you do something like: LET d$(10 TO 12)=STR$ x. The details of the best method to use will depend very much on the range and precision of numerical values you want to store, and on how keen you are to save space. For example, if x is a whole number between 0 and 255 you can use e.g.: LET d$(65)=CHR$ x. (LET x=CODE d$(65) is the reverse.) You can limit a value to a fixed number of decimal places using e.g. LET x=INT(x*100)/100 before storage (this limits to two decimal places).

VARIABLE-LENGTH RECORDS

Sometimes fixed-length records are unsuitable because the data you are dealing with is so variable in size that too much disk space would be wasted. Sometimes variable-length records may be just simpler to program, particularly if you rarely or never want to update a record (which is tricky, expecially if the new data is longer). Also, you may want to read data from existing serial files made up of variable-length records. Once you are used to random-access you may find the delay associated with reading a record part-way through such a file (using multiple INPUTs) rather irritating. Unfortunately, the variable record length means that it is not possible to use the normal form of POINT to access a given record; e.g.:

```
POINT #stream,(record number-1)*record length
```

There are devious ways round this, like keeping a file of fixed-length data giving pointer values to each variable-length

record in another file, or a later part of the same file. But let's keep things simple, and exploit another feature of POINT instead. Something like:

```
POINT #5, OVER 10
```

will start from wherever the current file pointer is, and pass over 10 carriage returns before setting a new pointer position. So to point to the 2000th. record in a file, we could use:

```
POINT #5,0: POINT #5, OVER 1999
```

This will point to the start of the file, and then pass OVER 1999 carriage returns. Although POINT will have to read the first 1999 records in order to do this, it reads at about 22.5K per second, (or over 50K per second from RAM disk) so that for many files the time required is insignificant. Besides, if you know what record you have just INPUT, you often do not have to start at the beginning of the file again. For example, if you have INPUTed record number 2000, and want to INPUT record 2100 next, POINT #5, OVER 99 will work. It is even possible to re-define the character POINT OVER "passes over" as the program runs, using POKE DVAR 10, (character code), so you can separate records by one character (say, CHR$ 128) and fields by another, and use POINT OVER to find both the record and the field you want, very quickly. To illustrate this, the listing below creates a file of 1000 records, each terminated by CHR$ 128 and containing 6 random-length fields ending in CHR$ 13. The semi-colons at the end of lines 60 and 80 prevent carriage returns being sent to the file. You canload the program from the MasterDOS disk using LOAD "prog3". RUN to create the file. This will take some time, as the file will be about 150K long. Go for a tea or coffee break!

```
10 CLOSE #4
20 OPEN #4;"varifile"RND
0 FOR r=1 TO 1000
40 PRINT AT 10,10;r
50 FOR f=1 TO 6
60 PRINT #4;"Record:";r;" Field:";f;
70 PRINT #4;" abcdefghijklmn"( TO RND*14)
80 NEXT f: PRINT #4;CHR$ 128;
90 NEXT r
100 CLOSE #4
110 STOP
```

When you get back, the program below will let you read any desired field and record from the file:

```
200 CLOSE #4
210 OPEN #4;"varifile"RND
220 POINT #4,0
230 INPUT "Record? ";r
240 IF r=0 THEN CLOSE #4: STOP
250 INPUT "Field? ";f
```

```
260 POKE DVAR 10,128
270 IF r<>1 THEN POINT #4; OVER r-1
280 POKE DVAR 10,13
290 IF f<>1 THEN POINT #4; OVER f-1
300 INPUT #4;r$: PRINT r$
310 GO TO 220
```

OPENING MULTIPLE FILES

Our examples have only dealt with one file being open at a time, but it is quite possible to have many files open at once in random-access mode. However, if the files you are using involve writing to a new file, or extending an old one, then all those files must be on the same disk and disk drive, or disk space will be used very inefficiently. (Other files can use the second drive.)

MOVE and Random-Access Files

It is possible to MOVE a file to a stream OPENed to a random-access file. Since MOVE uses the equivalent of PRINT # to transfer data, the results are easy to predict. If you have just OPENed the random-access file, the file pointer will be at zero and the MOVEd data will overwrite the existing file until it has all been overwritten and the file begins to extend. If you use POINT and the file length function to set the file pointer to the file end, the data will extend the file without overwriting any of the existing data.

READING AND WRITING SECTORS

Technically-inclined users can use the READ AT and WRITE AT commands to access disk sectors directly. Both commands have a similar syntax:

READ or WRITE AT drive, track, sector, address, count.

For example:

```
WRITE AT 1,10,1,100000,20
```

This SAVES memory to drive 1, starting at track 10, sector 1, taking the data from address 100000 onwards. Twenty sectors are SAVEd (10K) so all of track 10 and 11 will be used. If the final number is omitted, 1 sector is assumed. The transfer speed for large numbers of sectors is about 22K/second with real disks. WARNING: WRITE AT takes absolutely no notice of what may be already in the sectors it writes to!

```
READ AT 3,0,1,65536,42
```

This example reads from disk 3 (a RAM disk), starting at track 0, sector 1, and LOADS 42 sectors to memory at 65536.

The drive must be 1-7, or you can use an asterisk to mean "current drive". The track should normally be 0-79 or 128-207 (side 1 and side 2 of a disk) and the sector must be 1-10. The address must be 16384-540671. The sector count must be 1-1024. This allows up to 512K to be handled at a time, in theory.

RAM disks will often have odd numbers of tracks, but they always start at 0, increase towards 79, and then skip to 128 and upwards, just like real disks.

When either command reaches the highest-numbered track on one side of a disk, and has to step to the next track, it will move to the first track on side two of the disk.

DOS VARIABLES and DVAR

You can change a DOS variable by typing: POKE DVAR n,x where n is the DOS variable number and X is the new value. You can also POKE with a string, or PEEK the DVAR using e.g. PRINT PEEK DVAR 2.

If you want to change a DOS variable so that you do not have to POKE it on every session (and remember, you can do this effortlessly with an AUTO file) the procedure is a little more complicated. You cannot SAVE the DOS directly from memory, so LOAD a mint copy from disk to some address - say, 65536. Then, where you would have used: POKE DVAR n,d use instead: POKE 65536+535,d. (The +535 is the offset that the DVARs are stored at inside the DOS file.) Then SAVE the modified DOS using: SAVE "name" CODE 65536,15700. Type: CALL 0 to reset the computer, and when you press F9 to re-HOOT you will find the new value is ready-set in the DOS variable.

Note: Some of the variables are likely to only be of interest to the very technically-inclined user. The DOS variables are:

0 Controls BORDER flashing when a file is being read from the disk. POKE DVAR 0,0 for no flash, or POKE DVAR 0,7 for the normal setting.

1 Drive 1 tracks and sides data. Contains the number of tracks on one side of a disk (usually 80) plus 128 if the disk is double-sided. The usual value is 208, but it can be POKEd if you have an different disk drive, or if you want, say, to partially FORMAT a disk.

2 Drive 2 data, as above. The DOS automatically sets this variable to 208 when it loads, if you have a second drive, unless you have POKEd the variable with a different value. Normally the value will be zero if you have just 1 drive.

3 Drive 1 step-rate. The default value of 0 lets the drive step as quickly as possible. Other values introduce a delay (in milliseconds) after each step command. E.g. POKE DVAR 3,3 would add a delay of about 3 milliseconds. Delays may be needed for reliable operation of older designs of disk drive.

4 Drive 2 step-rate, as above.

5 The character used as a blank in directory listings. Normally

this is a space, but e.g. POKE DVAR 5 ,"-" will fill the

blanks in the directory with hyphens. Experiment!

6 Reserved

7 Version number of the DOS, times 10, plus 201 (Times 10 so that version 1.0 will give 10, and plus 20 to distinguish MasterDOS from SAM DOS, which used numbers up to 20.) PRINT PEEK DVAR 7 will give 30 or more.

8 Setting for number of columns in a simple directory. Normally it is zero, which means "use as many columns as will fit". This will vary according to the current screen MODE and WINDOW settings. You can POKE other values e.g. 1 for a single-column listing or 7 if you want to get a wide listing on an e0-character-wide printer.

9 Alphabetic sorting control for simple directories. Normally it is 1, meaning "sorting on". POKE DVAR 9,0 to turn off sorting.

10 Character used by POINT #(stream), OVER x as a record delimiter. Normally CHR$ 13. See the section on handling variable-length records for details of its use.

11 Character treated as an extension separator when printing file names, as in a directory. Normally ". . Can be POKEd with other characters, or POKE it with zero to disable alignment of extension names.

12 Main symbol recognized as a separator in path names. Used in PATH$ and directories. Normally "\"

13 Alternate symbol recognised as a separator in path names. Normally "/".

14 Skew between tracks in a disk FORMAT. Normally 255, (which acts as minus 1 here) which means that for each track, the last sector on the track (10) is opposite sector 9 on the next track. This means that the drive head has about 20 milliseconds to step to a new track and settle down before sector 1 arrives, because the first sector it will reach on a new track will be sector 10, which will be ignored and will take 20 msec. to pass by. SAM DOS used a skew of 254, which allowed 40 msec. and might be better with old drives, although LOAD and SAVE will be about 10% slower. You could experiment with a skew of 0, allowing only a few msec. for drive stepping. This just about works - but if it corrupts your files, don't blame me!

15 Default drive number used internally.

16 Number of directory tracks on last-used disk. 17 Code number for current subdirectory.

18 Code number for alternate current subdirectory in file TO file operations.

19 DIR's "show DATES" flag. 20 Count of OPEN files.

21 Highest subdirectory code number so far.

22 (2 bytes) Address of hook code address table when DOS is paged in at 16384.

24 MOVE TO #2 flag. Determines what happens to characters with codes above 127 when they are MOVEd from a file to the screen. Normally 0, meaning that such characters are reduced by 128 and then printed in INVERSE 1. POKE with 1 to print them instead as UDGs or block graphics.

25 MOVE TO #2 control code replacement character. Used instead

of CHR$ 0-31 and CHR$ 255, which might cause errors. Normally ".".

26 Page switched in at 32768 when the BREAK button is pressed with SAMCO's Spectrum emulator loaded, and then the keys 1 or 5 are pressed. Normally 4.

27 (2 bytes) Address CALLed in circumstances above. Normally 4, which has no effect, but allows for e.g. Screen dumps in future.

29 Number of 0.25 second delays before a SAVE, minus 1.

30 (3 bytes) Reserved.

33 (2 bytes) ON ERROR address when a command fails syntax.

35 Reserved.

36 ON ERROR page used if ON ERROR address >32767

37 (2 bytes) Increment used between sectors during multi-sector READ AT and WRITE AT. Normally 512. Other values can be used when working with non-Coupe disks, and 510 can be useful for looking at SCREEN$ files.

39 (5 bytes) Tracks per drive for each RAM disk, or zero.

44 (5 bytes) First 16K page for each RAM disk.

49 (7 bytes) Current subdirectory number for each drive.

56 (7 bytes) Path length for each drive.

63 (14 bytes) Random double-byte for each drive.

77 (2 bytes) Random double-byte when first file OPENed.

79 (2 bytes) Temporary clock variable.

81 (9 bytes) DATE as dd/mm/yy, return, after clock read.

90 (8 bytes) DATE max/min values for d/m/y inputs.

98 (9 bytes) TIME as hh:mm:ss, return, after clock read.

105 (8 bytes) TIME max/min values for h:m:s inputs.

111 (7 bytes) Drive each drive pretends to be. Normally 1-7.

118 (32 bytes) External RAM table. One bit denotes each possible 16K external memory page, being zero if it is available, or 1 if it does not exist or is in use. Bit 7 of the first byte represents the first 16K page.

150 Port value used by add-on clock/calendar.

DIRECTORY ENTRY FORMAT

This information is intended for those interested in using READ AT and WRITE at to deal directly with the disk directory when writing utility programs.

The first tracks on side 1 of a disk are used for the directory, starting at track 0, sector 1. Each sector holds two 256-byte directory entries. The format of each entry is as follows:

BYTE

0 Status/File type, as returned by the FSTAT function with a parameter of 4. Zero if entry unused or ERASEd.

1 (10 bytes) File name. If the first byte is zero, the entry is assumed to have never been used and directory reading will not proceed further.

11 MSB of number of sectors used in the file.

12 LSB of number of sectors used in the file.

13 Track number of start of file.

14 Sector number of start of file.
15 (195 bytes) Sector usage map for the file. Bit 0 of the first
    byte represents track 4, sector 1, bit 1 represents sector 2,
    etc. If a bit is set the sector is used by the file.
210 (10 bytes) In all entries except the first one on the disk,
    these bytes hold Plus D/Disciple-form information. In the
    first entry, the disk name is stored here. Bit 7 of byte 210
    can be set or reset without altering the disk name.
220 Flags
221 (10 bytes) File-type specific information. If the file type is
    SCREEN$ then byte 221 is the screen MODE, minus 1.
232 (4 bytes) Reserved.
236 Start page number in bits 4-0, bits 7-5 are undefined.
237 (2 bytes) Start offset in the range 32768-49151.
239 Number of pages in length.
240 (2 bytes) Length MOD 16384. Bits 15 and 14 are undefined.
242 Execution page for CODE files, or 255
243 (2 bytes) Execution offset (32768-49151) for CODE files, or
    auto-run line for a Basic program.
245 Day of SAVE, or 255
246 Month of SAVE
247 Year of SAVE 248 Hour of SAVE
249 Minute of SAVE
250 For a DIR file, code number tagging files in that sub-
directory.
251 Reserved.
252 (2 bytes) In the first directory entry only, Random word
    identifying the disk.
254 Subdirectory code number of this file.
255 In first entry only, number of directory tracks, minus 4.

ERROR MESSAGES

Master DOS uses the following error messages, in addition to using
the non-DOS messages available on the basic Coupe.

84 Escape requested
    <ESC> key pressed.
85 TRK- ,SCT- Error The sector at the specified track and sector
    could not be loaded.
86 Format TRK- lost
    The specified track has been corrupted.
87 Check disk in drive
91 Invalid device 93 Verify failed
    Data on disk does not match that in the computer.
94 Wrong file type
    e.g. LOAD "name" when "name" is a CODE file.
99 Reading a write file
    Reading a file OPENed with OUT, or default of OUT (new file).
100 Writing a read file

```
     writing a file OPENed with IN, or a PROTECTEd file.
101  No AUTO*  file
     Second use of BOOT looked for "auto*" and didn't find it.
103  No such drive
     Drive not fitted or RAM disk not FORMATed.
104  Disk is write protected
     Move the write-protect tab!
105  Disk full
     Disk data area is full.
106  Directory full
     Free slots in the directory have run out.
107  File not found
109  File name used
     OPEN DIR "name" when "name" exists, or OPEN.
111  Stream used
     You'll have to CLOSE the stream, or use CLEAR #.
112  Channel used
113  Directory not found
114  Directory not empty
     ERASE DIR when the directory contains files.
115  No pages free
     COPY and BACKUP require at least one free page.
116  PROTECTED file
     SAVE, COPY or ERASE when file is protected.
```

MasterDOS provides hook (command) codes which enable the machine code programmer to use the DOS's facilities without having to return to or call SAM BASIC.

If an error occurs, MasterDOS puts an error number into the A register; otherwise the A register will be zero.

The Hook Codes currently available are:

- INIT 128 (80H)

  Look for an AUTO file on the current disk. No action (or error) occurs if there is no AUTO file, otherwise it is loaded (and executed if it is an auto-running Basic or CODE file). This Hook can only be used in sections B and C of the memory map.

- HGTHD 129 (81H)

  Get file header. This routine should be called with IX pointing to the UIFA, which should hold the file type required (at IX+0) and the file name (at IX+1 to IX+10). The routine looks for the file in the current directory on the current drive and either returns with an error code, or transfers the data from the file directory entry to IX+80 dec, in UIFA form. The calling code and the UIFA can be in sections B, C or D of the memory map. (Note: this hook works correctly in SAMDOS, provided IX=4BOOH.)

- HLOAD 130 (82H)

  Load data from the file you have just got the header of using HGTHD above. HL must point to the destination address, paged in between 8000H and BFFFH, i.e. in section C of the memory map. The C register should hold the number of 16K pages in the file, and DE should hold the length MOD 16K. These values can be read from the header loaded by HGTHD. See also Hook 143.

- HVERY 131 (83H)

  Like HLOAD, but verify the data on the disk against the data in memory. Error code 93 dec if verify failed.

- HSAVE 132 (84H)

  Save the file whose UIFA is pointed to by IX. All relevant data in the UIFA must be complete - for a CODE file, type, name, start, length and execute address. If

in doubt, try a SAVE from BASIC and then look at 4500H-4B47H to find the required values.

- HSKSF 133 (85H)

  Seek Safe. On some machines, pressing the Reset button can corrupt the disk sector under the drive head. This is often on the track containing the last sector of the last file loaded. MasterDOS tries to minimise the problem by parking the drive head on the last track in the directory, after a LOAD or a SAVE. This track will be unused unless the directory is fairly full. Using the HSKSF hook will move the head of the current drive to the last track in the directory, unless this would be track 4 (which contains the first sector of DOS) in which case track 3 is used instead.

- HAUTO 136 (88H)

  Like Hook 128, but an error code of 101 dec is returned if there is no AUTO file.

- HSKTD 137 (89H)

  Seek Track D. Move the drive head of the current drive to the track specified in the D register.

- HFMTK 138 (8AH)

  Format Track. Format the track under the drive head, using the D register to supply the track number and the E register as the number of the first sector (1-10). Later sectors will be numbered 1 higher till 10 is reached and numbering goes back to 1. [Does not exist?]

- HVAR 139 (8BH)

  Supply the address of a DVAR by putting it on the floating point calculator stack. On entry, the FPCS should hold the desired DVAR number. Note: it is probably easier to page in DOS (the DOS page is held at 5BC2H) and read the disk variables directly. DVAR 0 is at an offset of 0220H within the page - this will not change.

- HEOF 140 (8CH)

  Supply the End-Of-File status (1 or 0) of a specified stream. The stream number should be on the FPCS. It will be replaced by the EOF status.

- HPTR 141 (8DHl)

Supply the PTR value for a specified stream. The stream number on the FPCS is replaced by the PTR value.

- HPATH 142 (8EH)

  Supply the current PATH$ on the FPCS Use CALL 0124H (JSTKFETCH) to get page (A) offset (DE) and length (BC) of the string.

- HLUPG 143 (8FH)

  As Hook 130, but on entry the A register should hold the page number of the destination address. This need not be paged in.

- HVEPG 144 (90H)

  As Hook 131, but on entry the A register holds the page to verify against.

- HSDIR 145 (9lH)

  Select Directory. Similar to DIR="name" in Basic. On entry, the registers hold details of the location and length of the desired subdirectory name. DE is the offset, A is the page of the name start, and BC is the name length.

- HOFSM 146 (92H)

  Open a File Sector Map for an OPENTYPE file. IX must point to the UIFA. The routine will create the map and clear the disk buffer.

- HOFLE 147 (93H)

  Open a file on the disk. IX must point to the UIFA. The routine will create a sector address map, and save a 9-byte header to the disk buffer.

- HSBYT 148 (94H)

  Save the byte in the A register to the disk file (If the buffer is full it will be written to the disk and the byte will go into the start of the next buffer.)

- HWSAD 149 (95H)

  Write Single Sector. On entry, the A register is the drive number (1-7) which is used to access the table at DVAR 111 to get the actual drive to use. D holds the destination track, and E the sector. HL points to the source in memory, which must be in sections B, C or D of the memory map. 512 bytes will be written to disk.

- HKSB 150 (96H)

  Save a block of data to the disk file. The A register holds the length to save in pages, and DE holds the length MOD 16K. HL points to the start of the data to save, paged into section C of the memory map.

- HDBOP 151 (97H)

  Save BC bytes to the disk file. DE points to the start of the data to save, paged into section C of the memory map. Used by DOS to write strings to OPENTYPE files.

- HCFSM 152 (98H)

  Close a file. This routine writes the last buffer to a disk file and creates a directory entry for it. IX should point to the UIFA.

- HORDER 153 (99H)

  Sort list into ASCII order. HL should point to the start of the list in sections B, C or D of the memory map. The BC register should hold the length of each item in the list, and the DE register the number of items. The A register specifies the number of characters to sort on. No paging is performed so the entire list must be paged in by the user before this hook is called.

- HGFLE 158 (9EH)

  Get a file from the disk. The IX register must point to the UIFA. The return is made with the first sector of the file loaded into the disk buffer and RPT pointing to the first byte.

- HRSAD 160 (A0H)

  Read Single Sector. On entry, the A register is the drive number (1-7) which is used to access the table at DVAR 111 to get the actual drive to use. D holds the source track, and E the sector. HL points to the destination in memory, which must be in sections B, C or D of the memory map. 512 bytes will be read from the disk.

- HLDBK 161 (A1H)

  Load a block of data from the current disk file. HL points to the destination of the data in memory, paged into section C of the memory map. The A register is the length to load, in pages, and DE holds the length MOD 16K.

- HMRSAD 162 (A2H)

  Read Multiple Sectors. Equivalent to READ AT in Basic. The A register is the drive to use (1-7, using DVAR 111 table), D holds the track, E the sector, C the page and HL the offset (8000H-BFFFH) of the destination. IX holds the number of sectors to load.

- HMWSAD 163 (A3H)

  Write Multiple Sectors. Equivalent to WRITE AT in Basic. As above, but C and HL hold the source address, rather than the destination.

- HREST 164 (A4H)

  Restore. Move drive head to track 0. The disk need not be formatted.

- HP2IR 165 (A5H)

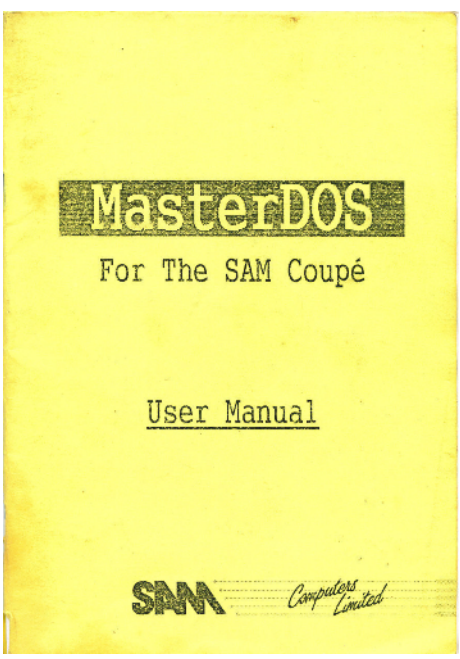
  Print directory. If the A register holds 2, print a simple directory. If it holds 4, print a detailed directory. Neither option does a CLS first. The current stream is used to output.

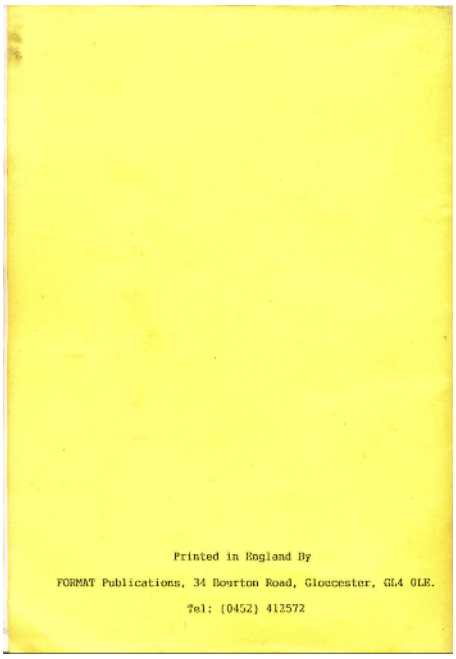
- HERAZ 166 (A6H)

  ERASE a file from disk. The file name should be at IX+1 to IX+10.

- HCHRD 168 (A8H)

  Read character from the disk file whose UIFA is pointed to by IX. The character and flags are passed out in the alternate BC register: EXX, PUSH BC, EXX, POP AF gives the character in A, and the carry flag set if the read was OK, else we hit end of file.





Sam's MasterDos Manual Front & Back Covers

This User Guide was OCRed with Textbridge Pro 11
& MS Word 2003. The PDF document was compiled with
JAWs Creator pdf version 6.3
by Steve Parry-Thomas 11 December 2004

For SAM Coupé uses everywhere.

www.samcoupe-pro-dos.co.uk

SAM MasterDos Manual
PDF version 1a – 11 December 2004

[Version number may change as errors and text formatting are corrected There is bound to an error or two, I've over looked or some text formatting that's been left for another day]