MasterBASIC for the SAM COUPE

User Manual

BETA SOFT

CONTENTS

INTRODUCTION

Thank you for buying MasterBASIC. This program provides major enhancements to the SAM Coupe's editing, data-handling, sound, graphics and printing capabilities. I hope you like it.

The master disk supplied with this manual contains a program called "autoMBM" that will combine MasterBASIC with your DOS (preferably MasterDOS, but SAMDOS2 will do) to give a single convenient BOOTable file. Before running the program, make sure you have at least one disk available which has been formatted. The first position in the directory should either be free or contain a file you do not mind overwriting. This disk or disks will hold working copies of the DOS/MasterBASIC file, with the name "SD+MBASxx" or "MD+MBASxx" according to whether SAMDOS or MasterDOS was used, with "xx" being the current version number. To run the program, just place the supplied MasterBASIC disk into drive 1 and press the F9 key. The program will tell you what to do at each stage.

You can load your working copies of MasterBASIC by resetting the computer, inserting the working disk and pressing the F9 key, as you do to load DOS. If you have somehow forgotten to do this to start with, you can over-write SAMDOS or MasterDOS in memory using BOOT 1. (Note: Any RAM Disks you are using will be lost.)

The "autoMBM" program performs extensive linking with the DOS to give full compatibility and improved DOS performance. When used with MasterDOS, the DOS version number (PEEK DVAR 7) is increased to 50 to reflect these changes. (Any known faults in MasterDOS are corrected by the "autoMBM" program.) A few MasterDOS features, such as an improved FORMAT, are provided even with a SAMDOS/MasterBASIC combination (DVAR 7 will be set to 21 to reflect this). However, it is a very good idea to purchase MasterDOS if you have not already done so.

MasterBASIC requires ROM 20 or later (PRINT PEEK 15) to work.

The DOS/MasterBASIC BOOT file takes up two of the Coupe's 16K pages. The DOS part is loaded into the highest free page, as usual, and the MasterBASIC part goes into the next highest free page. After altering the program as you wish using DVAR and XVAR POKEs, the entire program can be SAVEd by a simple SAVE BOOT command - see SAVE BOOT.

Ensure that you keep the original MasterBASIC disk in a safe place - though naturally I will replace it if anything bad happens to it. Just return the disk and enclose a stamped addressed envelope. If you live abroad forget the stamp but enclose an International Postal Reply Coupon instead.

If you bought this product directly from Betasoft, you are already recorded as a customer with possible upgrade privileges. If you bought it elsewhere, it would be a good idea to send me your name and address and tell me where you got the product. This information may be used to tell you about future products.

Thanks to the many users who contributed suggestions for an extended BASIC. If your idea isn't implemented here, do not give up hope - maybe I just haven't managed it yet! If you have any suggestions, or problems with this program or manual, please let me know. Don't assume that your problem is unimportant, or that I must know about it already. I am always interested in your comments or suggestions. Please write to:

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

EDITING AND DEBUGGING FEATURES

WORD LEFT or RIGHT

MasterBASIC allows you to move the cursor left or right by an entire word by pressing SHIFT and the left or right cursor key. This makes editing faster, particularly in long lines, and it is also handy when you are INPUTing or EDITing (see EDIT in this manual) a long text string.

The word left and right facility is provided by the use of two new control codes, CHR$ 24 and CHR$ 25. When these codes are received from the keyboard they tell the editor to perform the word left or right action. MasterBASIC does the equivalent of KEY 36+70,24 and KEY 27+70,25 to assign these codes to the shifted left and right cursors (see the Keyboard Map on page 180 of The Coupe User's Guide). You can assign the same codes to other keys if you like. Note that a word is taken to be any block of characters delimited by spaces, or the start or end of the editing area.

LAST LINE RECALL

MasterBASIC allows you to recall lines that you typed earlier by holding down CNTRL and pressing the up-arrow cursor key. You can then edit the line, if required, and enter it again. Once you get used to this facility you will wonder how you ever managed without it! For example, if you try to load a file from the wrong disk you can simply insert a new disk and press CNTRL/up-arrow and then RETURN to repeat the command. Or if you type something and the computer doesn't do what you expect, you can check to see if you really typed what you thought you did.

If you press CNTRL/up-arrow twice you will recall the line you typed before the last line. In fact you can keep recalling lines until eventually you go right "round" the line-storage buffer and come back to where you were when you came in. (The buffer capacity is 256 bytes.) You can also use CNTRL/down-arrow to move from, say, the second-to-last line to the last line. Try it!

An easy way to transfer a few lines from one program to another is to edit the lines you want to transfer, without making any changes to them (if you know what I mean). Then LOAD the program you want to add the lines to, and use CNTRL/up-arrow to recall the lines and RETURN to add them to the listing.

JOIN program lines

See also: SPLIT program lines

```
e.g.   JOIN 100
       JOIN
```

This command joins together the specified line (or, if one is not specified, the line with the current line cursor) and the one below, if there is one. The JOINed-on line will lose its line number, and will be separated from the text of the first line by the normal inter-statement marker, a colon.

SPLIT program lines

MasterBASIC allows program lines to be split by typing a slash (/) at the position for the "cut" and pressing RETURN. The slash must be the first non-space character after the colon inter-statement marker. The part of the line before the slash will be put into the listing. The remainder of the line, with a copy of the original line number, will remain in the editing area. The slash disappears automatically. The cursor will be just to the right of the line number, ready for you to alter it before pressing RETURN (unless you want to overwrite the first part of the line in the listing). If you enter:

```
10 PRINT "hello": GO TO 10:/ PRINT "goodbye"
```

Then

```
10 PRINT "hello": GO TO 10
```

will appear in the listing, and

```
10 (cursor)PRINT "goodbye"
```

will remain at the bottom of the screen.

SEARCHING THE PROGRAM

REF (reference)
REF (reference),first line
REF (reference),first line,last line

This command is used to search a program for a specified "reference", which can be a variable, number, or sequence of characters. You can specify the first line to search or limit a search to a range of lines. When the reference is located, the line containing it will appear in the edit line with the cursor just after the reference. Simply press RETURN if you do not wish to alter the line. To find any more examples of the reference, press RETURN again, and the search will continue, until eventually "O.K." appears. If you enter a command, rather than pressing RETURN, REF assumes that you are finished, and you will have to re-enter the REF command to look for more instances.

In the examples below, a character that cannot be a letter, a number or "$" if the match is to succeed is shown by "_ ".

REF a$	Looks for: a$
REF a$,100	Looks for: a$, starting at line 100
REF a$,5,90	Looks for: a$, from line 5 to 90
REF count	Looks for: _count_
REF "count"	Looks for: count
REF 1	Looks for: 1 (invisible form)
REF "1"	Looks for: 1
REF 12*4	Looks for: 12 (invisible form)*4(invisible form)
REF (a$)	Looks for: value of a$ (e.g. if a$="fish", looks for "fish", not "a$".
REF (x)	Looks for: value of x,such as 10(invisible form)

It does not matter whether any letters in the reference are in capitals or not. For example, both REF abc$ and REF ABC$ will find "abc$", "ABC$" or "AbC$".

When looking for numeric variables, the requirement for the target to start and finish with a character that is not a letter or a number prevents, confusion. between, for example, "count" and "account'' or "count '. When looking; for numbers, the search also looks for the invisible 5-byte form that follows them in a Basic line, again preventing confusion. Variables and numbers are not looked for inside strings - so if you REF zebra you will not find "The animal is a zebra'. To look for a sequence of characters anywhere in the program, including inside strings, just enclose the sequence in, quotes - e.g. REF "zebra" or, if you wish to use a string variable, enclose its name in brackets so that REF can tell you don't wish to search for references to the name of the variable, but to its value.

If you want to search for a keyword, the easiest way is to type something like: REF print" and then press RETURN. The line will fail syntax and the "token" form of PRINT will appear. Backspace twice and add a quote mark before PRINT. (This will cease to appear as a keyword, but this doesn't matter.) Now press RETURN. You could also use: REF (CHR$ 187) - the internal token code for PRINT. See Appendix A for a list of these codes.

LISTING PROGRAM REFERENCES

```
 PRINT REF (reference)
LPRINT REF (reference)
```

See also: REF

PRINT REF (reference) gives a list of the line numbers in which a specified "reference" occurs. This can be a variable name, a number, or a sequence of characters. PRINT REF is related to the REF command explained in this manual. See REF for more detail on what a "reference" can be. PRINT REF can specify a range of line numbers in the same way that REF can. For example:

```
   PRINT REF test,100,500
```

If the reference is used more than once in a line, the line number will be given more than once too. For example, if your program is:

```
   10 FOR n=1 TO 10: PRINT n
   20 NEXT n
```

then PRINT REF n will give:

```
   10
   10
   20
```

LPRINT REF n would send the list to the printer.

PROGRAM SEARCH AND CHANGE

ALTER (reference) TO (reference)

This command looks through the program for all occurrences of the first reference and alters them to the second reference. A "reference" may be a variable, number, or sequence of characters. (More details on the requirements for a successful search are given under REF.) A range of line numbers can be specified, as with REF. Below are some examples:

```
ALTER a$ TO andy$
```

will alter any occurrences of the variable name "a$" to "andy$".

```
ALTER count TO c,200
```

will alter the numeric variable name or procedure name "count" to "c", starting at line 200. Note that "account" or "counts" will not be affected.

```
ALTER 1 TO 23
```

will alter the number 1 to 23, and the invisible five-byte form which follows will also be changed. However:

```
ALTER 1 TO "23"
```

would replace "1" and the invisible five-byte form with just the two characters "23", and the program would not work correctly, although the altered lines would appear to be correct. In all the above cases, characters in the program enclosed in quote marks will not be found; it is assumed that you wish to alter variable names, procedure names, or numbers. However, if you alter a string of characters to something else, as in:

```
ALTER "break to stop" TO "any key to stop"
```

the string will be found anywhere in the program, even inside quotes. (The quote marks themselves are not looked for.) If you wish to do something like this, but with a variable in place of one or both literal strings, you must enclose its name in brackets so that ALTER can tell you don't want to alter the name of the variable, but something specified by its contents. For example, to correct a spelling error:

```
LET s$="excecute": ALTER (s$) TO "execute"
```

If you wish to use ALTER with strings containing keywords, you can enter them using the method explained for REF.

To DELETE a reference, use the null string, e.g.:

```
ALTER "word" TO ""
```

Ordinary numbers in a program are always followed by an invisible CHR$ 14 and the value of the number in a special invisible five-byte form that can be handled faster by the computer than the digits you can actually see. One consequence of this is that numbers often take more room than variables or expressions, and you can save memory by altering any number to VAL "number". An extra 2 bytes are used by the internal form of VAL, and 2 are used by the quotes, but we still save 2 bytes. To do this for all numbers from 1 to 100, we could use the following (on your disk as "alter"):

```
1 LET spare=FREE
2 FOR n=1 TO 100
3 PRINT AT 10,10;n
4 ALTER (n) TO (CHR$ 255+CHR$ 110+CHR$ 34+STR$ n+CHR$ 34),8
5 NEXT n
6 PRINT "Saved ";FREE-spare;" bytes"
7 STOP
```

(CHR$ 255+CHR$ 110 is the internal code for VAL, and CHR$ 34 is the code for a quote mark - see Appendix A.)

The "n" in line 3 is in brackets so that ALTER knows that we wish to look for the current value of the variable, not the variable name itself.

NOTE: Be cautious when using ALTER. You can make irreversible changes to a program if you are careless. For example, if you change all uses of the numeric variable "apple" to "a" and then realise that you have already used the variable name "a", you cannot simply alter all uses of "a" to "apple" to rectify matters! You should have checked for previous use of "a" using the REF or PRINT REF command.

LINE NUMBER TRACING

```
LINE
LINE delay
LINE STEP
LINE OFF
```

LINE controls display of the current line and statement number while a program is running. This feature, called TRACE or TRON in some other Basics, is useful for debugging a program. The information is displayed on the lower right-hand side of the screen, in PEN 0 on PAPER 15.

LINE used by itself shows the information without any delays. This may be too fast to read, so you can specify a delay to make life easier. LINE 1 will give a very short delay, LINE 10 is longer and LINE 200 is very long. You can also use LINE STEP - this will display the number of the line about to be executed and then wait for you to press CNTRL before continuing. You can hold down the CNTRL key to let the program run.

LINE OFF turns off the line number output.

SPEED IMPROVEMENTS

SAM BASIC keeps internal information on the locations of procedures and functions, and the values of LABELs. This has to be updated after the program is edited, before it can be executed. In long programs this can introduce an annoying delay before the program starts to do anything. MasterBASIC does this updating rather faster, especially with long programs, which makes writing and testing such programs easier. MasterBASIC also uses a more sophisticated method of moving memory around, which amongst other things speeds up the addition or deletion of lines in the listing. See POKE for more information on this improvement.

Cursor movement has been speeded up in LIST FORMAT 1 and 2. This is particularly noticeable when editing long lines in MODE 3.

DATA-HANDLING COMMANDS

EDIT variable

The EDIT command can be used to modify the values of variables that already exist. It is particularly useful when you have INPUT a long string and then discover an error in it. EDIT will give you the string back, with your cursor at the end of the line ready to alter it. Without EDIT, you would normally have to re-type the entire string. Here are some simple examples:

```
10 LET name$="Jon Brown"
20 EDIT name$
30 PRINT name$
40 LET num=365.253
50 EDIT num
60 PRINT num
```

EDIT has a syntax very similar to that of INPUT. You can use semi-colons, commas, AT, TAB, LINE, # stream and prompt strings in the same way as you can with INPUT. However, only one variable can be edited with a single statement; if you use more, all except the first one are treated as INPUT variables.

The following example (which is on your disk as "edit") lets you create an array and then edit it, as you might want to do with a database:

```
 10 DIM db$(10,15)
 20 FOR n=1 TO 10
 30    INPUT db$(n)
 40 NEXT n
 50 PRINT "Editing"
 60 PRINT
 70 DO
 80    INPUT "Record number? ";rn
 90 EXIT IF rn=0
100  EDIT ("Record ";rn;" ");db$(rn)
110 LOOP
```

If the variable specified in the EDIT command does not exist, the command is exactly equivalent to INPUT.

SORTING DATA

```
   SORT a$
or SORT ABS a$
or SORT ABS INVERSE a$
```

SORT arranges strings in a string array or letters in an ordinary string into ascending or descending order. It can also be used to sort numbers coded as strings. The command's use with string arrays will be discussed first. Here is a program to generate an array of 100 random 10-.letter strings (on your disk as "sort"):

```
100 DIM a$(100,10)
110 FOR str=1 TO 100
120    FOR ltr=1 TO 10
130       LET a$(str,ltr)=CHR$(RND(25)+65)
140    NEXT ltr
150 NEXT str
200 FOR str=1 TO 100
210    PRINT a$(str)
220 NEXT str
```

As soon as the array has been created - which will take a little while - it is printed for you. Now add: 190 SORT a$ and then GO TO 190 and the array will be SORTed and printed again (avoid RUN or you will lose your array). The time taken to SORT this array is about 0.14 of a second; this time will increase relatively little if you use longer strings. The number of strings is more important - 200 will take about 0.46 seconds, 400 about 1.6 seconds and 800 about 6 seconds.

The normal SORT does not care if letters are in capitals or not. (If you want to get technical, bit 5 of each character's CODE is irrelevant to sort order, so CHR$ 96-127 are seen as equal to CHR$ 64-95.) This is probably what you want for sorting text, but there is another option that is more suitable for other kinds of data: SORT ABS a$. With this option, strings are sorted entirely according to their ASCII CODEs, so that "A" comes before "a", and "Z" also comes before "a" - see the list in Appendix A of this manual. This is the form to use if you are sorting numbers coded as strings using the SVAL$ function (see SVAL$, and below) or other kinds of coded data. The sort speed is also slightly better using this ABS option.

SORT ABS can also do an INVERSE sort - try changing line 190 to 190 SORT ABS INVERSE a$. The strings will be sorted into reverse order.

You can select any given block of strings to sort with a slicer. For example:

```
   SORT a$(1 TO 20)
```

   will sort only the first 20 strings and:

```
   SORT a$(30 TO )
```

will sort all the strings from string 30 onwards.

It is also possible to sort according to particular parts of the strings:

```
SORT a$()(2 TO )
```

will sort the entire array on the basis of the second and subsequent letters of each string - the first letter is not taken into account, although it is moved with the rest of the string. (Note that we had to use two slicers even though we wanted to sort the whole array. This is because SORT expects the second of two slicers to specify the part of the string to consider.)

Complex forms are possible, such as: SORT ABS q$(4 TO 9)(1 TO 5)

SORT makes it very easy to develop a fast and flexible data base. In this context, it is common to call the array a "file" and its strings "records". Areas of each string would probably be reserved for particular kinds of information, and would be called "fields". You would often want to use relatively long strings; a file of names and addresses and other data - say, age - might be set up so that the first 20 characters in each record (string) were the person's name, the next 20 their address, and the last character their age. Since age is bound to be in the range 0 to 255, we can use something like:

```
LET a$(str,41)=CHR$ age
```

If we wish to place the age data into record STR. Such storage of numbers is simple and saves memory, but suppose we needed to store something more complex, like a bank balance? If you use:

```
LET a$(str,41 TO 46)=STR$ balance
```

the information will be stored in the string, but it will be left-justified (the "9" of a 9 Pound balance will be in position 41, as will the "1" of a 100 Pound balance). This will prevent SORT from working correctly on this field. The answer is to format everything neatly so that any decimal points line up, and we have all the units, tens and hundreds in the same position for each string. This is easy with the formatting function, USING$:

```
LET a$(str,41 TO 46)=USING$("000.00",balance)
```

(See USING$ for a complete description of this function, and SVAL$ for information on string-coded numbers.) You can now do things like sort a set of records on the basis of age, or bank balance, and then sort the top 20 into alphabetical order. Note that because the CODE for "1" comes before that for "2", SORT will give smaller numbers first, and SORT INVERSE will give larger numbers first, when the numbers are represented as strings in this way. You will need to be careful with the fielding of your data - e.g. make sure that the first letter of all the last names occurs in the same position in each string.

SORT will also work on ordinary strings and one dimensional string arrays:

```
INPUT s$: SORT s$: PRINT s$
```

will give " BdeFgglors" if "Fred Bloggs" is input. This doesn't look very useful, but it allows SORT to work on certain kinds of numeric data which can be stored most efficiently in strings using, for example:

```
LET s$(position) = CHR$(data)
```

You would probably use the SORT ABS form.

Below is a program to demonstrate sorting of numbers coded as strings. See also: SVAL$ and NVAL.

```
1000 DIM num$(100,5)
1010   FOR n=1 TO 100
1020   LET num$(n)=SVAL$(RND*100-50,5)
1030 NEXT n
1040 PRINT "Sorting"
1050 SORT ABS num$
1060   FOR n=1 TO 100
1070   PRINT NVAL num$(n) 1080 NEXT n
```

Note that mixed positive and negative numbers can be dealt with. Numbers coded in this way can be sorted about 4 times faster than "real" numbers could be - which is one reason that I have not implemented SORT for normal numeric arrays.

DELETING STRINGS AND STRING ARRAYS

```
DELETE a$
DELETE a$(start TO end)
```

DELETE is used to remove part or all of a string or string array. Some examples are on your disk as "delete". For example:

```
10 LET a$="123456789"
20 DELETE a$(3 TO 6)
30 PRINT a$
```

This could be done by other methods, but not so simply or quickly.

DELETE is ideal for removing unwanted information from a database stored in a string array. Here it is used to delete the fifth string:

```
110 DIM dat$(10,10)
120 FOR n=1 TO 10
130       LET dat$(n)=STR$ n+" gwerty"
140 NEXT n
150 DELETE dat$(5)
160 FOR n=1 TO LENGTH(1,dat$)
170    PRINT dat$(n)
180 NEXT n
```

LENGTH is a SAM BASIC function for telling you the size of an array - useful when this keeps changing! Try altering line 50 to DELETE dat$(TO 3) or DELETE dat$(3 TO 6) or DELETE dat$.

JOINING STRINGS AND STRING ARRAYS

JOIN TO a$,b$

JOIN TO is used to add a copy of the second string or string array onto the first. When using strings, JOIN TO a$,b$ is equivalent to LET a$=a$+b$, but it is faster and requires less free memory. Try this program (which is on your disk as "join"):

```
10 LET a$="",b$=STRING$(100,"a")
20 DO
30     JOIN TO a$,a$
40     PRINT AT 10,10;LEN a$
50 LOOP
```

Now replace line 30 with the line below and try it again:

```
30 LET a$=a$+b$
```

Another example shows that the second string is unchanged:

```
110 LET a$="abcdefg",b$="123456"
120 JOIN TO a$,b$
130 PRINT a$,b$
```

String arrays can be joined together provided the strings in both arrays are the same length. For example:

```
210 DIM fi$(10,8),se$(1,8)
220 LET se$(1)="testing"
230 JOIN TO fi$,se$
240 PRINT fi$(11),se$(1)
```

Note that the array fi$ now has 11 strings in it, not 10. The function LENGTH (see The User's Guide) can be used to keep track of changing array sizes. JOIN TO is useful for adding new data to an existing array, either one string at a time, as in the previous example, or by merging two entire arrays.

DATA-HANDLING FUNCTIONS

SEARCHING STRING ARRAYS

```
INARRAY (a$(start element),target$)
INARRAY (a$(start element,slicer),target$)
INARRAY (a$(start element,slicer),target$,ABS)
```

See also: INSTR function in The User's Guide.

This function searches a specified string array for a target string, and returns the number of the first string in which it is contained, or 0 if it is not found. It is essentially an array version of INSTR; I suggest you make sure that you understand INSTR before you try out INARRAY - see The User's Guide.

The example below (on your disk as "inarray") will find all the "howdy"s in an array:

```
10 DIM a$(20,10)
20 LET a$(RND*19+1)="howdy"
    REM repeat the above for more targets
30 LET num=1
40 DO
50    LET num=INARRAY(a$(num),"howdy")
60 EXIT IF num=0
70    PRINT num;" ";a$(num)
80    LET num=num+1
90 LOOP UNTIL num >20
```

The "a$(num)" with num=1 in line 50 specifies that the search should begin with the first string in the array. When an occurrence of the target string is found, it will be printed, and the search will continue from the next string (num+1). The EXIT IF will jump out of the loop when no more "howdy"s are found, and the loop will finish anyway if we try to continue the search beyond the last element (LOOP UNTIL num>20).

The example above used case-insensitive matching, so that "HOWDY" would be found too. To specify absolutely precisely the string to match on, follow it with ABS - e.g.

```
70 LET num=INARRAY(a$(1),"test",ABS )
```

This also has the advantage of being faster than the first method, but you are unlikely to notice the difference unless you are using very large arrays.

The whole of each string in the array will be searched for the target unless you specify otherwise. If you wish to limit the search to a specific section of each string, use a slicer like

```
50 LET num=INARRAY(a$(num,3 TO 7),"howdy")
```

(This will fail to find anything in our example array, where the "howdy"s are always in the first five characters of a string; i.e. a$(num,1 TO 5).) Use of a slicer increases the search speed,

and it also becomes possible to dedicate different parts of the strings to different kinds of data, such as street names or town names, and search each region separately. This avoids problems such as searching for "Oxford" and finding a string containing "Oxford Road".

You can replace some of the characters in the string being looked for with "#", which means "don't care", as it does for INSTR. The only time " " (literally) is looked for is when it is the first character of the string being looked for. If you have an array in which two items of data - say, surnames and towns - both start at a fixed position within each string, you can search for a combination of items - such as "Brown" and "London" - by looking for a long string containing both of the items, separated by the required number of "#" characters. The following example does this. The first section lets you create a small array of data. In each string, the first 20 characters are reserved for a name, and the last 15 hold a town.

```
110 LET namelen=20,townlen=15
120 DIM d$(10,namelen+townlen)
130 FOR n=1 TO 10
140       INPUT "name? ";n$
150       INPUT "town? ";t$
160    LET d$(n,1 TO namelen)=n$
170    LET d$(n,namelen+1 TO )=t$
180 NEXT n
190 PRINT "array filled"
```

The following section lets you look for a particular combination of name and address. The function STRING$ generates the correct number of "#" characters to separate the start of the name and the start of the town by 20 characters.

```
200 DO
205    INPUT "name? ";n$
210    INPUT "town? ";t$
220 LET s$=n$+STRING$(namelen-LEN n$,"#" )+t$
230 LET loc=INARRAY(d$(1),s$)
240 IF loc=0 THEN
          PRINT "not found"
       ELSE
          PRINT loc;" ";d$(loc)
250 LOOP
```

Note: INARRAY will not work with arrays of more than two dimensions. For example, it will not work on the array A$(8,9,5).

Note: An XVAR can be used to find the exact position within the array element that a string was found at - see XVAR.

USING$(format$,number)

This function converts a number to a string with a fixed number of characters before and after the decimal point. The desired format is specified by a string in which hash signs (#) stand for leading spaces, zeros stand for leading zeros, and either can be used for showing the number of digits after the decimal point:

```
10 FOR n=1 TO 18: LET x=RND*100
20 PRINT x,USING$("###.##°',x): NEXT n
```

Note how much neater the formatted numbers are. By comparing the two columns, you will see that USING$ rounds to the nearest printed digit. Experiment with different strings for formatting. You can use a string variable if you like. Some possible format strings, and their output (with spaces shown by "s") for the number 12.3456 are:

```
"##.#"          12.3
"###.#"         s12.3
"####.##"       ss12.35
"11000.00"      012.35
"00"            12
"$00.00"        $12.35
"110.00"        $..3
```

The second-to-last example shows that it is possible to include leading characters in the format string other than the usual hash and "0". The last example demonstrated an output with a "%" sign which indicates overflow of the specified format. Very large numbers like 1E+8 will always overflow. Very small numbers like lE-8 are handled by conversion to their non-exponent forms automatically. Trailing spaces in the format string are ignored - this is useful if you want to keep an array of format strings.

The advantage of USING$ over the PRINT USING provided by some BASICs is that it can be used with commands like LET to assign numeric data to a fixed part of a string in a known format. SORT can be used on such data when it is part of a string array.

SVAL$(number,characters)

See also: NVAL a$ function

This function converts numbers into 2, 3, 4 or 5-character strings, allowing compact storage of numeric data in a fixed space. NVAL provides the reverse conversion. Coding numbers as strings can be particularly convenient in random-access files, as implemented with MasterDOS, or in string arrays, since you always know how much space a value will take in the file, and you can place numeric information alongside associated string information. This is convenient as it allows each record in a database, or string in an array, to contain all relevant information. See also the USING$ function.

An array containing string-coded numbers can be SORTed according to their value, and it may be more compact than an equivalent numeric array, in which 5 bytes are always used to store each number. (For economy of disk space when saving numeric arrays, see SAVE MODE.)

SVAL$ allows you to set a balance between precision and memory usage. Below are the characteristics of the four length options:

2 Characters: Only whole positive numbers between 0 and 65535 can be encoded. Floating-point numbers are converted to whole numbers where possible. You might consider pre-scaling values before using this option - you can multiply up something like 87.643 to give 8764.3. SVAL$ would use the 8764 part of this to produce a two-character string; subsequent use of NVAL and division by 100 would give 87.64, which is acceptable accuracy for many purposes.

3 Characters: The complete range of positive and negative numbers (about 10E-39 to 10E38) is encoded, but only about the first 5 digits in the number will be correct.

4 Characters: As above, but first 7 digits are usually correct.

5 Characters: As above, but the first 9 digits are usually correct. This is the normal internal numeric precision of the Coup6. Not all the digits are displayable by printing the value, but numbers are stored and manipulated at this level of accuracy.

Normal numbers occupy 5 bytes. This example shows the precision of 2, 3 and 4-character SVAL$s (it is on your disk as "sval").

```
10 MODE 3
20 CSIZE 8,9
30 PRINT " Original 2 chars 3 chars 4 chars"
40 FOR n=1 TO 15
50     LET x=RND*60000
60     LET two$=SVAL$(x,2),three$=SVAL$(x,3),four$=SVAL$(x,4)
70     PRINT x;TAB 11;NVAL two$;
80     PRINT TAB 22;NVAL three$;TAB 33;NVAL four$
90 NEXT n
```

The string result of SVAL$ will often give an "Invalid colour" report if an attempt is made to print it, since it may contain characters equivalent to invalid print control codes.

NVAL a$

This function converts 2-5 character strings created by SVAL$ back into numbers. For example:

```
       PRINT LAVAL abc$
or     LET xyz=LAVAL da$(56 TO 57)
```

See SVAL$ for more details and a working example.

SHIFT$(a$,n)

This function is used to alter the capitalization of text strings, or remove inconvenient control codes from them. The action taken is determined by the number you specify in the brackets.

1 Forces upper-case. E.g. PRINT SHIFT$("Fred Bloggs",1) gives "FRED BLOGGS".
2 Forces lower-case. E.g. PRINT SHIFT$("Fred is 23",2) gives "fred is 23".
3 Forces case reversal. E.g. PRINT SHIFT$("AaBb test",3) gives "aAbB TEST".
4 Normally, replaces control codes with a full stop and replaces characters above 127 with their character code minus 128. However, POKE DVAR 24,1 will make characters above 127 print as block graphics and/or user-defined graphics instead. (BLOCKS 0 or 1 or 2 will modify the results.) POKE DVAR 24,0 for the original effect. The character used to replace control codes can be modified by POKE DVAR 25,(character).

Options 1 and 2 are useful when checking input. For example:

```
10 GET i$
20 IF SHIFT$(i$,1)="Y" THEN PRINT "Erasing...": etc.
```

This will work with either "y" or "Y".

Option 4 is useful when you want to look through memory quickly. For example, to look for hidden messages in the DOS:

```
10 FOR n=DVAR 0 TO DVAR 15000 STEP 256
20     LET a$=MEM$(n TO n+255)
30     PRINT SHIFT$(a$,4)
40 NEXT n
```

You will usually find that PRINT a$ by itself will fail because some of the characters are control codes.

SHIFT$ will only work with strings of 16383 characters or less.

EQU(a$,b$)

This function compares two strings to see if they are the same. Any differences in capitalization are ignored. For example:

```
10 DO: INPUT nm$
20 IF EQU(nm$,"Jones") THEN PRINT "yes": ELSE PRINT "no"
30 LOOP
```

This will say yes to "jONES", "JONES" or "jOnEs". The function is convenient for checking input without forcing it into a particular format.

SOUND COMMANDS

SOUND CLEAR
SOUND CLEAR size
RECORD SOUND TO a$
RECORD SOUND OFF TO a$
RECORD SOUND STOP

MasterBASIC allows sound data produced by the SOUND and PAUSE commands to be stored in strings for playing later under interrupt control. In other words, sound can be set going and then left to continue without further effort on the part of the program. Your disk contains a file called "soundFX" which is basically the sound effects section of the old Coupé demo with a few extra lines. The extra lines are these;

```
10 SOUND CLEAR 2048
20 RECORD SOUND PFF TO snd$
30 PRINT "Preparing sound data..."
40 GO SUB 100
50 RECORD SOUND STOP
60 BLITZ SOUND snd$
```

The first line allocates a storage area 2048 bytes long for interrupt-driven sounds. Space is allocated in 1K (1024-byte) units - the value you supply will be rounded up if need be. This buffer space will be used later by BLITZ SOUND. Line 20 has the effect of making all SOUND and PAUSE commands add special codes to the string snd$, without having their normal effect, since OFF is used in the line. This makes the subroutine execute faster. If you omit the OFF the data will still be added to the string but the SOUNDs and the PAUSEs that determine their length will be acted on as well.

Line 50 tells the Coup6 to stop adding any further data to the string. At this point all the data that would have been sent to the sound chip, plus extra codes denoting any PAUSEs, have been stored in snd$. BLITZ SOUND snd$ transfers this data to the buffer we prepared earlier, and the computer then doles out this data as required. Meanwhile you can edit, load another program, or create a graphic display of some kind.

Fifty times a second, the Coupé looks at the sound buffer. If it finds a PAUSE code it simply sets up a counter showing how many 50ths of a second it can ignore the sound chip for. Otherwise data is sent to the chip until another PAUSE code is found or the data runs out. So if there are no PAUSEs in your original BASIC, there will be no PAUSE codes in the sound buffer and the whole sound will take a fraction of a second. You must use PAUSE!

If your sound buffer is big enough (and it can be up to 256K) hours of sound can be stored. Multiple BLITZ SOUND commands can be used to queue data for the sound chip. If the buffer becomes full, BLITZ SOUND will wait until some of the data is used and enough room becomes available.

To change the sound buffer size, simply use SOUND CLEAR with a

different value. Any data in the buffer will be lost. SOUND CLEAR 0 will delete the buffer and free the memory for other uses. SOUND CLEAR used on its own will clear the buffer and silence the sound chip, without altering the buffer size.
Strings like snd$ can be saved to disk and reloaded later, using SAVE and LOAD "name" DATA snd$. Only lines 10 and 70 of the example need be retained.

Many existing sound routines can be converted for BLITZ SOUND. The main requirement is that timing is controlled by PAUSE, and not by FOR-NEXT loops or the speed of execution of BASIC. It is also a good idea not to generate hundreds of numbers per note, or the sound string will become quite long.

If a sound routine sounds satisfactory before you start recording it, and then sounds odd while recording is turned on (with RECORD SOUND TO a$) don't worry - this is due to the extra delay caused by adding the data to the sound string. The delay will not occur when you use BLITZ SOUND to replay the string.

NEW will silence the sound chip, so avoid it if you want the sound to continue. RUN and CLEAR normally silence the sound chip, but if sound is being generated by BLITZ SOUND this is prevented. RUN and CLEAR turn off RECORD SOUND TO.

The MasterBASIC disk contains more demonstrations of the use of BLITZ SOUND which I hope you will find interesting. Their names all start with "sound".

GRAPHICS

IMPROVED PUT COMMAND

The PUT command built into the Coupe's ROM allows blocks of graphics (usually read from the screen using GRAB) to be placed anywhere on the screen. MasterDOS includes a faster version of PUT which is used automatically in most cases. Since there is a slight chance that this change in speed could make existing programs too fast, I have also included a way of turning off the new PUT and reverting to the ROM version of the command. In the example below, I have used this facility to demonstrate the speed difference with the new version. (This listing is on your disk as "put".)

```
 10 DO
 20    POKE DPEEK XVAR 0,0
 30    PRINT AT 0,10;"ROM "
 40       movcirc
 50    POKE DPEEK XVAR 0,172
 60    PRINT AT 0,10;"MBAS"
 70       movcirc
 80 LOOP

100 DEF PROC movcirc
110    CIRCLE 20,154,18
120    FILL 20,154
130    GRAB cir$,0,173,40,40
140    FOR n=0 TO 100
150       PUT n*2,n+20,cir$
160    NEXT n
170 END PROC
```

The "MBAS" version moves the circle about 40% faster. (This difference was at one time more remarkable, but since then I have improved a general "move memory" routine that speeds up even the ROM version of PUT. The "MBAS" version of MOVCIRC runs about 75% faster than the same procedure does without MasterBASIC.)

For advanced users:

The MasterBASIC version of PUT is unable to handle string data that straddles two of the I6K "pages" that make up the Coup6's RAM. These cases are automatically handled by the ROM version of the command. This could make execution speed unpredictable. If this worries you, you can check the location in memory of the strings you are using, by e.g. PRINT LENGTH (O,circ$) and create dummy strings before doing a GRAB if the string straddles a page boundary, or you can use POKE to place the strings at known positions above RAMTOP, and use for example:

```
PUT x,y,MEM$(65536 TO 65826).
```

Sometimes it is desirable to save the area of screen about to be overwritten by PUT, for example in designing a pull-down menu system. This would usually be done using GRAB, but MasterBasic provides a faster way of doing things with a combination PUT and GRAB that swaps the data in the string being PUT with what is

already on the screen. Doing this again will reverse the swap and restore the original screen, as shown here:

(This example is on your disk as "put grabl".)

```
 10 PAPER 1
 20 CLS
 30 CIRCLE 16,155,15
 40 GRAB bl$,0,173,32,48
 50 PAPER 15
 60 CLS
130 GRAB m$,0,173,32,48
140 CLS #
150 LIST
160 DO
170    PUT GRAB 10,160,b1$,m$
180       PAUSE 50
190 LOOP
```

The first four lines give us a blue block with a circle on it in bl$, and an ordinary PUT can place it anywhere on the screen - try PUT 100,100,b1$. The next three lines give us a block of the same size in m$. This block will be used as a "mask" to determine exactly which parts of bl$ will be swapped with the screen. PUT GRAB has to use a mask, so we need one even if we want to use all of bl$. All bits which are 1 in the mask will let the corresponding bit in bl$ be swapped with the screen, and all bits which are 0 in the mask will prevent their corresponding bits being swapped. In the example the mask has the colour PEN 15; since 15 is 1111 in binary, the mask is solid is and all of bl$ is used.

We can modify the program to use an irregular mask. The mask will be partly PEN 0 and partly PEN 15. Add or modify these lines, or load "put grab2" from your disk.

```
 50 PAPER 0
 60 CLS
 70 PEN 15
 80 PLOT 0,155
 90 DRAW 30,10
100 DRAW -10,-40
110 DRAW TO 0,155,-1
120 FILL PEN 15;6,155
```

You may find it interesting to PUT 100,100,m$ to see the shape of the mask.

You can get interesting effects by making the mask other than PEN 0 and PEN 15. Try altering the PEN specified in line 120 and/or line 50 - this will give a "transparent" mask, because some of the bits making up pixels on the screen are combined with bits from bl$, giving a composite image with modified colours.

Finally, I'd like to remind you that most of the principles outlined above work with the normal PUT. If you remove the GRAB from line 170, bl$ will still be PUT, but it cannot be removed by a second PUT.

COPY SCREEN number TO number

This command copies whatever is on the first screen to the second screen. For example (on your disk as "copyscr"):

```
10 MODE 4
20 CLOSE SCREEN 2: OPEN SCREEN 2,4
30 CIRCLE 128,88,70: FILL 128,88
40 SCREEN 2: PAUSE 50
50 COPY SCREEN 1 TO 2: PAUSE 50
60 SCREEN 1
```

The screen is copied very quickly in the example above, because both source and destination screens are in MODE 4, and no conversion work needs to be done as the image is copied.

Now alter line 20 to finish with OPEN SCREEN ,2,2 or 2,1 and run the program again. This illustrates one of the main purposes of COPY SCREEN - the conversion of an image created using commands specific to MODEs 3 and 4, such as FILL, PUT or SCROLL, to a less memory-hungry MODE 1 or 2 format. The results can then be used for animation or whatever. Obviously conversion from the high colour resolution of MODEs 3 and 4 to MODE 1 or 2 will often result in imperfections due to attribute clashes.

In contrast, copying a MODE 1 or 2 screen to MODE 4 should always produce a perfect result, apart from the lack of character-based FLASH. This allows easy conversion of Spectrum screens to MODE 4 for manipulation by GRAB, PUT, FILL etc.

Faster Animation with POKE address,a$

If you already know how to animate graphics by POKEing a string onto the screen, all you need to know is that it now works faster than before. However, there is an example on your disk for those who have not used this technique before, called "spinner".

The general idea is to create a series of screens that differ by a small amount, saving them as you go into an array using SCRAD to find the screen (see SCRAD in this section) and MEM$ to handle part of memory as a string. Then a series of POKEs is used to replay the sequence of screens. SPINNER uses MODE 2 because one third of the screen can be stored in only 2K, if you ignore colour information, and the memory is arranged in a way that gives smoother results than in MODE 1. The program uses 48 frames to achieve its effect. The spinning objects seem to pass behind two of the coloured columns because this MODE uses an attribute-mapped colour system and both PEN and PAPER colours are the same in the area of these columns, making the objects invisible there.

ALTER DISPLAY number TO number LINE n

See also: DISPLAY command in the Coupe User's Guide.

This command allows parts of two SCREENs to be displayed at once. The display alters at the specified line (y coordinate) on the screen. The two screens do not have to be in the same MODE, so it is possible to mix MODE 4 graphics and MODE 3 text or MODE 1 FLASHing. Here is an example, on disk as "alt displ".

```
10 MODE 4
20 SCREEN 1: CLS #: PALETTE 3,127: LIST
30 CLOSE SCREEN 2: OPEN SCREEN 2,3
40 SCREEN 2: PAPER 3: PEN 0
50 WINDOW 0,63,10,18
60 CLS: LIST
70 ALTER DISPLAY 1 TO 2 LINE 84
```

These lines make SCREEN 1 a MODE 4 screen with a listing of the program on it, and SCREEN 2 a MODE 3 screen with an inverse listing of the program on it, restricted to the lower part of the screen using WINDOW. Line 70 means that only the top half of SCREEN 1 (down to y-coordinate 84) is shown before the display switches to the lower part of SCREEN 2. If you type:

```
SCREEN 1
```

then that screen will become the current screen for LIST, PRINT or graphics - but you will have to type blind because you cannot see the editing area for that screen! Such things are best handled within the program. Type:

```
SCREEN 2
```

to get back to a usable edit area. Notice that ALTER DISPLAY does not alter the current SCREEN that output is sent to, but rather which screen is displayed, and where. In fact the current screen does not even have to be one of the displayed screens.

Try removing line 50 of the program - you will have to LIST several times before the program comes into view.

Note: The PALETTE command in line 20 is needed because the palette used for the whole screen is that of the top part (SCREEN 1 here) and I wanted PEN 3 to be bright white to suit an inverse display for the lower part of the screen in MODE 3. It is important to use any PALETTE commands before ALTER DISPLAY is turned on. PALETTE acts differently according to whether the current SCREEN is displayed or not, and will act unpredictably while ALTER DISPLAY is running.

ALTER DISPLAY OFF turns off the display switching, as does CLS #. CLS, ZAP, POW, ZOOM, BOOM and tape and disk commands will cause flickering because the interrupts used by ALTER DISPLAY are turned off temporarily. CLS 1, which clears just the current WINDOW, does not cause this problem.

Your disk contains a longer example under the name "alt disp2".

BLOCKS 2

The ROM supports:

BLOCKS 0 - Turn block graphics off and display CHR$ 128-168 as user-defined graphics.
BLOCKS 1 - Turn block graphics on, only CHR$ 144-168 are shown as user-defined graphics.

MasterBASIC supports in addition:

BLOCKS 2 - Turn block graphics off and swap the entire set of user-defined graphics with the set stored within MasterBASIC. This is initially an IBM standard foreign character set. Using BLOCKS 1 now will re-enable the block graphics but leave CHR$ 144-168 as user-defined graphics. Using BLOCKS 0 will turn off the block graphics and swap the user-defined graphics with the original, stored within MasterBASIC.

To replace the foreign character set with another, you could do this, assuming the new set is a CODE file:

```
BLOCKS 2
LOAD "newset" CODE UDG CHR$ 128
BLOCKS 0
SAVE BOOT "filename"
```

This SAVEs MasterDOS with the new character set included within it. Make sure the file is saved into the first position in the disk directory.

The lower- and upper-case cursor characters are normally CHR$ 128 and 129, but to prevent the cursor changing when a new set of characters is switched in by BLOCKS 2, the cursors are temporarily re-defined as CHR$ 169 and 170, the patterns for which are stored separately.

This example is on your disk as "blocks"

```
10 FOR bl=0 TO 2
20     BLOCKS bl
30     FOR c=128 TO 170
40         PRINT CHR$ c;
50     NEXT c
60 PRINT
70 PRINT
80 NEXT bl
```

CLS *

This command is equivalent to PEN 0: PAPER 15: BORDER 15: CLS. Many users seem to use their computers in this black-on-white mode, and CLS * is provided for their convenience. CLS #, or key F6, is a quick way to get back to the normal white-on-black display.

CSIZE Improvements

SAM BASIC allows some control of character size using the CSIZE command. Single and double-height characters are available in any screen mode, and MODE 3 allows 6-pixel or 8-pixel wide characters to be displayed, using e.g. CSIZE 6,9 or CSIZE 8,9.

MasterBASIC allows a much more flexible choice of character size, for example:

```
CSIZE 16,9
CSIZE 16,18
CSIZE 8,32
CSIZE 40,40
```

The width should be divisible by 8 unless you are working in MODE 3. The height can be 6-173. INT(height/8) gives the height multiplication factor, double, triple, quadruple, etc. The remainder is used to space out the rows of characters. The maximum width is 248 and the maximum height is 173 pixels. It is possible to specify CSIZEs that would make it impossible to type any command in, so as a safety measure the CSIZE reverts to something sensible when the program stops. Try:

```
10 CSIZE 240,160: PRINT "A"
```

Indentation of program lines is turned off when the character width exceeds 40 pixels.

In MODE 3, if the width you specify is divisible by eight, output is based on magnified 8-pixel wide characters, so CSIZE 16,9 gives 32 characters per line and CSIZE 32,18 gives 16. If the width is not divisible by 8 it is divided by 6 instead to give a multiplication factor for 6-pixel wide characters. CSIZE 12,9 gives a rather nice 42 characters per line mode, but many other possibilities are not very pretty.

CSIZE combined with the DUMP command makes it easy to produce large banners printed sideways on the paper.

SCRAD

This function gives the address of the start of the current screen. It is useful when you want to POKE or LOAD to the screen and you do not know where it is. This function, or something that does the same job, should be used if you want direct access to the screen to work correctly in programs written for both 256K and 512K Coup6s. Here are some examples:

```
POKE SCRAD,1,2,3,4,5,6
LOAD "name" CODE SCRAD
PRINT SCRAD
OPEN SCREEN 2,4: SCREEN 2: PRINT SCRAD
```

PRINTERS and SCREEN DUMPS

INTERRUPT-DRIVEN PRINTING

LPRINT CLEAR
LPRINT CLEAR size

Printers work considerably more slowly than computers, which means that you normally have to sit and wait for a long program to LLIST, even though the computer's part of the job could be done very quickly. Screen dumps require more work from both printer and computer, but the same principle applies. MasterBASIC, however, is able to store output that is intended for the printer and feed it to the printer later, when the printer is able to take it. This is called Interrupt-Driven Printing, because the computer feeds data to the printer 50 times per second, at every frame interrupt. You can switch the printer temporarily off line during printing if you want, and you can use NEW or LOAD another program while the printer is running. You must avoid using LPRINT CLEAR or LPRINT MODE until the printer stops, or you will lose the data. The data for many printer commands can be queued. For example:

```
LPRINT CLEAR 50000
LOAD "any prog": LLIST: LPRINT: CIRCLE 128,88,60: DUMP 1
```

It is probably a good idea to save a valued screen image to disk before doing a DUMP - you may find the printer jams after you have cleared the screen!

To use this facility, you must first of all specify the size of the storage area you want to reserve for the printer data, using LPRINT CLEAR. This "buffer space" can be anything up to 256K, depending on the free memory available. Space is allocated in 1K units. The number you supply is rounded up, so LPRINT CLEAR 2048 and LPRINT CLEAR 1025 both reserved 2048 bytes (2K). Large screen dumps may need 55K or more for a complete dump to fit in memory. If the buffer becomes full, the computer will wait for the printer to deal with some of the data before finishing the LLIST, DUMP or LPRINT.

LPRINT CLEAR 0 has the special function of turning off interrupt-driven printing and freeing the buffer space for other uses.

LPRINT CLEAR used on its own clears the existing buffer space and stops transmission of data to the printer.

Large dumps take a lot of computer time, even with interrupt-driven printing, but you will have control of your machine back much faster than before.

This facility should work with well-behaved programs (including word-processors) that output to the printer via channel "P" (like LPRINT) or channel "B" and do not over-write MasterBASIC or the buffer space. For example, Tasword II works, as do the SDC SAM Supplement printing option and SAM Scratch.

SERIAL INPUT AND OUTPUT

LPRINT MODE 1 - use parallel printer
LPRINT MODE 2 - use serial printer

The serial printer driver program that was previously required to use the serial interface is now part of MasterBASIC. You can switch from parallel to serial output simply by entering LPRINT MODE 2. The baud rate, number of data bits and number of stop bits come from internal MasterBASIC variables that you can POKE - see XVAR. The new values become effective when you next select LPRINT MODE 2. You can also POKE the program so that it starts up in LPRINT MODE 2 automatically - see XVAR.

Any use of the LPRINT MODE command will clear and close any printer buffer space you may have allocated, but subsequent use of LPRINT CLEAR (space) will allow either parallel or serial output to be interrupt-driven. Serial output can of course be sent to a suitably equipped computer instead of a printer.

Either setting of LPRINT MODE allows serial input. For example:

```
10 CLOSE #5: OPEN #5;"b"
20 DO
30 INPUT #5;a$
40 PRINT a$
50 LOOP
```

For increased speed, if you have MasterDOS, replace line 30 with LET a$=INP$(#5,0). Note: This example will not work unless you are transmitting something to the Coupé on the serial interface!

DUMP

A wide range of screen dump options are provided by MasterBASIC, including those available in the separate DUMP utility provided with SAMDOS 2.0. This utility can still be used, but it is no longer required. The five main DUMP variants available are:

```
DUMP 1 - small shaded dump.
DUMP 2 - medium shaded dump.
DUMP 3 - large shaded dump (sideways).
DUMP 4 - medium non-shaded dump. Like DUMP with utility.
DUMP 5 - text dump. Like DUMP CHR$ with utility.
```

The first three dumps all scan the screen palette, determine the brightness of the colours in use, and use a pattern of dots of approximately the correct darkness when reproducing each colour on the printer. If the screen is fairly dark, you may want to save wear on your printer ribbon by using DUMP INVERSE 1, 2 or 3, which gives a dump with dark colours light and vice versa.

The number 1, 2 or 3 actually specifies the width magnification of the dump; the height magnification is assumed to be the same unless you specify differently by including a second number. E.g.

```
DUMP 1,2 - single width, double height
DUMP 3,1 - treble width, single height
```

DUMP works in all screen modes, but MODE 3 is slightly different from the others because it has 512 pixels across the screen instead of 256. This gives screen dumps which are twice as wide as in the other MODEs, which you may not always want. DUMP 1,2 or DUMP 2,3 can be used to reduce the width relative to the height. If you are dumping a screen of text, it is often a good idea to print it in PEN 15, and use DUMP INVERSE (size).

For DUMPs 1 to 3, POKEing the relevant XVAR will allow you to modify the number of times the printer strikes the paper, the control codes used to control the graphics dumps, and the dumped area. You can produce mirror-image DUMPs, or change their orientation. (The normal setting produces upright dumps unless the width would be too big to fit the printer.) See XVAR.

The next two DUMPs, 4 and 5, are essentially the same as DUMP and DUMP CHR$ supported by the utility program supplied with SAMDOS.

DUMP 4 is an unshaded dump in which anything which is the current PAPER colour comes out white, and anything else comes out black. It is most suitable for something like a graph or line drawing, where you want a clear black-on-white output. The User's Guide, page 176, documents some SVARs that can be used to control the size of this DUMP. (This DUMP, and the equivalent of DUMP 5, were at one time built into the ROM, but had to be discarded because of lack of room.)

All the DUMPs can be stopped immediately by ESC, but this may leave the printer in bit-image mode and require you to reset it. It is better to hold down the space bar, which will stop the DUMP but leave the printer in the normal text mode

DUMP 5 is a text dump. Instead of sending a picture of the screen to the printer, the computer reads any characters it can recognize from the current screen window and sends them as single bytes, much faster than a graphic screen dump. Any user- defined graphics that are recognized will be converted to the appropriate single character code. Unrecognized characters are printed as spaces. The colour the characters appear in on the screen is irrelevant. The ROM's SCREEN$ routine is used - the colour of the top left-hand pixel is assumed to be the background colour and an uncoloured version is compared with the character set in an attempt to match the character. Only single-height characters in one of the CSIZEs provided by the ROM can be recognized.

One possible problem with DUMP 5 is that the number of characters across a MODE 3 screen can exceed the width your printer can handle, making the text dump look odd. Usually it, will look double-spaced because the first 80 characters will fill. the line and then a few more will be printed on the next line down. To get round this, you can tell the printer to produce smaller characters, so that more fit on a line. I use:

```
OPEN #4;"b": PRINT #4;CHR$ 15;
```

On my Epson this switches to a about 132 characters per line. the WINDOW size:

```
10 MODE 3: CSIZE 6,9
20 WINDOW 0,79,0,18: REM 30 LIST
40 PRINT STRING$(80,"MB") 60 DUMP 5
```

POUND (£) and HASH (#) characters

Many users will be aware that a program like the one below can look slightly odd when LLISTed

```
10 PRINT #3;"£10.00"
```

Try it, then CALL 0 to reset the machine and try it again without MasterBASIC loaded. With a little luck the LLIST with MasterBASIC loaded will look the same as a LIST. Without the program loaded, you may see the hash symbol replaced by a Pound sign, and/or the Pound sign replaced by a single quote mark. These problems happen with many printers because the Coupe (and the Spectrum) use a code of 96 for Pound whereas printers often use this for a single quote mark. Also, if the printer is set up for an English character set, the code used for hash (35) on the computer will often print as a Pound sign. If you select another character set on the printer, the hash becomes printable but the possibility of getting a Pound by printing CHR$ 35 is lost.

MasterBASIC deals with this problem by converting any hash or Pound signs LPRINTed or LLISTed into a sequence of characters that switch between different character sets on the printer. The initial set-up is suitable for use with an Epson compatible printer, but can be modified - see XVARs.

NEW TIMING FACILITIES WITH MASTERDOS AND THE SAMBUS

TIME +
TIME -
TICS function

MasterDOS provides the commands TIME and DATE to set and read the time and date on the clock/calendar built into the SAMBus, and it also allows use of the TIME$ and DATE$ functions. Date-stamping of files is done automatically.

MasterBASIC provides extra support for the clock, provided you have got the SAMBus and MasterDOS. The new features are intended for those who want to time things accurately.

It is of course possible to work out elapsed time from two readings of the clock. For example, part of a keyboard speed test might look like this (the file is on your disk as "tics"):

```
10 LET q=RND(899)+100
20 LET t$=TIME$
30 INPUT ("Enter: ;q;" ");a
40 IF a<>q THEN PRINT "Wrong!"
50 PRINT TIME$'t$
```

The information displayed by line 50 is sufficient to work out how long you took, to 1 second accuracy, but it is a bit awkward, especially near the end of an hour or minute. Hence the need for the new function, TICS.

PRINT TICS will give you the number of seconds elapsed in the month so far, so it can give a value between 0 and 2678399 (31*24*60*60-1). It will restart at zero at midnight on the last day of the month. Try modifying the example above by including:

```
20 LET t=TICS
50 PRINT TICS-t
```

This gives the elapsed time very easily, but it is still not very accurate. It would be nice if the clock gave time in hundredths of a second, but it doesn't. However, there IS a special high-speed test mode which makes the clock run 5416.3 times faster than normal. This can be turned on and off by TIME + and TIME -.

```
100 TIME +
110 FOR n=1 TO 2000
120    PRINT AT 0,0;TIME$'DATE$
130 NEXT n
140 TIME -
```

Arrgh! There are your carefully set time and date changed to 00:00:00 and O1/00/00 and then sent whirling out of control, apparently. Fortunately, MasterBASIC saves the real time and date before turning on this high-speed mode. When you change back to normal using TIME -, the real value is updated by the time elapsed in high-speed mode (divided by 5416.3 to get it right) and replaced. Therefore the time and date should be correct unless you leave the clock/calendar in the TIME + mode

continuously formore than one month of "fast" time (about 8 minutes of real time), which would exceed Masterbasic's abilities to correct. Even if you leave the TIME + mode on for longer than this, the date should be correct on return to normal mode unless the real time was close to midnight. If you turn the computer off with TIME + selected, the clock will stay in high-speed mode until you reBOOT. The date and time will be rubbish.

TICS works in the faster TIME + mode, and you may anticipate that you will be able to take two readings and divide the difference by the magic number 5416.3, but in fact life is even easier than this, because MasterBASIC does the division for you! Try this:

```
10 TIME +
20 DO
30    PRINT AT 0,0;TICS;" "
40 LOOP
```

You get a floating-point number accurate to about 0.0002 of a second. This is set to zero when you select TIME +. The first example in this section could make use of these modified lines:

```
20 TIME +
    LET t=TICS

50 PRINT USING$("##.##",TICS-t)
    TIME -
```

USING$ is used to format the result to two decimal places. It is a good idea to get into the habit of adding TIME - after a timing has been performed, so that the clock time and date stay correct.

HIDE TO line number

One of the best features of SAM BASIC is the ability to define new commands using DEF PROC. Many users accumulate a set of procedures that they want to use frequently, either in many of their programs, or direct from the keyboard as "utility" commands. These may be loaded at the start of the day, or always included as part of the current program, but there are one or two disadvantages. For example, the procedures are lost if NEW or LOAD are used, and they may get in the way in listings. For this reason Master-BASIC allows some lines of a program to be HIDDEN. Such lines are safe from NEW and LOAD and cannot be LISTed.

The line numbers in the hidden section become irrelevant and there is no conflict with line numbers in the normal program, even if they are the same. This means that you cannot use GO TO, GO SUB or any command that depends on line numbers to refer to part of the hidden section. You must use procedure or DEF FN names to make use of the hidden code. Here is a simple example, which is included on your disk as "hide":

```
10 DEF PROC mymode
20 MODE 3: CLS #1: PALETTE 0,107: PALETTE 3,0: CSIZE 8,9
30 END PROC
40 DEF FN pal(c)=PEEK(&55d8+c): REM read PALETTE entry
50 REM still here
```

Now type: HIDE TO 40 and all the lines except 50 will be hidden. Typing: mymode or: PRINT FN pal(3) will still work, however, even after NEW or LOAD. You can add some new lines numbered 10 to 40 without any problems - at least, until you decide you want to alter the procedure MYMODE. To do this, type: HIDE TO 0. This has the special function of revealing any hidden lines. If you added some lines as I suggested earlier, you will notice that you now have several lines with the same line number and probably two with a current line cursor! (The top one is the "real" one.) This can make editing tricky, and sometimes you can get a prolonged automatic LIST. You can re-HIDE the lines with HIDE TO 40 or use a combination of manual renumbering and RENUM to allow certain lines to be edited. However, it is probably better to use NEW before using HIDE TO 0 to reveal and edit the lines. A problem can still arise when you un-HIDE several accumulated blocks of hidden lines. Life will be easier if you add new hidden lines with lowish but increasing line numbers. One problem you may run into is the use of DATA in hidden lines. This DATA will not normally be found by READ, but you can force READ to look at the right place by POKEing "address of DATA" with "address of next line", like this:

```
10 DEF PROC reader
20 POKE SVAR 138,PEEK SVAR 156: DPOKE SVAR 139,DPEEK SVAR
       157+4 30 DATA "hello"
40 READ a$: PRINT a$
50 END PROC
```

Note: Line 20 must be a single line. Hidden lines will not be SAVEd as part of the current program.

NEW EXIT COMMANDS

MasterBASIC includes three minor but useful new program control keywords. These are EXIT PROC, EXIT FOR and EXIT DO. They provide neat ways of leaving procedures, FOR-NEXT loops or DO loops.

EXIT PROC

You can leave a procedure prematurely by GO TO the END PROC, but in modern programming GO TO is discouraged because it can lead to programs that are hard to read, understand and modify. You could use an extra END PROC instead, but the indentation of the program would suffer, and the computer would no longer be able to automatically avoid executing the procedure if it ran into it in the program. EXIT PROC does not have these disadvantages, and it will work correctly even if you use it within a loop. For example (this example is on your disk as "exit"):

```
 10 waitspace
100 DEF PROC waitspace 110 LET t=0
120       DO
130       LET t=t+1
140       IF INKEY$=" " THEN EXIT PROC
150    LOOP UNTIL t=1000
160    PRINT "time is up!"
170 END PROC
```

EXIT DO

You can already branch out of a DO loop using EXIT IF (condition). However, users have found themselves writing EXIT IF 1 (which is always true) quite often, and this looks a bit odd and isn't very elegant. Now you can use EXIT DO instead, e.g.:

```
210 LET accum$=""
220 DO
230    INPUT wd$
240    IF wd$="end" THEN
              PRINT "Terminated"
              EXIT DO
250    LET accum$=accum$+wd$+CHR$ 13
260    LOOP
270    PRINT accum$
```

EXIT DO will leave the DO-loop by jumping to the statement after the appropriate LOOP. It will jump over any nested DO-loops if necessary - for example, it will still jump to line 270 even if you add:

```
255 DO UNTIL INKEY$=""
256 LOOP
```

EXIT FOR

SAM BASIC does not offer any structured way to jump out of a FOR-NEXT loop. You either have to set the FOR variable to a value outside the limit for the loop, or GO TO the line after the NEXT. EXIT FOR is more convenient and does not depend on line numbers.

```
310 FOR n=1 TO 20
320    PRINT n;
330    PAUSE 50
340 IF INKEY$=" " THEN EXIT FOR 350 PRINT " hello"
360 NEXT n
370 PRINT "Out of loop"
```

EXIT FOR has no effect on the value of the FOR variable.

DOS ENHANCEMENTS

FILE COMPRESSION with SAVE MODE

It seems to be some sort of rule in computing that however much disk space you have, it gets filled. This is particularly true if you save lots of screen images, large arrays or big sections of the computer's memory. To enable you to pack more files onto a disk, MasterBASIC provides a file compression facility. Once this is turned on, all SCREEN, CODE, String Array and Number Array files are compressed as they are saved. A compressed file will be automatically expanded again on reloading. The options available are controlled by SAVE MODE:

SAVE MODE 1

SAVE normally, no compression.

SAVE MODE 2

Compress SCREEN, CODE and Array files. The method used is fairly fast but it requires at least one 16K page of memory as a working area during both SAVE and LOAD. If there is no page free, the screen will be used - this will corrupt the display. (If you are saving' a SCREEN file, it will be saved correctly and can be reloaded by freeing at least one page before loading.)

The amount of compression achieved can be gauged by looking at the sectors used as shown in a directory, compared with the normal figure. Compression is best where there are repeats of the same value in the data - for example, simple screens, string arrays with lots of trailing spaces in most strings, or CODE files with blank areas. Numeric arrays are highly compressible if they are mainly filled with whole numbers. No attempt is made to compress BASIC programs because it is not usually worthwhile.

SAVE MODE 3

As above, except that a slower but more intelligent routine is used to compress SCREEN$ files. This can give worthwhile reductions in file size compared with SAVE MODE 2, particularly for complex screens. Another advantage of this alternative screen compression method is that it does not require a spare memory page. CODE and Array files are handled just as they are by SAVE MODE 2.

The SAVE MODE setting is completely irrelevant when you come to reload a file - any required expansion is handled automatically, provided MasterBASIC is loaded. (If it's not, you will just get a scrambled mess, instead of a file!)

The compressed status of a file is not shown in the DIRectory, but it can be obtained using the FSTAT function if required, provided you have MasterDOS. See FSTAT in this manual.

SAVING THE DOS/ MASTERBASIC FILE

SAVE BOOT "filename"

This command SAVEs MasterBASIC and whatever DOS it is associated with as one file to disk. This allows you to modify the program by altering the DVARs or XVARs and then SAVE it, without worrying about exactly where it is in memory, or how long it is. For the program to be BOOTable later on, it must be the first program in the disk directory. Make sure that position is free before using SAVE BOOT, or use the same name as the first program so that you overwrite it. Otherwise, the name you use is not important. Personally I'd start it with "MD+MBAS..." or "SD+MBAS..." according to which DOS is part of the file. For example:

```
SAVE BOOT "MD+MBAS1-a"
```

The file created by SAVE BOOT is a special CODE file consisting of your DOS combined with MasterBASIC. You can do the usual things with it that you could do with any CODE file.

Faster MERGE with MERGE *"filename"

The MERGE command built into the ROM is very flexible. It can handle MERGEing of random assortments of lines and variables from disk or tape with the program in memory. Being able to do this means handling lines one-by-one and shuffling a lot of memory around, a process that can take a long time if the program is large. This is annoying when the most common case of MERGE is in fact a simple merging of a block of lines, which can be done quite quickly. MasterBASIC allows the use of MERGE * for these common cases. Any existing lines in the area spanned by the line numbers of the merged program are obliterated. For example, if the program in memory is numbered from 10 to 200 in steps of 5, and the merged program is numbered from 100 to 150 in steps of 10, all the lines in the original program between 100 and 150 inclusive will be replaced by the merged program. Any variables with the merged program will be lost.

Note: Some people use MERGE to stop a program auto-running, but if you want to do this, it can be faster to use LOAD "filename" LINE 65432 which forces the program to start at a non-existent line number.

PROTECTING CODE FILES FROM BEING STOPPED

It is normally possible to prevent auto-executing CODE files from starting by using MERGE "name" CODE. However a CODE file can be prevented from being stopped in this way by using:

```
SAVE CHR$ 2+"name" CODE start,length,execute address
```

The CHR$ 2 does not become part of the name, but it turns on a protection system.

FORMAT Enhancement

The first version of the MasterDOS FORMAT command formatted disks without asking (y/n) if the disk had never been formatted or was unreadable, and asked: FORMAT "disk name" (y/n) if the disk was readable. Unfortunately this could fail when the disk name was corrupt, because the "name" could consist of illegal control code combinations. Therefore the command was changed before MasterDOS was released, so that it just asked: FORMAT "" (y/n) when the disk was readable. MasterBASIC reinstates the original concept and prevents any problems with odd disk names by filtering out any illegal characters. This works even if you have merged MasterBASIC with lowly SAMDOS.

Faster DIR (drive number)

A command such as DIR 1 now reads the directory a track at a time, rather than a sector at a time. This gives greater speed when reading long directories.

MASTERDOS ENHANCEMENTS

Speed Improvements to RAM Disks.

To LOAD a file, the DOS normally has to look through the directory for the file entry before loading can start. If the file is, say, the 40th file on the disk, then IOK of directory data must be read before it is found. On a "real" disk drive, as opposed to a RAM Disk, this takes about 0.4 of a second, once the disk is spinning - a trivial time compared with the overall loading time. Using the RAM Disk, IOK of directory data still has to be read, but this takes a very significant part of the loading time when you are trying to do animations by loading, say, 6K of data for each frame.

However, LOAD (file number) has the potential to go straight to the correct directory entry without reading all the preceding directory entries, because there is a simple relationship between the file number and the location of the directory entry on the disk. MasterBASIC modifies the DOS to take advantage of this, giving a worthwhile improvement to RAM Disk loading speed when LOAD (filenumber) is used. I would particularly recommend it for animations, where you can use something like:

```
10 FOR frame=l TO 20
20 LOAD frame
30 NEXT frame
```

Even when LOAD (filenumber) is not used, improvements to the loading code mean that some files load from RAM Disk faster than before. The speed of READ AT from RAM Disk has also been improved, by a factor of about two.

ALTER DEVICE drive TO drive

MasterDOS allows you to change which physical disk drive is referred to by a given "logical" drive number employed by the user. This is done by POKEs to a table at DVAR 111. MasterBASIC allows the same thing to be done in a more comprehensible manner, using ALTER DEVICE. The first drive number is the "logical" drive number you normally type, and the second one is the number of the "real" drive. So:

```
ALTER DEVICE 1 TO 3
```

will make commands like:

```
      DIR 1
or:   LOAD"dl:name"
or:   DEVICE dl: LOAD "name"
```

work with drive 3 (a RAM Disk). This is useful for making software run from RAMdisk without any program changes.

To make drive 2 act as if it were drive 1, and vice versa, you can do this:

```
ALTER DEVICE 2 TO 1: ALTER DEVICE 1 TO 2
```

OPEN BLOCKS n - Reserving serial file buffers.

MasterDOS allows you to OPEN serial or random-access disk files. These files each create a file buffer in a special area located before the BASIC program. One consequence of this is that the program is moved as the files are OPENed and CLOSEd and this can cause problems because addresses used by any procedures within the program are out of date. Another is that any variable addresses found using LENGTH(0,name) will be out of date. It is possible to get round this by forcing the program to update itself when needed by DELETEing a non-existent line. However, this is inelegant and can be a little slow in long programs. Therefore MasterBASIC allows the use of OPEN BLOCKS to create space for a given number of file buffers. The BASIC program will be updated and will not move after this unless you OPEN a larger number of files at once. The example below shows this.

```
 10 PRINT FREE
 20 OPEN BLOCKS 2
 30 PRINT FREE
 40 OPEN #5;"TEMPI" OUT
 50 PRINT FREE
 60 OPEN #8;"TEMP2" OUT
 70 PRINT FREE
 80 OPEN #6;"TEMP3" OUT
 90 PRINT FREE
100 CLEAR #
110 PRINT FREE
```

The initial amount of free memory is reduced by the space needed for two file buffers when OPEN BLOCKS 2 is executed. This value does not change when two files are OPENed, because their file buffers are created in the reserved area. The third OPEN causes FREE to fall because a new area had to be created before the BASIC program. FREE will not change when files in the reserved area are CLOSEd, but CLEAR # deletes the entire area, as well as any file buffers outside it, restoring FREE to the original value.

MasterBASIC also modifies MasterDOS so that it does an automatic program update if OPEN #, CLOSE # or CLEAR # move the program.

Any temporary files which are OPENed by e.g. MOVE "auto" TO #2 will be CLOSEd automatically the next time you access a different disk.

Alternative Syntax for COPY, RENAME, BACKUP and MOVE

A slightly shorter syntax is now allowed as an alternative to COPY/RENAME/BACKUP/MOVE "name" TO "name". The TO keyword can be replaced by a comma. For example, you can use:

```
COPY "name","othername"
RENAME "prog","newname"
BACKUP "dl","d2"
MOVE "name","other"
```

FSTAT extensions

e.g. PRINT FSTAT("m*",7)

If you have merged MasterBASIC with MasterDOS, the FSTAT function provided by MasterDOS is expanded to support more than the original four options. Here is the extended list:

1. File number in directory, or 0 (not found) or -1 (no disk).
2. File length in bytes, excluding any header. Gives expanded length for compressed files saved with SAVE MODE 2 or 3.
3. File type - see MasterDOS manual. (Note: type 5 is SNP 48K.)
4. File type plus 64 if protected, plus 128 if hidden.
5. Start address - usually only important if CODE.
6. Auto-start line for BASIC, execute address for CODE. Zero if none, or another file type.
7. The file date, as a number such as 231291 (23/12/91) or 30754 (3/07/54). Zero if the file is undated.
8. File flags. Individual bits have the following meanings when set to 1:

   Bit 0 - Not used.
   Bit 1 - If the file is a CODE file, it cannot be stopped by MERGE
   Bit 2 - The file is compressed.
   Bit 3 - The file is a SAVE MODE 3 SCREEN$ file.

DIR$ extension

The MasterDOS DIR$ function has been extended to allow ALL files on a disk to be listed, including those in subdirectories. Follow the name string with a question mark. For example:

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

INP$ extension

The MasterDOS INP$ function has been extended. As well as allowing N characters to be input from a stream using e.g.:

```
LET a$=INP$(#stream,N)
```

you can use a special value of zero in place of the number of characters. This reads in all characters until a carriage return is found, like INPUT. So these lines have the same action:

```
LET a$=INP$(#5,0) INPUT #5;a$
```

The advantages that INP$ offers are that it does not clear the lower part of the screen, it does not beep with each input character even if you have POKEd SVAR 569 with a value, and it is much faster than INPUT. Often loops using INP$ will run several times as fast as loops using INPUT #.

SPECIAL PURPOSE FEATURES

LOCN Function - Searching Memory

LOCN(start,length,a$)
LOCN(start,length,a$,ABS)

This function searches a specified part of memory for a desired string. "Match anything" characters (hashes) can be included in the string, as with INSTR and INARRAY. The first form shown above does a search which is case-insensitive, so that you can search for e.g. "text" or "TEXT" or "Text" at the same time. The search speed is about 90K/second, unless there are frequent occurrences of the first character of the target string. The example below searches from the address 32768, for up to 200K, looking for "test". It is on your disk as "locn".

```
10 POKE 123456,"test"
20 PRINT LOCN(32768,200*1024,"test")
```

You may prefer to search for a string which is absolutely as specified, in which case a final ABS is included in the brackets. This is the form to use when searching for, say, a particular section of machine code. As well as being absolutely specific, the search speed increases to over 200K/second. The example below will deal with multiple instances of a desired string. It also shows that including the Basic program area in a search can be confusing, since the string you are looking for may exist in the program, variables area, or special buffer areas, simply because you are looking for it!

```
110 LET start=16383,tar$=CHR$ 205+CHR$ 65+CHR$ 66
120 POKE 33000,tar$ '
130 DO
140   LET start=LOCN(start+1,40000,tar$,ABS )
150 EXIT IF start=0
160   PRINT start
170 LOOP
```

RESERVED(space)

This function reserves space in the system heap, and returns its address. For example:

```
10 PRINT RESERVED(10)
```

If you run this several times, the address will keep changing because a new area is reserved each time. It is a good idea to avoid this in a real program by making sure the line is only run once. Heap space is limited and valuable. Its main purpose is to hold short machine code routines that page in and CALL larger routines in another page. It grows at the expense of BASIC's GOSUB/DO/PROC stack and could make a "BASIC stack full" message more likely later on.

If you try to reserve more space than is available, you will get an "Out of memory" error and no space will have been reserved.

You can de-allocate heap space using e.g. LET junk=RESERVED(-10). However, you should only de-allocate space that you allocated yourself in the first place. Otherwise you may well crash the machine by allowing some other programmer's code to be over-written.

If the number of de-allocated bytes is more than the heap contained to start with, the heap size will be reduced to nil (the HEAPEND pointer, SVAR 456, will be the same as the HEAPSTART pointer, SVAR 458).

INKEY$ #0 improvement

The following line used to work fine until you pressed CAPS:

```
10 DO: PRINT INKEY$ #0;: LOOP
```

This caused odd things to appear in the lower screen.... Now it doesn't! INKEY$ #0 has the advantage over a simple INKEY$ that you only get one copy of each keystroke, and unlike GET a$ the keystrokes come from the keyboard buffer so you are unlikely to miss any. To see what I mean, try changing the line to:

```
10 DO: PRINT INKEY$ #0;: FOR n=1 TO 500: NEXT n: LOOP
```

XVAR function - extra system variables

Certain locations within MasterBASIC contain values that control specific aspects of the program. These are analogous with DOS variables (DVARs) and the ROM's system variables (SVARs). The contents can be changed by e.g. POKE XVAR 2,20 and examined by e.g. PRINT PEEK XVAR.

0 PUTSWA. PUT Switch Address. 2 bytes. Unusual in that it holds the ADDRESS of a place you can POKE, rather than the XVAR itself being POKEable. POKE DPEEK XVAR 0,0 will make PUT use the ROM's version of PUT. POKE DPEEK XVAR 0,172 (the code for PUT) to use MasterBASIC's faster version again.

Because the PUTSWA variable is located right at the start of MasterBASIC, and XVAR gives the ADDRESS of a variable, PRINT XVAR 0 gives the start of the program.

2 SOFV. Screen Off Value. Controls time before screen turns itself off. Small values give shorter times. E.g. POKE XVAR 2,12 gives a blank screen after about 1 minute without a keypress, while 75 gives a 5-minute delay, 255 gives an 11 minute delay and 0 gives the normal 22-minute delay.

3 IAPOS. INARRAY Position. After you have used the INARRAY function, if the target string was found in the array, PRINT DPEEK XVAR 3 will return the position within the array string that the target string was found at (1 is the first position).

5 DTTH. DUMP Times To Hit the paper. Normally 1, meaning DUMPs 1, 2 or 3 strike the paper once. Can be POKEd to 2 or more for darker copies. E.g. POKE XVAR 5,2 for double-strike.

6 SORP. Serial Or Parallel. Has a zero value when LPRINT MODE 1 (parallel) has been set, and is non-zero when LPRINT MODE 2 (serial) has been set. If you POKE XVAR 6,1 and then use SAVE BOOT "MD+MBAS1" to save the program, the next time you boot the DOS/MasterBASIC program serial output will be set automatically. (However, the POKE by itself does not do this - it is not examined until BOOT time.)

7 VERSION. Version. number of MasterBASIC, times 10.

8 ILPC. Interrupt LPRINT Characters. Number of characters that MasterBASIC will try to send to a printer per interrupt, when using interrupt-driven printing. Normally 15. Can be increased if you want the printing to take a bigger (but probably still minor) slice of the computer's time (making programs run more slowly) or decreased if you prefer.

9 ILPD. Interrupt LPRINT Delay. 2 bytes. Delay before MasterBASIC gives up, if the printer signals it isn't ready during interrupt-driven printing. Normally 12 (units of about 25 microseconds). If this value is too short, only one character will be sent to the printer per interrupt (50 times per second) because the printer will get one character, signal "not ready" briefly as it accepts it, and MasterBASIC will give up waiting. The printer will print at 50 characters per second - probably rather slower than normal. If you have a serial printer running at 9600 baud then the time to send one character will be about 1000 microseconds and the printer will be "not ready" during this time. The delay may need to be 40 or more - try DPOKE XVAR 9,50. (You may want to reduce the number of characters the computer tries to send with each interrupt to just 4 or 5 - e.g. POKE XVAR 8,4.)

If the delay value is too long, MasterBASIC will waste time waiting for the printer when there is no hope of quick readiness - for example, after a carriage return.

11 SPORT. Serial Port. The port used by the serial driver. Normally 236.

12 BAUD. Baud rate setting for serial input/output. This XVAR and the next two have no effect after being POKEd, until LPRINT MODE 2 is used, or the program BOOTs. The value is related to the actual baud rate in an odd way:

Value	Baud rate
0	50
17	110
34	134.5
51	200
68	300
85	600
102	1200
119	1050
136	2400
153	4800
170	7200
187	9600 (Initial value)
204	38400

13 DBITS. Data Bits in serial output setting.

Value	Bits
147	8 (Initial value)
146	7
145	6
144	5

14 SBITS. Stop Bits in serial output setting.

Value	Bits
23	1
31	2 (Initial value)

15 SDORI. Shaded DUMP Orientation. Normally 0, meaning "Upright DUMPS unless DUMP 3, or DUMP 2 and MODE 3". Can be set to:

1=Sideways. This might be preferred for making banners. Also, most printers introduce some distortion into the output and you may get a better match with the appearance of the screen in a sideways DUMP. E.g. POKE XVAR 15,1.

2=Sideways Mirror Image.

3=Force Upright. Might be used in combination with bit-image print modes that allow more dots across the printer width.

4=Upside Down, Mirror Image.

The next four XVARs control the area dumped when using DUMPS 1, 2 or 3. The coordinate system used has 191 at the extreme top of the screen, 0 at the bottom, 0 at the left and 255 at the right, whatever the screen MODE.

16 SDLHS. Shaded DUMP Left-Hand Side. Normally 0.
17 SDRHS. Shaded DUMP Right-Hand Side. Normally 255.
18 SDTOP. Shaded DUMP Top. Normally 191.
19 SDBOT. Shaded DUMP Bottom. Normally 0.

Control codes sent to printers by DUMP assume that your printer understands Epson's ESC "*" codes, and that it will not produce an automatic line feed. The codes can, however, be modified by altering the XVARs below.

20 GCMX2. Eight bytes available. Message sent to printers before transmission of bit-image data by DUMP 1, 2 or 3. Normally 4,13,27,42,4,0,0,0 for "4 byte message", carriage return, ESC, "*", CHR$ 4, spare, spare, spare.

The carriage return is used to return to the left-hand side in case multiple strike is being used. An earlier Epson standard than ESC "*" CHR$ 4, supported by more printers; is ESC "K". To use this control code, POKE XVAR 20,3,13,27,75. If you own a 24-pin printer, you may want to us(- POKE XVAR 20,4,13,27,42,0 because this will give minimal distortion of height versus width in the output. See also XVAR 35.

28 GCMX4. Three bytes available. Message sent to printers at the end of each line by DUMP 1, 2 o. 3. Normally 2,1;1,10 for "2 byte message", carriage return, line feed.

31 DPVARS. Four bytes..Length, Width, Width multiplier, Height multiplier for DUMP 4. For compatibility with the original DUMP utility, these bytes are copied to the ROM's system variables area at BOOT time. See The User's Guide, page 176.

35 GCMX1. Nine bytes available. Message sent to printer before DUMP. Normally 6,27,108,8,27,51,24,0,0 for "6 byte message", set left margin at 8, set line advance to 24,'216", spare, spare. To alter the left margin which is in characters) POKE XVAR 38,margin. If you are using a 24- pin printer, you will need to alter the line advance - POKE XVAR 41,20.

44 GCMX2. Eight bytes available. Applies to DUMP 4 only. Copied to ROM's system variables at BOOT time. See The User's Guide, page 176. Normally 5,13,10,27,42,4,0,0 for "5 byte message", CR, LF, ESC,"*",CHR$ 4, spare, spare.

52 GCMX3. Six bytes available. Applies to DUMP 4 only. Copied to ROM's system variables at BOOT time. Normally 4,13,10,27,64,0 for "4 byte message", CR, LF, reset printer, spare.

58 DMPTL. Two bytes. Top left address for DUMP 4. Copied to SVAR 45 (not SVAR 16 - The User's Guide is wrong here).

60 MODCHAR1. First character to modify if it is LPRINTed.
Normally 96 (computer's code for Pound).

61 MODCHAR2. Second character to modify if it is LPRINTed.
Normally 35 (computer's code for hash).

63 MODMSG1. Eight bytes available. Sequence to use instead of MODCHARI. Normally 4 (bytes in sequence),27,82,2,35,0,0,0. (ESC,"R",CHR$ 2,CHR$ 35=Epson codes for "select English character set, print CHR$ 35, 3 spare.) To return to the normal effect of printing the Pound sign, POKE XVAR 63,1,96.

71 MODMSG2. Eight bytes available. Sequence to use instead of MODCHAR2. Normally 4 (bytes in sequence),27,82,0,35,0,0,0. (ESC,"R",CHR$ O,CHR$ 35=Epson codes for "select American character set, print CHR$ 35, 3 spare.) To return to the normal effect of printing the hash sign, POKE XVAR 71,1,35.

87 ALTUDG. Two bytes. Displacement of the start of the alternative set of UDGs from XVAR 0. This set is initially an IBM standard international character set covering CHR$ 128168. PRINT XVAR O+DPEEK XVAR 85 gives the address of the first character. Do not POKE this XVAR.

89 ACRSU. (After Carriage Return Set Up) One Byte, normally set to 10. When you switch on your Coup6 SVAR 15 is set to 10 so that every time a Carriage Return (char 13) is sent to the printer a character 10 (Line Feed) is also sent. Many printers allow you to set them to do an 'Auto LF after CR' so you would get a double line feed if the Coup® sent one as well. POKEing SVAR 15,0 will stop this but you will need to do the poke every time you switch on. However, if you POKE XVAR 89,0 and then save a copy of DOS/MasterBASIC, XVAR 89 will be copied automaticaly to SVAR 15 every time you BOOT up the system.

DVAR extensions

The following extra DVARs can be used in addition to those described in the MasterDOS manual:

151 BEEPT. Duration of warning BEEP using with the OVERWRITE and FORMAT y/n queries. Normally 133, but other values can be DPOKEd. DPOKE DVAR 151,0 turns the BEEP off completely.

153 FVFG. Format Verify Flag. Normally 0, meaning "verify the disk after formatting". POKE with a non-zero value to avoid the verify stage. Warning: if you skip the verify stage, any faults on the disk will not show up until later - probably when you want to LOAD a vital file!

154 CMPFG. Compression Flag. Set to 0,1 or 2 for SAVE MODE 1/2/3.

155 DBSTP. Double Step. Normally 0, but can be POKEd to a nonzero value to cause double stepping of the drive head as it moves from track to track. This is useful when reading 40track disks.

APPENDIX A - ASCII AND KEYWORD CODES

These tables show all possible byte values in decimal and hexadecimal notation, and the characters and/or keywords that they code for on the Coup6. The character codes between 32 and 127 follow fairly closely the ASCII standard. Codes below 132 only represent keywords when preceded by 255 (FFH).

Dec	Hex	Char / Keyword	Dec	hex	Char / Keyword
0	00		45	2D	- SHIFT$
1	01		46	2E	. SVAL$
2	02		47	2F	/ USING$
3	03		48	30	0 TIME$
4	04		49	31	1 DATE$
5	05		50	32	2 INP$
6	06	PRINT comma	51	33	3 DIR$
7	07	edit	52	34	4 FSTAT
8	08	cursor left	53	35	5 DSTAT
9	09	cursor right	54	36	6 FPAGES
10	0A	cursor down	55	37	7 SCRAD
11	0B	cursor up	56	38	8 INARRAY
12	0C	delete	57	39	9
13	0D	carriage Return	58	3A	:
14	0E	number prefix	59	3B	; PI
15	0F		60	3C	< RND
16	10	PEN control	61	3D	= POINT
17	11	PAPER control	62	3E	> FREE
18	12	FLASH control	63	3F	? LENGTH
19	13	BRIGHT control	64	40	@ ITEM
20	14	INVERSE control	65	41	A ATTR
21	15	OVER control	66	42	B FN
22	16	AT control	67	43	C BIN
23	17	TAB control	68	44	D XMOUSE
24	18	cursor left word	69	45	E YMOUSE
25	19	cursor right word	70	46	F XPEN
26	1A		71	47	G YPEN
27	1B		72	48	H RAMTOP
28	1C		73	49	I
29	1D		74	4A	J INSTR
30	1E		75	4B	K INKEY$
31	1F		76	4C	L SCREEN$
32	20	space	77	4D	M MEM$
33	21	!	78	4E	N
34	22	"	79	4F	O PATH$
35	23	#	80	50	P STRING$
36	24	$	81	51	Q
37	25	%	82	52	R
38	26	& EXIT PROC	83	53	S SIN
39	27	' EXIT DO	84	54	T COS
40	28	(EXIT FOR	85	55	U TAN
41	29	) LOCN	86	56	V ASN
42	2A	* RESERVED	87	57	W ACS
43	2B	+ EQU	88	58	X ATN
44	2C	, TICS	89	59	Y LN

Continued on next page.

Dec	Hex	Char / Keyword	Dec	hex	Char / Keyword
90	5A	z EXP	141	8D	ì THEN
91	5B	[ABS	142	8E	Ä TO
92	5C	\ SGN	143	8F	Â STEP
93	5D	] SQR	144	90	É DIR
94	5E	↑ INT	145	91	æ FORMAT
95	5F	_ USR	146	92	Æ ERASE
96	60	£ IN	147	93	ô MOVE
97	61	a PEEK	148	94	ö SAVE
98	62	b DPEEK	149	95	ò LOAD
99	63	c DVAR	150	96	û MERGE
100	64	d SVAR	151	97	ù VERIFY
101	65	e BUTTON	152	98	Ÿ OPEN
102	66	f EOF	153	99	Ö CLOSE
103	67	g PTR	154	9A	Ü CIRCLE
104	68	h XVAR	155	9B	Ç PLOT
105	69	i UDG	156	9C	£ LET
106	6A	j NVAL	157	9D	¥ BLITZ
107	6B	k LEN	158	9E	₧ BORDER
108	6C	l CODE	159	9F	ƒ CLS
109	6D	m VAL$	160	AO	á PALETTE
110	6E	n VAL	161	A1	í PEN
111	6F	o TRUNC$	162	A2	ó PAPER
112	70	p CHR$	163	A3	ú FLASH
113	71	q STR$	164	A4	ň BRIGHT
114	72	r BIN$	165	A5	Ñ INVERSE
115	73	s HEX$	166	A6	ª OVER
116	74	t USR$	167	A7	º FATPIX
117	75	u	168	A8	¿ CSIZE
118	76	v NOT	169	A9	BLOCKS
119	77	w	170	AA	MODE
120	78	x	171	AB	GRAB
121	79	y	172	AC	PUT
122	7A	z MOD	173	AD	BEEP
123	7B	{ DIV	174	AE	SOUND
124	7C	¦ BOR	175	AF	NEW
125	7D	}	176	BO	RUN
126	7E	~ BAND	177	B1	STOP
127	7F	© OR	178	B2	CONTINUE
128	80	Ç AND	179	B3	CLEAR
129	81	ü < >	180	B4	GO TO
130	82	é <=	181	B5	GO SUB
131	83	â >=	182	B6	RETURN
132	84	ä -	183	B7	REM
133	85	à USING	184	B8	READ
134	86	å WRITE	185	B9	DATA
135	87	ç AT	186	BA	RESTORE
136	88	ê TAB	187	BB	PRINT
137	89	ë OFF	188	BC	LPRINT
138	8A	è WHILE	189	BD	LIST
139	8B	ï UNTIL	190	BE	LLIST
140	8C	î LINE	191	BF	DUMP

Continued on next page.

Dec	Hex	Char / Keyword	Dec	Hex	Char / Keyword
192	C0	FOR	224	E0	OUT
193	C1	NEXT	225	E1	POKE
194	C2	PAUSE	226	E2	DPOKE
195	C3	DRAW	227	E3	RENAME
196	C4	DEFAULT	228	E4	CALL
197	C5	DIM	229	E5	ROLL
198	C6	INPUT	230	E6	SCROLL
199	C7	RANDOMIZE	231	E7	SCREEN
200	C8	DEF FN	232	E8	DISPLAY
201	C9	DEF KEYCODE	233	E9	BOOT
202	CA	DEF PROC	234	EA	LABEL
203	CB	END PROC	235	EB	FILL
204	CC	RENUM	236	EC	WINDOW
205	CD	DELETE	237	ED	AUTO
206	CE	REF	238	EE	POP
207	CF	COPY	239	EF	RECORD
208	D0		240	F0	DEVICE
209	D1	KEYIN	241	F1	PROTECT
210	D2	LOCAL	242	F2	HIDE
211	D3	LOOP	243	F3	ZAP
212	D4	IF DO	244	F4	POW
213	D5	LOOP	245	F5	BOOM
214	D6	EXIT IF	246	F6	ZOOM
215	D7	IF	247	F7	BACKUP
216	D8	IF	248	F8	TIME
217	D9	ELSE	249	F9	DATE
218	DA	ELSE	250	FA	ALTER
219	DB	END IF	251	FB	SORT
220	DC	KEY	252	FC	JOIN
221	DD	ON ERROR	253	FD	EDIT
222	DE	ON	254	FE	
223	DF	GET	255	FF	
					Keyword prefix

Typesetting and origination by:-

FORMAT PUBLICATIONS.
34, Bourton Road, Gloucester, GL4 OLE, England.
Tel:- 0452 412572

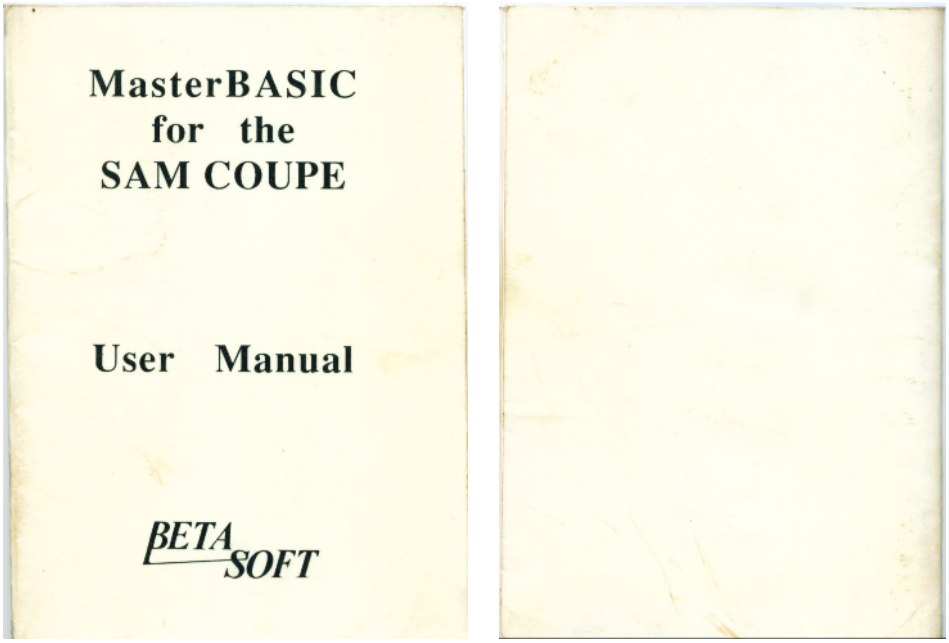
The following pages were not part of the original manual.

ERRATUM

Page 16
"Join To a$ to a$"
should be
"Join To a$ to b$"

page 49
"POKE XVAR 8,4"
should be
"POKE XVAR 8,20"

Page 51
"Normally 4 (bytes in sequence),27,82,2,35,0,0,0"
should be
"Normally (bytes in sequence) 4,27,82,2,35,0,0,0"



Sam's MasterBASIC 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 14 December 2004

For SAM Coupé users everywhere.

www.samcoupe.org

SAM MasterBASIC Manual
PDF version 2 – 14 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]