BETA BASIC

Release 3.0

SPECTRUM 48K

Key	Above	Below
1	DEF PROC	DEF KEY
2	PROC	DEFAULT
3	END PROC	LOCAL
4	RENUM	KEYIN
5	WINDOW	EDIT
6	AUTO	JOIN
7	DELETE	REF
8	KEYWORDS	CSIZE
9		
0	EDIT	
Q	POP	
W		
E	ELSE	
R	ROLL	
T	TRACE	
Y		
U	USING	
I	EXIT IF	
O	ON	
P	DPOKE	
A	ALTER	
S	SCROLL	
D	DO	
F	FILL	
G	GET	
H		
J	WHILE	
K	UNTIL	
L	LOOP	
ENTER		
(blank)		
Z		
X		
C	CLOCK	
V		
B		
N	ON ERROR	
M	SORT	
(blank)		
BREAK		

Copyright BETA SOFT 1985

Scanned, Typed, OCR-ed, and PDF by

Steve Parry-Thomas 25th July 2004.

This PDF was created to preserve this
Manual for the future.

For all ZX Spectrum, Beta Basic
And www.worldofspectrum.org users

(PDF for Michael & Joshua)

Please note if you find a mistake please leave a message
on the www.worldofspectrum.org Forum, with the Error.
So a new version can be made.

CONTENTS

SUBJECT

INTRODUCTION

SUMMARY SECTIONS:
EDITING
PROCEDURES
STRUCTURED PROGRAMMING
STORAGE
DATA HANDLING
GRAPHICS
TOOLKIT FEATURES

ALTER	alter screen attributes
ALTER	alter references in a program
AUTO	auto line numbering
BREAK	more powerful break
CLEAR	move RAMTOP without variable loss
CLOCK	digital clock
CLS	clear a window
CONTROL CODES	cursor and shape control
COPY	with arrays and strings
CSIZE	set character size
DEFAULT	set default variable values
DEFAULT	select SAVE/LOAD device
DEF KEY	define a key
DEF PROC	define a procedure
DELETE	delete program lines
DELETE	delete arrays and strings
DO	start a DO loop
DPOKE	double POKE
DRAW TO	draw TO a point
EDIT	edit a line
EDIT	edit a variable
ELSE	used with IF-THEN
END PROC	end a procedure
EXIT IF	jump out of a DO loop
FILL	fill an area
GET	get a key value
GET	get a screen area
JOIN	join program lines
JOIN	join arrays and strings
KEYIN	enter a string
KEYWORDS	control keyword listing/entry
LET	multiple LET
LIST and LLIST	list a block of lines
LIST DATA	list all variable values
LIST VAL	list numeric variable values
LIST VAL$	list string variable values
LIST DEF KEY	list user key definitions
LIST FORMAT	control indented listings
LIST PROC	list procedures
LIST REF	list references in a program
LOCAL	local variables
LOOP	end a DO loop
MERGE	with auto-running programs
MOVE	MOVE all file types

ON	select statement or line number.
ON ERROR	error handling
OVER	OVER 2 (ORing the screen)
PLOT	PLOT with strings
POKE	POKE with strings
POP	POP Basic's stack
PROC	execute a procedure (optional)
READ LINE	READ strings without quotes
REF	find a reference in a program
REF	parameter type specifier
RENUM	renumber or copy program blocks
ROLL	roll the screen (wrap round)
SAVE	save a program block/variables
SCROLL	scroll the screen (no wrap round)
SORT	sort an array or string
SPLIT	split a program line
TRACE	trace program execution
UNTIL	used with DO loops
USING	format a number
VERIFY	verify a program block/variables
WHILE	used with DO loops
WINDOW	screen windows
XOS, XRG, YOS and YRG	set graphics origin and scale

FUNCTIONS:

General points

AND	bitwise AND
BIN$	convert decimal to binary
CHAR$	convert integer to string
COSE	fast cosine
DEC	convert hex to decimal
DPEEK	double PEEK
EOF	end of file
FILLED	filled area
HEX$	convert decimal to hex
INARRAY	search an array
INSTRING	search a string
ITEM	type/presence of DATA
LENGTH	array dimensions/location
MEM	available memory
MEMORY$	entire memory string
MOD	modulus
NUMBER	convert string to integer
OR	bitwise OR
RNDM	improved RND
SCRN$	improved SCREEN$
SHIFT$	changes case, control codes, etc.
SINE	fast sine
STRING$	multiple strings
TIME$	current time
USING$	format a number
XOR	bitwise OR

APPENDIX A: CHARACTER SET
APPENDIX B: REPORTS
APPENDIX C: ERROR CODES
APPENDIX D: SPECIAL VARIABLES
APPENDIX E: USE OF PRINTERS

Extra Turtle Graphic commands
BETA BASIC THE DISCIPLE AND PLUS D INTERFACES.

INTRODUCTION

Congratulations on buying Beta Basic 3.0. The many new commands and functions it provides give the Spectrum a Basic of great power and flexibility, which has few rivals on any microcomputer. This manual may strike you as a bit formidable, but there is no need to try to read it straight through - read the Introduction and the section on Editing, and then dip into the rest as you like. Because Beta Basic is compatible with Spectrum Basic, you will not have to "unlearn" anything, although we recommend that you look at the advantages of structured programming and the use of procedures.

LOADING BETA BASIC

To load the program from tape, enter:

LOAD "" or LOAD "Beta Basic"

Three lines of Basic (lines 0, 1 and 2) will be loaded and line 2 will run automatically, thus loading the main part of Beta Basic, which is a block of machine code 18K in length. RAMTOP is lowered to 47070 to protect the machine code at the top of memory. Loading has finished when you see the copyright message at the bottom of your screen. If this does not happen, there has been a tape loading error - try again. (Use a different volume setting or try the second copy of the program which is on the other side of the tape.) Lines 1 and 2 will be automatically deleted once the copyright message has been displayed, and just line zero will remain. (Normally only the figure zero is visible.)

BACKUP AND MICRODRIVE COPIES

Line 1 is a SAVE routine which will save the current Basic program, followed by the machine code part of Beta Basic. If you wish to use it, LOAD the Beta Basic program as usual, then MERGE the first (Basic) part to get a copy of lines 1 and 2. Now RUN will cause the program to be saved to tape. To save to the Microdrive instead, type: (space) default-m1. This will make the normal tape commands work with the Microdrive, and RUN will therefore save the program to drive l. (See the DEFAULT SAVE/LOAD device command for more details.) However, before doing this you still need to edit line 2 to the conventional Microdrive syntax. (The CODE has to LOAD before DEFAULT will work!) If Beta Basic is initialised with Interface 1 attached, it makes some modifications to its own code to suit different issues of the interface ROM. This modification is not performed by copies of Beta Basic, only the original, so if you change to a different version of Interface I you will have to make another copy from your original tape. (Version 2 of Interface 1 was introduced at a serial number of 87316.)

Note: Beta Basic's CODE will not VERIFY if it is saved while it is actually working, because internal system variables are changing. If this concerns you, initialise Beta Basic, edit lines 1 and 2 to the conventional Microdrive syntax, then turn Beta Basic off again using RANDOMIZE USR 59904 before using RUN to SAVE the program.

PLEASE NOTE: The facility to save Beta Basic has been added to allow you to make backup copies for your own use, or transfer to Microdrive. Copying the manual, or providing other people with normal copies of the program, is a breach of copyright which deprives the programmer of a fair return for his (considerable!) work in writing the program. We try to supply good customer support; please support us by observing our copyright. If you wish to give your friends copies of Beta Basic programs that you have written, please modify the Beta Basic CODE as follows (using a single program line) before copying it:

POKE 64218,0: POKE 64219,0: POKE 64220,0: POKE 64861,201

This ensures that programs can be RUN but not altered.

DIFFERENCES FROM NORMAL OPERATION

Line 0 contains function definitions for Beta Basic's new functions, and it is always present, although it is not normally listed unless it is the only line in the program. The other immediate sign that Beta Basic is resident is the inverse "current line pointer". The normally faint keyboard click has been lengthened - if you dislike this, POKE 23609, 0 to turn it off. Any existing program that you now MERGE with Beta Basic will run normally, except that you will have to enter "KEYWORDS 0" (see the KEYWORDS command) if you want to display user-defined graphics instead of Beta Basic's keywords. (Don't LOAD a program that was not written under Beta Basic or you will lose line zero - use MERGE instead.)

Programs are LISTed with program lines which are longer than one screen line indented so that only line numbers occupy the first four columns on the left side of the screen.

NEW is less drastic than it usually is - it simply deletes any program that is present, except for line 0, and performs CLEAR. (If you want to get rid of Beta Basic, the effect of briefly unplugging the computer can be duplicated by RANDOMIZE USR 0.)

SPEED.

You will probably see some increase in program speed compared to Spectrum Basic, particularly in long programs.

FOR-NEXT loops run at a constant speed anywhere in the program, unlike the case with Spectrum Basic. The best speeds are obtained when the start, limit and step are all whole (possibly negative) numbers less than 65536. Such a FOR-NEXT loop will run about 5 times faster than normal at the 100th. program line, or 17 times faster at the 500th. line.

GOTOs and GOSUB are faster than before, when the destination line is part-way down a program. RETURN is also much faster - a RETURN to the last line of a program is now just as fast as a RETURN to the first line (unlike the case with Spectrum Basic).

Some of these speed increases are achieved by use of stored addresses, rather than line numbers. This means that if you stop a program part way through a FOR-NEXT loop, for example, and add or edit a program line higher in the listing, CONTINUE may not work because the stored address to loop to is no longer accurate. (However, the opening and closing of Microdrive buffers does not cause a problem.)

USE WITH EARLIER BETA BASIC RELEASES.

If you load a program written under release 1.0 or 1.8, the old version of line 0 (which contains function definitions) will be present. If you wish to use the new functions which have been introduced with release 3.0, you must replace the old version of line 0 with the new version as follows:

1. LOAD "Beta Basic" release 3.0.
2. LOAD your release 1.0/1.8 Basic program.
3. MERGE "Beta Basic" release 3.0 to get a copy of the new line 0.
4. Remove lines 1 and 2 if desired.
5. SAVE your Basic program again.

You must do this in the case of release 1.0 programs.

There are two points of incompatibility between release 3.0 and earlier versions: the variable called "line" that was created by ON ERROR and TRACE is now called "lino" (to prevent it being turned into the keyword LINE as you enter it), and procedure names can no longer contain internal spaces. In either case the use of the ALTER (program references) command will allow you to modify your old programs easily.

Some of the commands that you are familiar with have been improved. You should re-read ON, RENUM, ROLL and SCROLL. CLOCK, ON ERROR and TRACE can have a different form. GET (number) is slightly different - it no longer matters if letters are in capitals or not. Procedures have been greatly extended.

SAVING AND LOADING PROGRAMS WRITTEN WITH BETA BASIC

Once you have written a program containing some of the new keywords, you can save it in the normal way. When you wish to load it back in the future, first of all load Beta Basic if it is not already present in the computer, then load your program (a copy of line 0 will have been saved with the program, so there is no need to use MERGE). Alternatively, you can use a version of lines I and 2 of the loader program to SAVE both your Basic program and the CODE part of Beta Basic. Both parts will then re-load automatically.

If you LOAD a program written with Beta Basic when the Beta Basic CODE is not present, the new commands will appear as single characters, which will give "Nonsense in BASIC" reports when you RUN the program. If you are using tape, you can use:

```
MERGE "Beta Basic": GO TO 2
```

to load Beta Basic in this case. The "GO TO 2" is needed to make line 2 "auto-RUN" (the normal auto-RUN is stopped by MERGE).

ABOUT THIS MANUAL

The rest of this manual is organised as follows:

l. Sections on what Beta Basic has to offer in some major programming areas. These vary from detailed descriptions to simple lists of the main associated keywords, depending on the topic. You should read the section on EDITING first.

2. Detailed descriptions of each command in alphabetical order.

3. Detailed descriptions of each function in alphabetical order.

4. Appendices

We have done our best to ensure that this program and manual are free of errors. If you encounter any problems, please write to the address below, describing the circumstances as fully as possible. Experience has shown that users sometimes suffer relatively minor problems in silence, assuming that we either know about them or don't care. We would rather they complained! We can assume no responsibility for any losses incurred due to programming or documentation errors; however, we will attempt to correct most problems. We would also appreciate suggestions for future enhancements, or other comments.

We wish to thank the many customers who have sent complimentary letters or interesting suggestions for improvements since the last release. Unfortunately, it hasn't been possible to incorporate all deserving ideas.

If you bought this program from a shop, please retain your original cassette as evidence of purchase in order to qualify for a reduced price on possible future releases.

EDITING

Associated keywords:

EDIT, KEYWORDS, LIST FORMAT, CSIZE, JOIN, SPLIT

Beta Basic makes entering and editing programs considerably easier than it is with Spectrum Basic. If you type in a program, or MERGE a Spectrum Basic program that you already have, you will be able to try out the new features. No features of the normal Spectrum Basic editor have been lost, so you should have few difficulties. The paragraphs below concerning keywords (which are capitalised) give a brief description of what the keyword does - for more detail, see the main part of the manual.

Current Line Cursor.

Perhaps the first thing you will notice is that the program cursor showing the current line is INVERSE to make it easier to see. It can also be moved up and down under the control of the cursor keys much faster than before, because the whole screen is not re-listed at every move. (On rare occasions the cursor position may not correspond exactly to the program on the screen. If this happens, just press ENTER to re-list the program.)

EDIT program lines

Lines which are inconveniently far from the current line cursor can be easily edited by pressing the zero key, then entering the number of the line, and pressing ENTER. The line will appear at the bottom of the screen.

Cursor movement within the line.

The editing cursor can be moved up and down within the line using the cursor keys, as well as left and right. The cursor will not go through keywords, but takes the first following gap. If you attempt to move the cursor higher than the top of the line or lower than the bottom, the cursor will jump to the end of the line. You can use EDIT (line number), then cursor-up, as a fast, easy way to add more statements to the end of a line.

Cursor mode.

If you are in K mode and you enter a space, the mode will change to L/C mode. This is useful if you want to enter a procedure name or type in a keyword in full (see below). If you are in L/C mode you can change to K mode by holding down symbol shift and pressing ENTER. You can use this to enter keywords even if you have "turned off" K mode using KEYWORDS 4, and it is useful when putting keywords into strings (you might want do this with REF, ALTER or KEYIN).

KEYWORDS - controlling program entry.

The KEYWORDS command allows any keyword to be entered either by the normal "single key" method, or by spelling it in full. As supplied, KEYWORDS 3 is set, and both forms of entry will be accepted in the same line. Even if you know the old system very well, you will find it is easier to type keywords like IN as separate letters. The letters may be in upper or lower case.

Entering Beta Basic keywords.

There are two ways of entering Beta Basic's keywords. You can type them in full (using a leading space to turn off K mode if necessary) or alternatively use a "single key" method. If you use the latter method, commands and functions are treated differently. To enter a new command, first go into Graphics mode, then press a key. Most keys (including the symbol shifted numbers) will give a new keyword. The relevant key is given near the top of the manual entry for each command. To enter a new function, type FN, either conventionally, or as "f" + "n" + " " , or as Graphic Y. Then type the relevant letter and "$" or "(", as given at the top of the explanation of each function, and the new keyword will appear. See also the GENERAL introduction to the functions section.

Error checking.

Beta Basic checks syntax on entry, as Spectrum Basic does, and it also gives a BEEP if it detects an error. (The beep length can be changed by POKEing location 23608.) The sound is useful if you copy programs out of magazines without looking at the screen very often. If you are spelling out keywords in full, typing errors may often be accepted as procedure names. For example, if you type PAPRE 1, you will get a "Missing DEF PROC" message (unless the program contains a procedure called PAPRE 1) rather than a syntax error. Appendix B gives a full list of the new error messages that may be produced.

LIST FORMAT

This command allows you to produce an automatic "pretty listing" with indentation to aid program readability.

CSIZE

You can reduce the character size to give listings with up to 64 characters per line, or increase the size for titles or demonstration purposes.

JOIN (program lines) and SPLIT

You can JOIN adjacent program lines together into one line, or SPLIT a line into two.

"New line" characters within a line.

You can enter a "new line" character anywhere in a program line or an INPUT line by pressing caps shift and then ENTER. Unlike the normal ENTER, you can continue to type on the next line. The line will be accepted when you press ENTER without caps shift. See CONTROL CODES (CHR$ 15).

PROCEDURES

Associated keywords: DEF PROC, END PROC, LOCAL, DEFAULT, REF, READ LINE, LIST PROC, ITEM( ) function.

This section gives a broad overview of the use of procedures in Beta Basic. You will probably need to consult the entries for individual keywords to obtain greater detail.

The use of procedures is an idea which Basic programmers are becoming increasingly aware of. Newer versions of Basic often allow at least some use of procedures, and educators push the idea enthusiastically. However, the main reason you should be interested in procedures is not because they are fashionable or beloved by academics, but because they make programming easier! Beta Basic has one of the most complete versions of procedures available on any micro, giving great flexibility and power.

The great advantage of the procedure concept is that you can write a program "module" that does a specific job, without any unwanted effects on the main program, such as alterations in the values of important variables. Once the "module" is working correctly, you should be able to forget the details of how it works and use it in any program, simply by MERGEing it and using its name. Essentially, you can add new commands to Basic, in a way that anyone can master. Each procedure should be short enough to understand fairly easily; complicated jobs should be split up into logical sections, with a procedure for each section.

A procedure is a named part of a program, starting with a DEF PROC statement that gives the name of the procedure and the names of variables that will be used to handle data being passed into or out of this part of the program, and ending with an END PROC statement. We will refer to this section of program as a "procedure definition".

Let's start with a simple (and rather useless) example, a procedure called "greet":

```
100  DEF PROC greet
110     PRINT "HELLO!"
120  END PROC
```

RUN this - and nothing happens! This is because the procedure definition is ignored unless it is called by name (rather like DEF FN is ignored until you use FN). Add:

```
10 greet
```

using a space before the "g" to get out of "K" mode (this isn't necessary if you are using KEYWORDS 4). Alternatively, add:

```
10 PROC greet
```

which will do the same thing - that is, print "HELLO". The use of the PROC keyword has been retained to maintain compatibility for owners of earlier versions of Beta Basic, but it is no longer necessary.

We will refer to the use of a procedure's name (resulting in the definition part being executed) as a "procedure call". The name of a procedure must start with a letter. Space, ":", ENTER, REF or DATA terminate the name. Most other characters can be used as part of the name, but it is best to stick to letters, numbers and "_". It doesn't matter whether the letters are in upper or lower case. The DEF PROC can be anywhere in the program, providing that the DEF PROC is the first statement in a line (this restriction speeds things up a bit). The procedure definition can be any number of lines long; it is terminated at any point by END PROC. (The computer will not be able to successfully "jump over" a procedure definition in the listing if you have used multiple END PROCs with a single DEF PROC - if you want to do this, you will have to jump over the definitions yourself, or put them at the end of the program after a STOP.)

Now change the program to:

```
100   DEF PROC greet times
110      FOR n=1 TO times
120         PRINT "HELLO"
130      NEXT n
140   END PROC
```

The "times" in the DEF PROC statement is a variable name which can accept data from the corresponding position in the procedure call. We will refer to such variable names in a DEF PROC statement as "formal parameters". The values in the procedure call, which are "passed" to the formal parameters, are called "actual parameters". (We try to avoid using unfamiliar terms, but it is hard to talk about procedures without introducing a few!) The formal parameters must be variable names, but the actual parameters can be expressions, constants or variables (unless REF is being used - more about this later).

You will find that the name "greet" used by itself will give you a "variable not found" report at line 110, because "times" has not been given a value. (You might have expected an error at line 100 - but Beta Basic does not require all the actual parameters to be supplied at this point.) However:

```
greet 5
```

will print "HELLO" five times because the value 5 (the actual parameter) has been "passed" to (or assigned to) the formal parameter "times". You have a new command that will print "HELLO" any number of times, by using an expression after "greet". Ideally, the procedure should have no other effects on the program in which it is used. Is this true? Haven't we created the variables "times" and "n"? Try:

```
PRINT n,times
```

You will find that "n" exists but "times" does not. The reason is that "times" was used after DEF PROC, which automatically made it a LOCAL variable (LOCAL gives more details). This means that whenever the procedure is called, any pre-existing variable (often called a "global" as opposed to a "local" variable) called "times" will be stored, before the name is re-used by the procedure. At the end of the procedure, the original value of "times", if there was one, will be put back. If there was no pre-existing "times", as in this case, the variable will be erased. The variable "n" is different because it was not included in the DEF PROC statement. A procedure that changes "n" is a potential problem, because you may already be using that letter. To make "n" a local variable like "times", add to the program:

```
105 LOCAL n
```

and add these lines so that you can see that the procedure really doesn't change the global values of the variables:

```
10 LET n=1234, times=5678
20 greet 10
30 PRINT n, times
```

The procedure "greet" was educational, rather than useful. Below is a rather more interesting example:

```
100 DEF PROC box x,y,width,height
130 PLOT x,y: DRAW width,0
140 DRAW 0,-height: DRAW -width,0
150 DRAW 0,height
160 END PROC
```

The procedure "box" has four parameters - the x and y coordinates of the top left hand corner of the desired box, and its width and height. So:

```
box 100,100,10,40
```

will draw a tall rectangle near the middle of the screen. It would be convenient if we could have a procedure that would assume we wanted height to equal width (i.e. a square) if we didn't bother to specify the height. At the moment, we would get "Variable not found" at line 140 if we tried:

```
box 100,100,50
```

To prevent this, add:

```
120 DEFAULT height=width
```

which means, "if height does not exist, LET height=width". (If you have not included a height parameter in the procedure call, height will definitely not exist at this point - because even if it was in use by the main program, its appearance after DEF PROC will have made it LOCAL i.e. the original value will have been hidden from the procedure.)

Now, if you are really lazy, and want the procedure to assume a 20-pixel square if both width and height are omitted, add:

```
110 DEFAULT width=20
```

which allows " box" to be used with just x and y values. So far, we have omitted only the last parameters from our procedure calls, but it is possible to omit any parameter; e.g.:

```
box ,y,25,10
```

will assign y, width and height successfully - you would have to add a DEFAULT x to the procedure, though.

PASSING PARAMETERS BY REFERENCE

So far, we have only dealt with information being passed into procedures - i.e. the actual parameters in the procedure call are passed to the formal parameters given in the DEF PROC statement. However, we might want to pass information the other way - in other words, get some sort of result from the procedure. We could write a procedure which calculated such a result and put it in, say, x, but this violates the concept of procedures as independent plug-in units, because we would have to remember that the procedure used x and avoid its use for anything else in the main program. It would be nice if we could 'tell' the procedure the name of the variable to put the result in when we called it. Well, we can! (At least with Beta Basic - lesser versions such as BBC Basic do not allow this.)

Normally, only the values of any variables in the procedure call are passed - if we wish to pass the name as well, this must be specified in the DEF PROC statement by preceding the parameter name with the "REF" keyword. This stands for "reference", and the parameter is said to be passed "by reference". Essentially, the variable named in the procedure call is renamed with the name following REF in the DEF PROC statement. At the end of the procedure, the process is reversed, so that any changes which have occurred in the value of the variable will be available globally. Here is an example which swops two strings:

```
200  DEF PROC swop REF a$,REF b$
210     LOCAL t$
220     LET t$=a$,a$=b$,b$=t$
230    END PROC
```

To show this working, add:

```
10    LET x$="hi",y$="goodbye"
20    swop x$,y$
30    PRINT x$,y$
```

Without the use of REF, the values provided by x$ and y$ would still have been swopped within the procedure, but there would have been no global effects, because a$ and b$ would have been erased at the end of the procedure. With REF, a$ and b$ are just temporary names used to refer to the actual parameters x$ and y$, and thus changes to a$ and b$, such as swopping, are reflected in the actual parameters after the procedure has been called. Any global versions of a$ and b$ will be unchanged by the procedure. Note that there must be a variable in the procedure call corresponding to any REF parameter in the DEF PROC statement - you can't just use a value, like "hee hee" or 123.

Most versions of the procedure concept do not allow arrays to be used as parameters. Beta Basic does, but it insists that arrays are passed by reference - they are re-named, rather than having their contents copied to another, LOCAL array. This has the advantage that the computer does not need two separate copies of each array, which saves memory. (If you really want a LOCAL copy of the array, you can create a LOCAL array and COPY the data to it with Beta Basic's array handling COPY command.) Here is an example which adds all the elements of any one-dimensional numeric array:

```
300   DEF PROC total REF a(), REF sum 310  LOCAL n
320      LET sum=0
330      FOR n=1 TO LENGTH(1,"a()")
340         LET sum=sum + a(n)
350      NEXT n
360   END PROC
```

The array letter is followed by brackets in order to distinguish it from a simple numeric variable with the same letter. The parameter "sum" is preceded by REF so that its final value can be assigned to a specified global variable. The function LENGTH (see LENGTH) tells us how many elements are in the array so that the procedure will work with arrays of any length. We need an array to show the procedure working:

```
100   DIM t(10)
110   FOR n=1 TO 10
120      LET t(n)=n
130   NEXT n
```

and to total the array, add these lines:

```
140   total t(),answer
150   PRINT answer
      REM 55
```

PARAMETER LISTS

Sometimes rather than defining a particular combination of parameters for a procedure, you might wish to be able to deal with a list of parameters of unknown length. A special feature has been included to make this possible. If the keyword DATA is used in the DEF PROC statement instead of the usual list of formal parameters, then subsequent READs will take information from a list of actual parameters following the procedure call. For this feature to be useful, it must also be possible to tell when all the items in the list have been dealt with. This can be done with the function ITEM(), which returns 0 if there are no more items, 1 if the next item is numeric, or 2 if the next item is a string. (See ITEM for more information.) Here is an example that ERASEs a list of Microdrive files:

```
100  DEF PROC kill DATA
110     DO UNTIL ITEM()=0
120        READ a$
130        ERASE 1,a$
140     LOOP
150  END PROC
```

Note: Line 110 could also be written DO WHILE ITEM() with the same effect.

Now you could type, for example:

```
kill "test", "test2", "file5"
```

In the example, "READ a$" will generate an error unless the list consists only of strings, but you can write procedures to deal with mixed lists by checking the type of the next actual parameter (using ITEM) before you READ it. It is also possible to avoid the use of quotes, in many cases, by using READ LINE: instead of READ (see READ LINE). Alter line 120 and add line 125 as follows:

```
120 READ LINE a$
125 IF a$(1)=CHR$ 34 OR a$(LEN a$)="$" THEN LET a$=VAL$ a$
```

This allows for the use of quotes or string variables, but also allows:

kill test,test2,file5

Note that dealing with a list of actual parameters in a DATA statement means that no variables will be made LOCAL unless you specify them yourself.

RECURSION

A procedure that calls itself is said to be recursive. (An old joke says that the definition of recursion in computer dictionaries is: Recursion - see Recursion.) Recursion can often be an elegant and concise way to solve a programming problem. On the other hand it will probably not produce the fastest solution, and a considerable amount of memory may be needed temporarily (each call may generate another set of local variables). Also, although it is fairly easy to see what is happening in a general way, thinking about the details may make your head hurt! The example below is a procedure that draws a diamond - and then calls itself to draw a smaller diamond around each point of the first diamond, and even smaller diamonds around the points of those diamonds, etc. There is a lower limit on size to prevent the process continuing forever.

```
100 DEF PROC diamond x,y,size,diff
     DEFAULT diff=15
     PLOT x,y-size
     DRAW -size,size
     DRAW size,size
     DRAW size,-size
     DRAW -size,-size
110  IF size>4 THEN
        diamond x,y+size,size-diff
        diamond x,y-size,size-diff
        diamond x-size,y,size-diff
        diamond x+size,y,size-diff
120 END PROC
```

You can call the procedure with something like:

```
diamond 128,88,40
```

ERRORS

If you have not defined a procedure, calling it will cause report W, "Missing DEF PROC"; this will also happen if you use END PROC without DEF PROC. If you forget to include an END PROC, the main program will give report X, "No END PROC", when it tries to jump over the procedure definition and can't find where it ends.

If there are too many parameters in a procedure call, you will get a "Parameter error" report. If the type of the actual parameters does not match that of the formal parameters, you will get a "Nonsense in BASIC" report. END PROC may give a "Variable not found" report if a variable which is supposed to be passed out of the procedure does not exist.

STRUCTURED PROGRAMMING

See the section on PROCEDURES, and the individual keyword entries for details.

Associated keywords:

With procedures:

DEF PROC, END PROC, LOCAL, REF parameter, DEFAULT variable value, ITEM function.

DO, LOOP, EXIT IF, WHILE and UNTIL provide the equivalent of REPEAT and WHILE loops and are much more flexible.

ELSE is used with IF...THEN.

ON can select a line number or a statement from a list. It provides a limited form of CASE (or SELECT) command.

LIST FORMAT provides automatic indentation of programs to bring out their structure.

STORAGE

Associated keywords:

DEFAULT (SAVE/LOAD device), SAVE, MERGE, MOVE, LENGTH function.

DEFAULT SAVE/LOAD device.

Beta Basic allows use of a simplified form of SAVE, LOAD, MERGE and VERIFY, omitting the "*" and "m". For example:

```
LOAD 1;"program"
```

The DEFAULT (device) command allows you to select an assumed SAVE/LOAD device, simplifying things even more. For example, after entering DEFAULT =m1 the command SAVE "program" would save to Microdrive l. ERASE and CAT have also been simplified - ERASE "program" and CAT are both valid commands.

SAVE and VERIFY enhancements.

You can SAVE or VERIFY part of a program using a "slicer" to specify the block of lines concerned. You can also SAVE or VERIFY the entire variables area by putting the keyword DATA in the command. These features work with tape, the Microdrives, the Net or the RS232 link.

MERGE will merge an auto-running program from Microdrive.

MOVE will move programs, CODE and arrays, as well as data files.

The LENGTH function can tell you the location of an array, which may be of use when SAVEing or LOADing CODE.

DATA HANDLING

Associated keywords:

JOIN, COPY, DELETE and SORT arrays and strings; INARRAY and INSTRING search functions; LENGTH, CHAR$ and NUMBER functions.

EDIT variables, SAVE DATA, USING, functions EOF, SHIFT$, USING$.

The JOIN and COPY commands allow you to move or copy all or part of an array to another array. It is possible to change the size of an array without data loss. DELETE removes part of an array, and SORT performs very fast alphabetic or numeric sorting. The commands also work with strings. The INARRAY and INSTRING functions search arrays and strings respectively. The function LENGTH gives an array's dimensions and location; it may help you in splitting up and loading arrays which are too large to load in one piece with Beta Basic resident. CHAR$ and NUMBER functions allow you to create "integer" arrays.

You can EDIT the values of variables in a similar way to program lines. All of a program's variables can be saved with SAVE DATA. USING and the function USING$ allow you to format numbers neatly. The End-Of-File (EOF) function detects when the last item has been read from a Microdrive file. SHIFT$ (among other things) can change the case of strings.

GRAPHICS

Associated keywords:

ALTER, CONTROL CODES, CSIZE, DRAW TO, FILL, GET screen area, OVER 2, PLOT, POKE, ROLL, SCROLL, WINDOW, XOS/XRG/YOS/YRG, functions SINE, COSE, FILLED, MEMORY$, SCRN$.

Some highlights:

ALTER changes screen attributes in a flexible way.
DRAW TO draws to a specified point.
GET allows part of the screen to be stored in a string;
PLOT will put it back elsewhere, at a different size if
CSIZE is used to shrink or magnify it.
POKE can deal with larger areas more rapidly.
FILL can fill any enclosed area with a given INK or PAPER.
ROLL can move all or part of the screen in any direction;
SCROLL is similar but without wrap-round.
SCRN$ will recognise user-defined graphics.
WINDOW can keep your text out of the way of your graphics.
XOS, XRG, YOS and YRG change the graphics scale and origin.

TOOLKIT FEATURES

ALTER (references) - search and change command.
AUTO - automatic line numbering.
DEF KEY - 36 user-defined keys can hold commands or strings.
DELETE - a block of program lines.
LIST or LLIST - a 'slicer' of program lines.
LIST or LLIST DATA/VAL/VAL$ - the variables.
LIST or LLIST DEF KEY - the user-defined keys.
LIST or LLIST PROC - a procedure.
LIST or LLIST REF - the lines in which a 'reference' occurs.
REF - program search.
RENUM - renumber with block move and block copy facilities.

MEM() function - amount of free memory.

ALTER <attributes> TO attributes

Key: A

ALTER allows extensive manipulation of the screen attributes file (which holds INK, PAPER, FLASH and BRIGHT information for each character position). In its simplest form, ALTER can change the overall INK or PAPER colour without clearing the screen:

100 PRINT AT 10,10;"TEST": PAUSE 50: ALTER TO PAPER 1

will change the PAPER colour of every character position to blue. You can also change the whole screen to some combination of attributes:

ALTER TO PAPER 2, INK 7, FLASH 1

will work. It is possible to be selective about which character positions are affected by including a description of their attributes before the TO:

ALTER INK 7 TO INK 0

will change anything in white ink to black ink. It is possible to create a complicated graphic using ink that is the same colour as the paper, and then get it to appear instantly using ALTER to change the ink (or paper) colour. Complex statements are possible, such as:

ALTER INK 3, BRIGHT 1, PAPER 7 TO INK 5, FLASH 1

which will only affect character positions with the attributes INK 3, BRIGHT 1, PAPER 7.

The following program shows some of the possibilities - it creates a large flashing checker-board. Try changing the ALTER statements to vary the effect. (You can change the rate of flashing by altering lines 170 and 220.)

```
100  LET a=2,b=4
110  FOR 1=1 TO 5
120    FOR n=1 TO 16
130      PRINT INK a; PAPER b;"XXXX"; PAPER a; INK
         b;"OOOO";
140    NEXT N
150    LET c=a,  a=b, b=c
160  NEXT 1
170   LET t=30
180  ALTER INK a TO INK b
     PAUSE t
190  ALTER PAPER a TO INK a
     PAUSE t
200  ALTER INK a TO PAPER b
     PAUSE t
210  ALTER INK b TO PAPER a
     PAUSE t
220  LET t = t - t/10 + 1
230  GO TO 180
```

ALTER reference TO reference

Key: A

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.) For example:

```
ALTER a$ TO b$
```

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

```
ALTER count TO c
```

will alter the numeric variable name or procedure name "count" to "c" - but "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 its contents. For example:

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

If you wish to use ALTER with strings containing keywords, just type them in by using symbol-shift/enter to force "K" mode.

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

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

One consequence of the invisible five-byte form is that numbers often take more room than variables or expressions - so you can save memory by altering e.g. the number 1 to SGN PI (saves 5 bytes) or any number to VAL "number" (saves 3 bytes). To do this for all numbers from 1 to 100, we could use:

```
1 LET free=MEM()
2 FOR n=1 TO 100
3 ALTER (n) TO "VAL"+CHR$ 34+STR$ n+CHR$ 34
4 NEXT n
5 PRINT "Saved ";MEM()+28-free;" bytes"
```

The "n" is in brackets so that ALTER knows that we wish to look for the current value of the variable, not the variable name itself. The line numbers are low because a FOR - NEXT loop should not move around in memory while running (and it will if earlier parts of the program are modified by ALTER). The "28" in line 5 compensates for the creation of "free" and "n" by the routine.

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 LIST REF command.

AUTO <line number> <,step value>

Key: 6

AUTO turns on an automatic line numbering facility for convenient entry of programs - the computer enters the line numbers for you. If the step value is omitted, ten will be used. If AUTO alone is entered, the line number assumed will be the current line (with the program cursor), plus ten. AUTO is turned off when the line number is less than 10 or more than 9983, or when any report is printed. A convenient method for leaving AUTO is to press BREAK for more than about a second.

If you wish to skip a block of line numbers while using AUTO, just delete the line number offered and enter your own. Type the rest of the line and enter it. The next line number offered will be the line number you provided, plus the step value.

`AUTO`	- from the current line + 10, step 10
`AUTO 100`	- from line 100, step 10
`AUTO 100, 5`	- from line 100, step 5

BREAK

Key: Caps shift-space (not graphics mode)

(Break is not a keyword.) The normal BREAK provided by Sinclair Basic is perfectly adequate for most purposes; however, many people interested in machine code will have experienced a "lock-up" when their code becomes stuck in an endless loop which cannot be terminated by BREAK. Beta Basic adds a back-up BREAK system - if you press shift space for more than about a second, the back-up system decides that you must be stuck and performs BREAK even if the normal routine doesn't work. This can be helpful if you have incautiously disabled "STOP in INPUT" or "BREAK into program" using the ON ERROR command. It will also get you out of INPUT LINE, EDIT and AUTO.

Notes: i) If you have gone back to the Sinclair copyright message, BREAK cannot help.
ii) (For machine code programmers) - you must not disable interrupts if you wish to use this form of BREAK, (Interrupt mode 2 is always set.)

CLEAR bytes

This command will be of most interest to machine code users. CLEAR with a value of 767 or less will move RAMTOP down by the specified amount. (Why 767? It is only 3 digits long, so you are unlikely to mix it up with the normal CLEAR, which requires a 5 digit number - and it is easy to check for in assembly language.) The screen, variables and GOSUB/DO-LOOP/PROC stack are not affected. User-defined keys and window definitions (which are stored immediately above RAMTOP) are moved at the same time; the space is created after the end of these definitions and before the start of Beta Basic's code at address 47070. (Note that the space is not cleared with zeros.)

CLEAR with a negative number will move RAMTOP and the user-defined keys and windows up by the specified amount. (It makes a negative amount of space.) No check prevents you moving RAMTOP up past the bottom of Beta Basic's code, so be careful. You will not want to use this command without having previously moved RAMTOP down at some stage. The example shows RAMTOP being moved without loss of a variable. Remember that the space is created below Beta Basic's code, which is often not just above RAMTOP.

```
100 LET ramtop=23730
110 PRINT DPEEK(ramtop)
120 CLEAR 100
130 PRINT DPEEK(ramtop)
140 CLEAR -50
150 PRINT DPEEK(ramtop)
```

CLOCK number or string

Key: C

CLOCK controls the operation of an interrupt-driven 24 hour clock, which can display the current time in the upper right hand corner of the screen in hours, minutes and seconds. When a preset time is reached, an audible alarm can be sounded and/or a given subroutine can be GOSUBed. "Interrupt driven" means that the clock keeps running even while you write or RUN a program. Only during tape, Microdrive or Interface I operations, or during BEEP will the clock stop. To control which of the facilities mentioned are in operation, CLOCK should be followed by a numeric expression in the range of 0 to 7, which will set the clock to the specified mode. The results will be as follows:

MODE	ALARM GOSUB	AUDIBLE ALARM	DISPLAY
0	NO	OFF	OFF
1	NO	OFF	ON
2	NO	SET	OFF
3	NO	SET	ON
4	YES	OFF	OFF
5	YES	OFF	ON
6	YES	SET	OFF
7	YES	SET	ON

The CLOCK will start running as soon as Beta Basic is loaded, starting at "00:00:00" if the load was from the original tape. Enter CLOCK 1 to see the display. No doubt the time is wrong! To set a time, CLOCK should be followed by a string, for example:

```
CLOCK "09:29:55"
```

Actually, you don't really need the ":" separators - CLOCK will (with one exception) simply take the first six digits in the string to be the current time, and ignore anything else. If the string contains less than six digits, the rest are assumed to be zeros:

```
CLOCK "xyz10"
```

gives a display of 10:00:00. The exception mentioned above is "a" (or "A") - this letter causes the following digits to be used as the alarm time:

```
CLOCK "A06:20"
```

will set the alarm to twenty past six. When the current time reaches the alarm time, the audible alarm will sound briefly if it has been set by entering CLOCK 2, 3, 6 or 7 (see the table above).

It is also possible for a specified subroutine to be GOSUBed when the alarm time is reached, if CLOCK 4, 5, 6 or 7 has been executed. This will only happen if some kind of program is running at the time (your program writing will not be interrupted). The program can be as simple as:

```
10 GO TO 10
```

or as complicated as a word processor or a game. Once the alarm time is reached, CLOCK waits for the current statement to finish (which could take a little while if it is INPUT or PAUSE) and then executes a specified subroutine. No GOSUB will occur from a program that is entirely in machine code. The routine to be GOSUBed is selected by one of two methods. Either use:

```
CLOCK line number
```

with the number being between 8 and 9999 to distinguish it from a mode setting, or use:

```
CLOCK: statement: statement:...: RETURN
```

which will cause the rest of the line after CLOCK to be GOSUBed when the alarm goes off - but it will be ignored at other times. Within the subroutine you should use variable names different from those used by the main program, unless you intend to alter its operation. Alternatively, make the main part of the alarm routine a procedure, and use LOCAL variables. If you require the subroutine to save data, and the main program uses CLEAR or RUN, you will have to POKE the data into an appropriate area of memory, or it will be lost.

Possible applications for the subroutine include variations on the normal alarm - e.g. tune playing and graphic displays, chiming of the hours (the current time is available as a function and could be used by the routine - see TIME$). The clock speed can be changed by POKEs to address 56866, which holds "1/50ths. of a second per clock advance". POKE 56870 with 58 to allow 100 "seconds" per "minute", or use 54 to return to normal. Electronics enthusiasts could collect data from the outside world every minute or hour or whatever with something like this:

```
8999 STOP
9000 PRINT "Subroutine activated"
9010 LET pointer= DPEEK(USR "a"): POKE pointer, IN 127
9020 LET pointer=pointer+1: IF pointer >
     65535 THEN LET pointer= USR "a"+2
9030 DPOKE USR "a",pointer : LET z$=TIME$()
9040 LET hours = VAL z$(1 TO 2),mins = VAL z$(4 TO 5)
9050 LET mins = mins+1: IF mins = 60 THEN LET
     hours = hours+1, mins = 0
9060 CLOCK "a"+USING$("00",hours)+USING$("00",mins):
     RETURN
```

Don't RUN this! Instead, enter as a direct command:

```
DPOKE USR "a",USR "a"+2: CLOCK 9000: CLOCK 5
```

The direct command initialises a pointer (stored in memory locations USR "a" and USR "a"+ l) to point to the next location in the user graphics area. Line 9000 is selected as the one to be GOSUBed when the alarm time is reached, and the alarm GOSUB facility is turned on. Set the alarm to go off in the near future using:

CLOCK "AXXXX"

where "XXXX" is soon after the current time. Now RUN some other program! The CLOCK subroutine will be activated every minute; it gets a value from port 127 which is stored to a location in the user graphics area indicated by the regularly incremented pointer (which goes back to its start position if it reaches the top of memory) for later display (not done by this subroutine). If you don't have any port-mapped devices, you can use much of the program above (lines 9040-9060) to do something else at regular intervals. These lines reset the alarm for one minute after the current time, before RETURN to the main program.

CLS <window number>

See WINDOW for more detail.

CLS used on its own clears the current screen window. CLS with a number other than zero clears the specified window, if it has been defined. (If not, you will get an "Invalid I/O device" report.) CLS 0 will perform a normal CLS no matter what window is currently selected.

CONTROL CODES

Control codes are special characters that have actions when PRINTed (or PLOTed) rather than appearing on the screen. Beta Basic provides control codes related to the cursor (current PRINT or PLOT position) and control codes to control the special screen blocks created by GET.

CURSOR CONTROL CODES:

```
CHR$ 2  cursor left     not window-limited
CHR$ 3  cursor right    not window-limited
CHR$ 4  cursor down     not window-limited
CHR$ 5  cursor up       not window-limited

CHR$ 8  cursor left     window-limited
CHR$ 9  cursor right    window-limited
CHR$ 10 cursor down     window-limited
CHR$ 11 cursor up       window-limited
CHR$ 12 delete          window-limited
CHR$ 15 extra ENTER     window-limited
```

Beta Basic allows CHR$ 8 to 12 to work as one might expect when printed (e.g. PRINT CHR$ 10; will move the print position down one line, PRINT CHR$ 9; will move the print position right by one character position.) "Window-limited" means that if you have defined a WINDOW for printed output, the control codes cannot move the cursor outside the window. This is convenient for text processing. In contrast, control codes 2 to 5 do not have this restriction, and are more useful when PLOTing strings. (See PLOT command in this manual.) In fact, PLOT automatically converts CHR$ 8 to 11 to the unrestricted equivalents (CHR$ 2 to 5) before using them.

Spectrum Basic's CHR$ 8 (cursor left) has a bug - it is not possible to backspace the cursor onto the top line of the display from any lower line, and if you start on the top line, you can backspace off the screen (onto the program!). This has now been corrected.

Spectrum Basic's CHR$ 9 (cursor right) has a different bug that stops it working altogether. This has been corrected.

Spectrum Basic prints CHR$ 10, 11 and 12 as "?", which is not very useful. They now work as one might expect.

As one example, the control characters can be included in strings to allow complex shapes to be PRINTed or PLOTed rapidly and easily, once the string has been set up. (The GET command can do this for you if you wish to produce a rectangular shape.)

```
10 LET a$="1234"+CHR$ 8+CHR$ 10+"5"+CHR$ 8
   +CHR$ 10+"678"+CHR$ 8+CHR$ 11+"9"
20 PRINT AT 10,10;a$ 30 PAUSE 100: CLS
40 FOR n = 32 TO 255
50 PLOT n,n/2; a$: NEXT n
```

This can be very effective with user-defined graphics; we leave you to design them.

CHR$ 15 acts like ENTER in that printing continues on the next line, but it has some advantages. It can be entered from the keyboard as caps. shift-enter, but it does not cause the line to be accepted, so it can easily be entered into strings or program lines at any point.

SCREEN BLOCK CONTROL CODES:

CHR$ 0 eight bytes of shape information follow.

CHR$ 1 one byte of attribute information and eight bytes of shape information follow.

These control codes, in combination with cursor control codes 8 and 10, are used by GET to store an area of the screen as a string. You do not need to know this, but may find it interesting.

When PLOT encounters CHR$ 0, it knows that the next eight bytes in memory are not ordinary characters, but data about the shape of an 8-pixel block which is to be placed on the screen. This also applies to PRINT, provided a CSIZE other than 0 is in use (you can use CSIZE 8 for normal sized display). The way the shape is coded is the same as for a user defined graphic. It will be placed on the screen in the current PAPER and INK colours, just like a UDG. In fact you can make an array of 9-character strings, all starting with CHR$ 0, and use each string as a UDG. CHR$ 1 is slightly different in that PRINT and PLOT expect the shape data to be preceded by one byte of attribute data. This will be used when putting the shape onto the screen, rather than the current INK or PAPER, so it is rather like having permanently coloured UDGs.

GET stores a block of screen in a string by creating a series of CHR$ 0 or CHR$ 1-type squares, inserting cursor-down and cursor-back codes in order to place each square correctly in relationship to the other squares in the screen block.
Note: If you wish to experiment with these control codes, make up strings containing them and then print or plot the whole string at once, not one character at a time. For example:

```
10 CSIZE 8
20 LET a$=CHR$ O+CHR$ 255+CHR$ 129+CHR$ 129+CHR$ 129
   +CHR$ 129+CHR$ 129+CHR$ 129+CHR$ 255
30 PRINT a$: PRINT CSIZE 16;a$: PLOT 128,88;a$
```

COPY with strings and arrays

This command is closely related to the JOIN (strings and arrays) command;
see JOIN for details.

CSIZE width<,height>

Key: Shift 8

CSIZE controls the character size used for PRINT, LIST and PLOT. Rather like INK and PAPER, it affects everything printed if used on its own, but has a temporary effect when used in a PRINT statement. Width and height are given in pixels, and height is assumed to be the same as width if you don't bother to specify it. The example below shows character sizes up to only 4 times normal (CSIZE 32) but you can go up to CSIZE 255,176 with one letter filling the entire screen! With very large characters you will need to use break to stop an ENTER-generated listing.

```
10 FOR n=8 TO 32 STEP 8
20 CSIZE n
30 CLS
40 PRINT "CSIZE ";n
50 LIST
60 NEXT n
70 CSIZE 0
```

The larger characters are made by magnifying up the normal Spectrum character set; CSIZE 16 is *2, CSIZE 24 is *3, etc. Small changes in CSIZE can affect the spacing of characters without changing the actual character size. Get rid of the STEP in line 10 to show this. CSIZEs 8, 9, 10 and 11 use 8*8 characters, then CSIZEs 12 to 19 use 16*16 characters, etc. With some values, it is a good idea to enter OVER 1 to prevent the PAPER margins of characters partially overwriting the previous character.

Some interesting effects can be caused by LISTing after:

```
     INVERSE 1: CSIZE 9
or   CSIZE 32,8
or   CSIZE 8,32
```

So far we have dealt with larger characters, but we can also get smaller ones. CSIZE 3,8: OVER 1 will give 85 characters per line using a smaller character set, but this is not really legible unless you use lower-case letters and have a monitor. CSIZE 4,8 will give a legible 64 characters per line using the same character set. CSIZE 5,8 also uses these small characters, spaced further apart, but CSIZEs 6,8 and 7,8 use the standard character set with closer spacing. If you want 40 characters per line, use OVER 1: CSIZE 6,8 which gives 42 characters, then use the WINDOW command to reduce the screen width to 240 pixels (40 characters).

The printing routine used for the different character sizes is very versatile. It is able to print at any pixel position on the screen, at any size, controlled by a window. (See WINDOW.) It is therefore slightly slower than the normal print output. A special value of CSIZE, CSIZE 0, has the effect of returning to the normal print output routine. This will cause WINDOW 0 (the whole screen) to be selected. When Beta Basic is loaded, it is in CSIZE 0.

All the print control features such as AT, TAB, comma, cursor-back, cursor-down, etc., have effects which are correct for the current CSIZE. CSIZE 4,8 allows TABs as great as 63, for example.

CSIZE may be used with temporary effect within a PRINT or PLOT statement:

```
10 PRINT CSIZE 8,16;"double height";CSIZE 8;"normal";
    CSIZE 4,8;"small"
20 PLOT CSIZE 32,16;100,100;"AS"
```

User-defined graphics.

With character widths of less than 6 pixels, only the right hand side of each UDG is printed out. You can design your own 4*8 UDGs quite easily, perhaps using BIN with just four characters after it.

Block Graphics.

These are the usual shape, shrunk or magnified as required.

GET screen blocks.

The areas of screen stored in strings by GET (see GET command) consist of one or more 8*8 pixel shapes, which will be shrunk or magnified according to the current CSIZE. However, you should stick to CSIZE 4, 8, 16, 24, etc. if the edges of each shape are to join up correctly to form the screen block.

If you use CSIZE 4,8 the block will be squashed to half width by the loss of every alternate pixel. CSIZE 4,4 would also squash the block vertically by a similar method. Structures only one pixel wide may vanish, but a block containing, say, a filled circle, should look O.K., but smaller.

DEFAULT variable=value<,variable=value>....

Key: Shift 2

See also: Section on PROCEDURES.

DEFAULT is rather like LET, except that it does nothing if the specified variable already exists. So:

```
10 LET a=10
20 DEFAULT a=20
30 DEFAULT b=30
40 PRINT a,b
```

prints 10 for "a" because the DEFAULT in line 20 is ignored, and 30 for "b" because the DEFAULT in line 30 checks and finds that "b" does not exist - so it is created with the default value. ("Default" is often used in programming to mean "assumed unless specified otherwise". It can be an adjective e.g. "the default Microdrive is 1" or a noun e.g. "The default is 10".) As with LET (in Beta Basic) many assignments can follow the same command. Each variable is checked separately to see if it exists:

```
100 DEFAULT a$="aardvark",zx=123,q=0
```

The command could be used anywhere in a program, but it is primarily intended to be used inside a defined procedure so that the user may omit some of the parameters from the procedure call.

DEFAULT = SAVE/LOAD device

Key: shift 2

This command allows the normal tape SAVE, LOAD, VERIFY and MERGE commands to work with the Microdrives, the Net, or the RS232 link, and it simplifies the use of the CAT and ERASE commands.

As supplied, the "SAVE/LOAD device" setting is "t", which means "tape", and the SAVE, LOAD, VERIFY and MERGE commands will operate with a tape recorder if you use the normal syntax. You can change the DEFAULT (assumed) SAVE/LOAD device by commands such as:

```
DEFAULT = m     Microdrive 1
DEFAULT = M1    Microdrive 1
DEFAULT = m2    Microdrive 2
DEFAULT = n5    Net, station 5
DEFAULT = B     RS232 link, "b" channel
DEFAULT = t     Tape (normal)
```

As an example, after entering DEFAULT =m, the following commands would work with Microdrive 1:

```
SAVE "test"
SAVE 1;"test"
SAVE *"m";1;"test"
SAVE 10 TO 100;"test": REM see SAVE in this manual
LOAD "test"
VERIFY "test"
MERGE "test"
ERASE "test"
CAT
```

To use Microdrive 2 instead, either change the default drive by: DEFAULT =m2, or use:

```
SAVE 2,"test"
LOAD 2;"test"
ERASE 2,"test"
CAT 2
```

You can use a variable for the number part of the device specification, as in:

```
LET x=2: DEFAULT =mx
```

However, for convenience and the avoidance of quote marks, the device letter is not a string, and you cannot use a string variable in its place.

Even if the DEFAULT SAVE/LOAD device is the tape recorder, a command such as SAVE 1;“abc" or LOAD n,"prog" will use the microdrives, since the drive number makes it clear that the tape recorder is not wanted.

The current DEFAULT SAVE/LOAD device setting will be saved with Beta Basic's CODE if you SAVE the program.

Note: Beta Basic and Spectrum Extended Basic both allow "," and ";" to be used interchangeably in Microdrive commands

DEF KEY one-letter string; string
or **DEF KEY** one-letter string: statement: statement: ....

Key: Shift 1 (same as DEF FN)

See also: LIST DEF KEY

Beta Basic allows strings or program lines of any length to be produced by any number or letter key, with the characters produced being either entered into the computer or remaining at the bottom of the screen until ENTER is pressed. The latter case is selected by making the last character of the string a ":", or by following the last statement of the line by ":". (The ":" is not used except to tell the computer what to do.) Try:

```
DEF KEY "1"; "HELLO:"
```

Now press "symbol shift" and "space" together. The cursor will change to a flashing star. If you press "1", "HELLO" will appear at the bottom of the screen. Since the other keys have not been defined, if you had pressed any other key, you would have got only the normal value.

```
DEF KEY "a": PRINT "Goodbye"
```

In the above example, the part of the program line after DEF KEY "a": is assigned to "a" (or "A" - they are treated the same). It is not executed when the line is entered. Also, since the last statement is not followed by ":", when symbol shift/ space, and then "a" is pressed, the assigned statement will be entered and executed. If we had used:

```
DEF KEY "a"; "10 REM hello"
```

the line would have actually been added to the listing, after passing syntax checking.

A key may be assigned a different value at any time - the old definition is over-written. If a null string is used, or no statements follow the key definition, then the key will have no definition. DEF KEY ERASE will erase all the key definitions, which are stored above RAMTOP and otherwise protected, even from NEW. The SAVE routine in the Basic loader will save any definitions with the CODE part of Beta Basic. (It saves from RAMTOP to the end of Beta Basic.)

To list all the key definitions, use LIST DEF KEY. If you wish to edit the value assigned to a key, and you no longer have the DEF KEY statement available, you can type in a line number and then press the defined key, which will give you an editable line you can make into a DEF KEY statement.

RAMTOP is automatically lowered to accommodate the key definitions; if you use the normal CLEAR to change RAMTOP this will probably prevent any key assignments that you have made being found. However, Beta Basic provides a form of CLEAR that makes it easy for you to make extra space for machine code if you need to (see CLEAR).

DEF PROC procedure name < parameter> <, REF parameter>...
or **DEF PROC** procedure name < DATA >

Key: 1 (same as DEF FN)

See also: Section on PROCEDURES; PROC, END PROC, LOCAL, DEFAULT, LIST PROC, REF (parameter type specifier), function ITEM

DEF PROC starts the definition of a named procedure. DEF PROC must be the first keyword in the line (but preceding spaces and colour control codes are allowed). The procedure name used must start with a letter, and end with a space, a colon, ENTER, REF or DATA. Names can include most characters apart from "space", but we suggest that you keep to letters, numbers, "_" and perhaps "#"and "$", in the interests of clarity. Upper and lower case letters are equivalent. A procedure may have the same name as a variable without any confusion.

The procedure name can be followed by a list of parameters, or the DATA keyword. The parameters given in a procedure definition are called "formal" parameters, and they must be variable names. When the procedure is called, any variables with the same name as the formal parameters will be protected, before the values given after the procedure call (the “actual" parameters) are assigned to the formal parameters. When REF precedes a formal parameter name, this also happens, but in addition, the value of the formal parameter will be assigned to the corresponding actual parameter at the end of the procedure. Array names such as "a$" and "b()" can be used, but they must be preceded by REF.

When the DATA keyword is used instead of a parameter list, no assignments are made, but the list of actual parameters after the procedure call can be dealt with using READ and the function ITEM.

DELETE <line number> TO <line number>

Key: 7 (same as ERASE)

Removes all lines in the specified block from the program. If the first line number is omitted, the first line after line 0 will be assumed; if the second line number is omitted the last line in the program will be assumed.

```
DELETE TO 100      - deletes all lines after line 0 up to and
                     including line 100.
DELETE 100 TO      - deletes line 100 and all following lines.
DELETE 100 TO 100  - deletes line 100 only.
DELETE 0 TO 0      - deletes line 0 only.
DELETE TO          - deletes the entire program except line 0.
```

The last example is different from NEW, in that it does not CLEAR the variables. Any line numbers specified must exist, or you will get error U, "No such line". DELETE can be included in a program, with certain reservations. When parts of the program higher in the listing are deleted by a DELETE statement which is part of a subroutine, procedure, FOR-NEXT or DO loop, the program will usually stop since stored addresses will no longer correspond to the proper place in the program. If the DELETE statement is to remove itself from the program, it should be the last statement in the deleted block.

A possible application of DELETE within a program would be the removal of DATA statements once they have been READ, freeing extra memory for variables (numbers in DATA statements take up at least 8 bytes per value). A program could also delete parts of itself before MERGEing more lines into the freed space.

DELETE array name <slicer>
or **DELETE** string name <slicer>

Key: 7 (Same as ERASE)

See also: Section on DATA HANDLING

As well as being used to delete blocks of program lines, DELETE can eliminate all or part of an array or string:

```
10 LET a$="123456789"
20 DELETE a$(4 TO 7)
30 PRINT a$: REM prints "12389"
40 DELETE a$
50 PRINT a$: REM variable not found
```

If the entire string is deleted, it ceases to exist, rather than just having LEN 0. The command works similarly with string and numeric arrays. First, create an array and look at it:

```
10 DIM a$(10,4)
20 FOR n=1 TO 10
30   LET a$(n)=CHR$(64+n)+"xxx"
40 NEXT n
50 FOR n=1 TO LENGTH(1,"a$")
60   PRINT a$(n)
70 NEXT n
```

The function LENGTH is used in line 50 rather than "10", because the number of strings in the array will be changing. RUN the example, then add an extra line, such as:

```
45 DELETE a$(3)
45 DELETE a$(3 TO )
45 DELETE a$( TO 4 )
45 DELETE a$
```

and RUN again to see which bits have been deleted. With a numeric array, a command such as DELETE a(6) will remove the sixth number from a one-dimensional array, or a whole "row" of numbers from a two-dimensional array. DELETE a(3 TO 8) would delete a "slice" of individual numbers or a "slice" of rows. (The number of "rows" in a numeric array is set by the first dimension.)

DO
or **DO WHILE** condition
or **DO UNTIL** condition

Key: D (WHILE is key J, UNTIL is key K)

See also: LOOP, EXIT IF

DO and LOOP, together with their qualifiers WHILE and UNTIL provide a control structure which has some advantages over those normally provided by Basic. On its own, DO simply serves as a marker which a matching LOOP statement can loop back to:

```
10 DO
20   PRINT "HELLO ";
30 LOOP
```

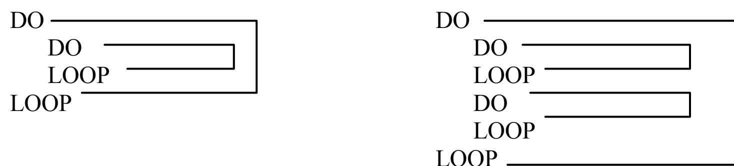
This keeps printing until break is pressed. DO can be qualified by WHILE (some condition). If the specified condition is true, the statements and program lines after DO will be executed until a LOOP is encountered. The part of the program between the DO and the LOOP is executed WHILE the condition is true. If the specified condition is false, the following part of the program is ignored until a LOOP is encountered; execution then starts with the statement after the LOOP. DO UNTIL (some condition) is exactly the opposite - the part of the program between the DO and the following LOOP will be executed only if the condition is false (or to put it another way, it is executed UNTIL the condition is true).

```
10 LET total = 0
20 DO UNTIL total > 100
30   INPUT "ENTER A NUMBER ";X
40   LET total = total + X
50   PRINT total
60 LOOP
70 PRINT "THAT IS OVER A HUNDRED"
```

Line 20 could be replaced by:

```
20 DO WHILE total <= 100
```

DO - LOOP pairs may be nested in the same way as FOR - NEXT loops. For example:



Since the address of the DO is stored, you should not jump out of a DO - LOOP unless you use EXIT IF or POP (otherwise, the address will clutter up the computer's stack). If DO is not followed at some point by LOOP, you will get report S, "Missing LOOP", if a conditional DO tries to skip over the DO - LOOP and can't find where it ends.

Control structures very similar to DO - LOOPs exist in some other computer languages under different names:

```
REPEAT (statements)       DO (statements)
UNTIL (condition)     =   LOOP UNTIL (condition)

REPEAT (statements)       DO (statements)
UNTIL FALSE           =   LOOP

WHILE (condition)         DO WHILE (condition)
```

```
(statements)          =     (statements)
ENDWHILE (or WEND)          LOOP
```

DPOKE address, number

Key: P

See also: DPEEK(address) function

DPOKE means Double POKE - the equivalent in Basic is:

```
POKE address, number - INT(number/256)*256
POKE address + 1,INT(number/256)
```

In other words, the least significant byte of the number is poked to the address, and the most significant byte is poked to the next (higher) address. Machine code stores numbers between 0 and 65535 in this way; many system variables have this format and are most easily altered with DPOKE. The function DPEEK provides the equivalent double PEEK.

DRAW TO x, y<,angle>

Use the two normal keywords.

This command draws a line from the current graphics position to the point x, y. This can often be more convenient than the normal draw command, which is relative to the last position. The example draws random lines which will never go off-screen.

```
10 FOR n=1 TO 100
20    DRAW TO RND*255,RND*175
30 NEXT n
```

You can use PAPER, INK, OVER etc. by placing them after the TO. If you use three numbers you can draw curved lines to a specific point. Some examples:

```
DRAW TO 10,10
DRAW TO INK 2;20,30 DRAW TO 100,90,1
```

EDIT <line number>

Key: 0 (zero; not graphics mode)

This is not the same thing as pressing shift 1! EDIT is a keyword - it is intended to get round the irritating sequence: LIST (some line), BREAK, shift 1. To be an improvement, EDIT really has to be an unshifted keyword - but all the letter keys are already used in this way, and the number keys are needed for line numbers. However, a line number doesn't normally start with zero, so this key can be used. The keyword EDIT is obtained if "0" is the first key pressed after a preceding ENTER. Even if the keyword does not appear, a zero at the start of a line number is taken to mean EDIT. (EDIT can also be obtained in Graphics mode as shift-5, but that is less convenient for this purpose.) If you follow EDIT (or zero) with a line number and press ENTER, the specified line is immediately ready to edit. If the line number is omitted, you will get the current line. Ease of editing (initiated by either shift 1 or EDIT) has been improved by allowing the up and down cursor keys to move the cursor within the line. The cursor will not go through keywords, but takes the first following gap. If you attempt to move the cursor higher than the top of the line or lower than the bottom, the cursor will jump to the end of the line. Use EDIT, then up-arrow, as a fast, easy way to add more statements to the end of a line.

EDIT string variable
or **EDIT** ;numeric variable

Key: Shift 5

As well as letting you edit program lines quickly, EDIT can be used to change the values of variables. For this purpose you usually cannot get the keyword EDIT by pressing the zero key - you have to use Graphic shift-5 or type the keyword in full in KEYWORDS mode 3 or 4. A typical application of this form of EDIT would be modification of the strings in an array holding names and addresses. It is easy to fill such an array using INPUT and some kind of loop, but it is normally difficult to modify anything without typing in entire strings again. Beta Basic allows you to use EDIT a$(n) to get the contents of string "n" back into the editing area, with the cursor at the end of the line. We will begin with some simple examples using non-array variables:

```
10 LET a$="Jon Brown"
20 EDIT a$
30 PRINT a$

40 LET num=365.253
50 EDIT ;num
60 PRINT num
```

In line 50, a ";" was used in order to distinguish EDIT- variable from EDIT-line number. A comma could have been used instead; this would have caused the edited variable to be printed at TAB 16.

EDIT has a syntax very similar to that of INPUT. You can use semi-colons, commas, AT, TAB, LINE and prompt strings in the same way as you would 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 lets you create an array and then edit it:

```
10 DIM a$(10,15)
20 FOR n=1 TO 10
30   INPUT a$(n)
40 NEXT n
50 PRINT "Editing"'
60 FOR n=1 TO 10
70   EDIT("record ";n;" ");a$(n)
80 NEXT n
```

The format used in line 70 is described in detail in the original Spectrum manual, page 104, for the INPUT command.

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

ELSE (statements)

Key: E

ELSE is part of the IF - THEN structure. Normally, when the statement following IF is false, program execution will jump to the next line. However, when the IF - THEN pair has an associated ELSE later in the line, the program will continue with the statements following the ELSE. When the condition after the IF is true, on the other hand, the line will only be executed up to the ELSE - then the next line will be jumped to. For example:

```
10 INPUT "Give me a number ";X
20 PRINT "Does that number = 1?"
30 PAUSE 50
40 IF X = 1 THEN PRINT "True": ELSE PRINT "False"
50 GO TO 10
```

The situation is more complicated if you have several IFs and ELSEs on the same line below are some examples to show you which ELSE will be used by different IFs when their specified condition is false.

If ELSE is used without IF - for example, as the first statement in a line - then the ELSE and the rest of the line will be ignored.

END PROC

Key: 3

See also: Section on PROCEDURES; DEF PROC

Marks the end of a named procedure, allowing the computer to avoid executing the statements between DEF PROC and END PROC unless the procedure is called. When a procedure is in use, END PROC terminates its execution, erases any LOCAL variables used and restores their original (global) values, if any. If parameters have been passed by reference (the formal parameter was preceded by REF) the current value of the formal parameter will be assigned to the actual parameter in the procedure call. If the formal parameter is not found, you will get a "Variable not found" report. Program execution returns to the statement after the procedure call. (See the PROCEDURES section for more details.) Using END PROC without DEF PROC gives report W, "Missing DEF PROC".

EXIT IF (condition)

Key: I

See also: DO, LOOP

This is part of the DO - LOOP structure - read about that first! EXIT IF is used to leave a DO - LOOP from somewhere in the middle, rather than at the DO or the LOOP. If the specified condition is true, program execution jumps to the statement following LOOP; otherwise nothing happens.

```
100 DO
110  PRINT "Line 110"
120  PAUSE 20
130 EXIT IF INKEY$ = " STOP "
140  PRINT "Line 140"
150  PAUSE 20
160 LOOP
170 PRINT "Out of LOOP"
```

This will loop until " STOP " (shift-A) is pressed. Note that "Line 140" is not printed when the DO-LOOP is exited. If no LOOP is found, you will get report S, "Missing LOOP".

FILL x, y
or **FILL** <INK colour;> x, y
or **FILL** <PAPER colour;> x, y

Key: F

This command fills an area of PAPER with INK if FILL or FILL INK is used, or fills an area of INK with PAPER if FILL PAPER is used instead, starting at coordinates x,y. If you try to FILL an area with INK and the point x,y and its surroundings are already INK, nothing will happen. Unlike the normal case in Spectrum Basic, the colour number may be omitted if a FILL with the current INK or PAPER colour is desired. For example, FILL PAPER; x,y is O.K.

```
10 FOR n = 1 TO 6
20   CLS
30   CIRCLE INK n;128,88,n*10
40   FILL INK n;128,88
50 NEXT n
```

It is possible to use a complex form of FILL such as:

```
FILL INK 2; PAPER 1; FLASH 1;x,y
```

In this case the first word after FILL determines whether INK or PAPER will be used, and the other words merely change the attributes of the filled areas.

Because it is not possible for more than two colours to occupy the same character square, it takes a little thought to produce some patterns without messy results where two areas of INK are adjacent. It becomes necessary to have the areas join along a character square boundary:

```
10 LET x=128,y=88,rad=70
20 LET L=(SQR 2)*rad/2
30 CIRCLE x,y,rad
40 PLOT x,y: DRAW -L,-L
50 PLOT x,y: DRAW L,-L
60 PLOT x,y: DRAW 0,rad
70 FILL INK 2;x-5,y
80 FILL INK 4;x+5,y
```

FILL will work on any shape. As an example of a complex shape try this, which will fill the whole screen except for the areas inside the "Q"s:

```
PRINT STRING$(704,"Q"): FILL 0,0
```

The method used works fastest if lots of memory is available - if you FILL a large blank area such as the whole screen you will see pauses towards the end as FILL checks its stored data for points it can forget about. In such cases it is faster to divide the area into several parts and FILL each separately.

The number of pixels filled during the last use of FILL is obtainable using the function FILLED().

A FILL can be interrupted at any time using BREAK.

GET numeric variable or string variable
(GET from the keyboard)

Key: G

Like INKEY$, GET is a way of reading the keyboard without the use of ENTER. The difference is that GET waits for a key to be pressed before continuing. Used with a string variable, GET obtains a one character string:

```
10 GET a$: PRINT a$;: GO TO 10
```

This works rather like a type-writer. You can shift between upper and lower case in the usual way. The cursor controls move the print position in all directions, and you can delete characters. ENTER moves to the start of the next line. A more sophisticated version of the above is:

```
10 GET a$
20 PRINT CHR$ 8;" ";CHR$ 8;a$; INVERSE 1;"B"; INVERSE 0;
30 GO TO 10
```

You might like to try changing to a different character size with the CSIZE command.

If GET is used with a numeric variable (e.g. GET x or GET key) then the variable will equal 1 if "I" is pressed, up to 9 if "9" is pressed. The value will be 10 for "A" or "a", 11 for "B" or "b", etc. GET is particularly useful in menu-driven programs (see ON).

GET string variable, x, y <,width , length> <;type>
(GET from the screen)

GET in this format is used to assign a block of the screen at the coordinates x,y to a string, which can then be PLOTed or PRINTed back elsewhere on the screen. (If you use PRINT, you must use a CSIZE other than 0 - you can use CSIZE 8 if you want normal size.) The size is given in character squares. In the simplest case, the name of a string variable is followed by the x and y coordinates of the upper left-hand corner of the block:

```
10 PRINT "QWE"
20 GET a$,4,175
30 PLOT 100,100;a$
```

Since no data follows the coordinates, the block size is assumed to be a one character square. The section of screen assigned to a$ contains the right half of "Q" and the left half of "W". Beta Basic's enhanced PLOT lets you put this section back anywhere, at different magnifications (see PLOT and CSIZE).

There is something strange here, though. How can a string contain a section of screen? (If you don't care, skip this bit!) If you PRINT LEN a$ you will find it is 9. The first character is a control code that says, "the next 8 characters are shape data". With larger blocks, cursor control codes are included to control the block shape. More details are given under CONTROL CODES.

Over the page is an example that draws a design on the screen, and then GETs this design as a 3*3 character square block in a$. This is then PLOTed back at random locations, much faster than it could be redrawn conventionally.

```
10 CIRCLE 10,165,10
20 CIRCLE 13,165,5
30 FILL 5,165
40 GET a$,0,175,3,3
50 PLOT RND*230,RND*150+20;a$
60 GO TO 50
```

If you are using OVER 0, you will see that the paper margins of the shapes over-write what is underneath. For different effects, enter OVER 1 or OVER 2 and run the example again. (OVER 2 is a Beta Basic feature that lets you PLOT without either over-writing or inverting the shape underneath.) If you wish to draw a lot of circles that are the same size, it is much faster to use GET and PLOT than CIRCLE. Try this by deleting lines 20 and 30 and entering OVER 2 (or use PLOT OVER 2;RND*... in line 50).

So far, the examples have been of type-0, which means "colourless". This is the assumed type if you do not specify one. A type-0 GET string has no colours of its own; it will be PLOTed in the current colours, or in the colours selected by local colour commands within the PLOT or PRINT statement.

To create a type-1 string, which is "coloured", put a ";" after the other parameters, then a "1". (To force type-0, use "0".) A type-1 string takes the attributes from the screen, as well as the pattern information. This information will be put back wherever the string is PLOTed, allowing different parts of the screen block to have different colours. Global or local colour commands will not affect this.

```
10 PRINT INK 2;"ABC"
20 PRINT: PRINT INK 4;"DEF"
30 GET s$,0,175,3,3;1
40 PLOT 100,100;s$
```

As usual, the fact that only 2 colours are possible per character square may cause problems in some cases. You may prefer to use PRINT AT in order to avoid this.

JOIN <line number>

Key: Shift 6 (same as &)

See also: SPLIT

Joins together the specified line (or, if one is not specified, the line with the current line pointer) and the one below, if there is one. The following line will lose its line number, and will be separated from the first line by a ":".

JOIN can be used together with SPLIT to remove part of a program line and add it to another.

JOIN strings and arrays
and COPY strings and arrays

Key: Shift 6 (same as &)

See also: Section on DATA HANDLING

The JOIN and COPY commands allow all or part of a string or array to be moved (with JOIN) or copied (with COPY) to any position in another string or array. Since the two commands are closely related, they will be discussed together.

Use with STRINGS.

```
JOIN source string <slicer> TO destination string<position>
COPY source string <slicer> TO destination string<position>
```

Here is an example:

```
10 LET a$="12345"
20 LET b$="ABCDEFG"
30 JOIN a$ TO b$
40 PRINT b$: REM prints "ABCDEFG12345"
50 PRINT a$: REM not found
```

Because JOIN was used and no slicer followed the source string, all of a$ was moved to b$, and a$ therefore no longer exists. If you change line 30 to:

```
30 COPY a$ TO b$
```

you will find that a$ is not affected when the program is run - it is copied to b$, not moved. The version with COPY appears to be the same as LET b$=b$+a$, but both COPY and JOIN will work even if the strings involved almost fill memory, whereas LET will not. (LET a$=a$+"x" will fail if a$ is longer than one third of the free memory.) As well as dealing with entire strings, a substring can be selected from the source string using a "slicer", and inserted into the destination string at any desired character position. If no "slicer" is specified for the source, the entire string is assumed. If no position is given, the last position plus 1 is assumed, so the strings will follow each other, as in the example above.

To demonstrate these features, try changing line 30 in different ways, such as:

```
30    JOIN a$(2) TO b$
      REM a$="1345", b$="ABCDEFG2"
30    JOIN a$(3 TO ) TO b$
      REM a$="12", b$="ABCDEFG345"
30    COPY a$ TO b$(3):
      REM a$="12345", b$="AB12345CDEFG"
30    JOIN a$(2 TO 3) TO b$(LEN b$+1)
      REM a$="145", b$="ABCDEFG23"
```

The last example could have been just:

```
30 JOIN a$(2 TO 3) TO b$
```

but it demonstrates that the position at which to add the new material can be as great as LEN+1. If you use greater values you will get a "Subscript wrong" report.

Here is an example that uses COPY, DELETE and INSTRING to search a string t$ and replace every instance of "Spectrum" (o$) with "Q L" (n$):

```
10 LET t$="The Spectrum is a versatile computer, but
      Spectrum owners may feel it should have better editing"
20 PRINT t$
30 LET n$="QL",o$="Spectrum"
40 LET p=1
50 LET p=INSTRING(p,t$,o$)
60 IF p<>0 THEN
      DELETE t$(p TO p+LEN o$-1)
      COPY n$ TO t$(p)
      GO TO 50
70 PRINT t$
```

Use with ARRAYS.

```
JOIN source array<slicer> TO destination array<position>
COPY source array<slicer> TO destination array<position>
```

Arrays are a convenient method of dealing with large amounts of data, but normally they have serious limitations. Perhaps the worst of these is that they are fixed in size once created. You may waste memory by DIMing an array bigger than you need, or you may find that you have run out of space. The latter is particularly annoying if there isn't enough space in memory to DIM another, larger array to transfer the data to.

Beta Basic makes array handling much more flexible. The JOIN and COPY commands allow all or part of an array to be moved (with JOIN) or copied (with COPY) to any position in another array. Space for the new material is made in the destination array so that nothing is overwritten. Arrays of one or two dimensions can be handled, which should cover most applications. One dimensional arrays are treated like two-dimensional arrays that have a second dimension of one; there is no need for the number of dimensions in the source and destination arrays to match.

It is not necessary for the strings or rows of numbers in each array to be the same length. When string arrays are being handled, the strings from the source array will be forced to the same length as those in the destination array by either cutting them short, or "padding" them with spaces. Numeric arrays are handled similarly, except that the "padding" is done with zeros.

Suppose that you have an array a$(100,30) which is full. To add another 20 strings you could use:

```
DIM b$(20,30): JOIN b$ TO a$
```

Because JOIN was used, strings will be actually removed from the array b$ and put into array a$, rather than copied (as with COPY). Because no slicer followed the source array name, all the strings will be moved, and array b$ will cease to exist (with COPY it would have been unchanged). The position in the destination array to move the strings to was not specified, and it will be assumed to be position 101 - one past the actual end of array a$. The first string from array b$ will become string 101 in array a$, and the last will be string 120. Because arrays can change their size in Beta Basic, you will often want to use the LENGTH function to find out how big an array is. In the example, LENGTH(1,"a$") will be 120.

If you use DIM b$(20,3) or DIM b$(20,50) and then JOIN b$ TO a$, the result will be the same: the strings of b$ will be padded out with spaces or cut short so that they can be part of array a$, which has 30-character strings. If you have an array and decide the strings in it should be longer, you can achieve this by DIMing a one-element array with the desired string length and JOINing the main array to it.

```
DIM b$(1,40): JOIN a$ TO b$
```

will pad each string in a$ to 40 characters and JOIN it to b$, without data loss. Unfortunately all our data is now in b$, not a$, so to get the name right, we would need to DIM a$(1,40): JOIN b$ TO a$. We also have generated two extra strings with our DIM statements which we can remove using DELETE a$(1 TO 2).

So far we have dealt with entire arrays, but JOIN and COPY are much more flexible than that. You can specify which strings or rows of numbers from the source are to be used, with a "slicer", and the position in the destination at which the new material will be inserted can also be specified.

The examples below use JOIN but work for COPY too - in which case the source array will not be affected.

```
JOIN a$ TO b$(4)
```

will insert the entire array a$ into b$ so that the first inserted string is the 4th. string in the enlarged array b$.

```
JOIN a$(2 TO 5) TO b$(1)
```

will move four strings from a$ to the start of array b$. Array a$ will be four strings shorter than it was, while b$ will be four strings longer.

```
     JOIN a$(10) TO b$
or   JOIN a$(10) TO b$(LENGTH(1,"b$")+1)
```

will move the 10th. string of a$ to the end of b$. The last position in the destination, plus 1, is the greatest value that can be specified before a "Subscript wrong" report is given.

If you JOIN a normal string to a string array, it is treated as an array of dimensions (LEN string, l).

Numeric Arrays

Numeric arrays of one or two dimensions are handled similarly to string arrays. Brackets must be used after the array letter to distinguish arrays from simple variables, even if you do not wish to use a slicer or a destination position. The example below creates two arrays and then JOINs them together:

```
10   DIM a(8)
20   FOR n=1 TO 8
30      LET a(n)=n
40   NEXT n
50   DIM b(5)
60   FOR n=1 TO 5
70      LET b(n)=n*10
80   NEXT n
90   JOIN b() TO a()
100  FOR n=1 TO LENGTH(1,"a(")
110     PRINT a(n)
120  NEXT n
130  REM prints 1 2 3 4 5 6 7 8 10 20 30 40 50
140  REM b() does not exist
```

KEYIN string

Key: Shift 4

KEYIN enters a string as though you had typed it in from the keyboard. This allows programs to be self-writing - the full implications of this are beyond the scope of this manual (and its author!). It is possible to automate the production of data statements:

```
10 LET A$ = "100 DATA"
20 FOR N = 0 TO 9
30 LET A$ = A$ + STR$(PEEK N) + ","
40 NEXT N
50 LET A$ = A$(1 TO LEN A$ - 1): REM chop off last comma
60 KEYIN A$
```

You will see that a new line has been added to the program after this is RUN. (A fairly useless line, since it gives data on the first 10 bytes of ROM, but it illustrates the principle.)

In KEYWORDS 3 or 4 all fully spelled-out keywords in the strings you KEY IN will be turned into single character keywords before the string is used.

KEYWORDS number

Key: 8

KEYWORDS 1 or 0 - PRINT and LIST control.

KEYWORDS used with 1 or 0 controls whether user-defined graphics (KEYWORDS 0) or Beta Basic's keywords (KEYWORDS 1) are displayed. The current entry mode (selected by KEYWORDS 2, 3 or 4) will not be affected. Initially, the system is in the KEYWORDS 1 state; when you want to use user-defined graphics execute KEYWORDS 0. This also allows you to stay in graphics mode indefinitely, whereas KEYWORDS 1 jumps out after ever keyword is entered. Although your listings may look confusing, programs will still run normally in this state. ("KEYWORDS" is the only keyword which cannot be turned off - the graphic on that key is in any case equivalent to "space".)

KEYWORDS 2 or 3 or 4 - entry control.

KEYWORDS 2, 3 and 4 control the form in which keywords can be entered into the Spectrum. On loading, Beta Basic is in the KEYWORDS 3 mode. The current entry mode will be saved if Beta Basic's CODE is SAVEd. There follows a description of each mode:

KEYWORDS 2

All keywords must be entered by the single key entry system, after various shifts if necessary. All the Spectrum Basic keywords are obtained as usual. Beta Basic's commands are entered in Graphics mode, and its functions are entered as FN, a letter, "$" or "("

KEYWORDS 3

The same as KEYWORDS 2, except that the entered line is examined for fully spelled-out keywords before being accepted, and the corresponding single-byte "token" is inserted. This mode is probably the most convenient, since all keywords can still be obtained in the normal way, as well as by spelling them out. (A leading space can be used to cancel "K" mode.)

KEYWORDS 4

There is no "K" mode unless you force it temporarily using symbol shift/enter. Keywords can be spelled out in full or entered via the various shifts. This mode will probably be preferred by users who want consistency with non-Sinclair computers.

Keyword Recognition (modes 3 and 4).

A minor limitation of modes 3 and 4 is that you cannot enter a variable name or a procedure name which is spelled the same as a keyword, because it will be changed into a keyword. However, a keyword can be part of a variable or procedure name without any problem.

Keywords will be recognised in upper or lower case. It is best to use lower case so that you can easily see what has been converted to tokens. The character, if any, before and after a keyword must not be a letter or "_":

```
printa
```

is not converted to PRINT a (it is assumed to be a procedure or variable name) but:

```
print a
```

will PRINT the variable "a", if it exists. Below are some examples of possible entered lines, and their form after "tokenisation":

```
print 10                PRINT 10
print fork,total        PRINT fork,total
alter to ink3,paper1    ALTER TO INK 3, PAPER 1 print
string$(10,"plot")      PRINT STRING$(10,"plot")
goto10                  GO TO 10
go to x                 GO TO x
gotox                   gotox
defproc pink            DEF PROC pink
mat_print               mat_print
```

The examples with GO TO show that internal spaces in the fully spelled-out form do not have to be included; "gosub", "exitif", "onerror", "defproc", etc. will be recognised as valid keywords. The "ink" contained in "pink" and the "print" in "mat print" are not recognised because a keyword must not have a letter or "_" preceding it.

LET variable=value<,variable=value>....

A minor improvement to LET allows a series of assignments to follow the keyword, separated by commas. This saves a certain amount of memory (one byte per LET avoided - and if you are wondering, the syntax modification itself uses only 16 bytes) and gives a neater listing:

```
10 LET x=1,y=2,z=3,a$="y", b$="n"
```

could replace:

```
10 LET x=1: LET y=2: LET z=3: LET a$="y": LET b$="n"
```

LIST <line number> TO <line number>
or **LLIST** <line number> TO <line number>

This syntax is now allowed, in addition to the normal version. The specified block of lines will be LISTed or LLISTed. The first or the second line number may be omitted. If the first line number is omitted, then the first line of the program after line 0 will be assumed. If the second line number is omitted, the last line of the program will be assumed (this is equivalent to the action of the normal LIST syntax). Some examples:

```
LIST 20 TO 100
LIST TO 200
LLIST 100 TO 180
```

If the first line number exists, it will be LISTed with a ">" cursor. If it does not exist, the next following line number will be used, but no cursor will appear.

If the second line number does not exist, the next following line number, or the end of the program, will be used.

If the two line numbers are the same, only one line will be LISTed or LLISTed.

LIST DATA
or **LIST VAL**
or **LIST VAL$**

These variants of the LIST (or LLIST) command list the current values of the variables.

LIST DATA lists the values of all the variables.
LIST VAL lists the values of all numeric variables.
LIST VAL$ lists the values of all string variables.

The Spectrum has 6 kinds of variables. LIST VAL lists the four numeric types, in this order:

1. Numeric arrays.
2. FOR-NEXT variables.
3. Single-letter numeric variables.
4. Multi-letter numeric variables.

LIST VAL$ lists the 2 string types, in this order:

5. String arrays.
6. Ordinary string variables.

LIST DATA lists all the types, from 1 to 6.

For each type, the variables are given in alphabetical order (for multi-letter numeric variables, only the first letter is considered). LIST DATA might produce something like this:

```
d(10,4)
k(3,3,4)

n STEP 1      500     LN 200

g             3.5
j             100
s             23.1
apples        1
number        9999
xos           0
xrg           256
yos           0
yrg           176

t$(100,10)

a$ LEN 5      "Hello"
b$ LEN 40     "Too long to dis
e$ LEN 5      "Bang!"
```

The dimensions of arrays are shown, but not their contents. FOR-NEXT variables are distinguished by a STEP value and LN (the looping line number). The maximum value of the loop is not shown. Only the first 15 characters of long strings are shown.

LIST DEF KEY

DEF KEY is key shift I

See also: DEF KEY

The command LIST DEF KEY will list the contents of any user-defined keys, in order. The key number or letter is given, then the string or statements that have been assigned to that key. If there is a final colon shown in the assignment for a key, this means that when the key is pressed ENTER will be suppressed; i.e. the assigned value will just appear at the bottom of the screen.

LIST FORMAT number

This command uses two Spectrum Basic keywords to control the listing format of the program. Listing does not actually occur until ENTER is pressed. LIST FORMAT 0 is the initial state of Beta Basic on loading. The full range of formats is:

number

0 This gives a listing similar to that of Spectrum Basic, the only difference being that program lines longer than one screen line are indented so that only line numbers occupy the left-hand five columns of the screen. This improves clarity, and is a feature shared by all the formats.

1 LIST FORMAT 1 gives a "pretty" listing, in which every statement is on a new line. Certain keywords (see below) cause automatic indentation of statements by 1 space.

2 The same as format 1, but the automatic indentation is by 2 spaces.

3 The same as format 0, but without line numbers.

4 The same as format 1, but without line numbers.

5 The same as format 2, but without line numbers.

Automatic indentation by two spaces is conventional, but if you are only using 32 characters per line it may be better to use only one space (LIST FORMAT 1). The following keywords change the degree of indentation of the program when automatic indentation is working:

DEF PROC, DO and FOR indent all following statements (for the rest of the program, potentially) by one or two spaces, until cancellation by END PROC, LOOP or NEXT.

IF, ON ERROR and ON indent statements in the rest of the line by one or two spaces.

ELSE and EXIT IF cancel one or two indentation spaces for the current statement only.

You may want to follow THEN or ELSE with a colon to force a new line. The following example shows the same listing in FORMAT 0 and FORMAT 2:

```
100 DEF PROC useless
110 FOR n=1 TO 10: PRINT "prett
     y": NEXT n
120 DO: PRINT a$: INPUT b$
130 PRINT "abc": LOOP
140 IF x=1 THEN: PRINT "yes": P
     AUSE 50: ELSE: PRINT "no"
150 END PROC
```

```
100 DEF PROC useless
110  FOR n=1 TO 10
        PRINT "pretty!"
     NEXT n
120  DO
     PRINT a$
     INPUT b$
130  PRINT "abc"
    LOOP
140  IF x=1 THEN
        PRINT "yes"
        PAUSE 50
     ELSE
        PRINT "no"
150 END PROC
```

Your first impression is probably that FORMAT 2 looks a bit odd, but I hope your second impression is that it is easier to read. The full effect, including indentation that extends for several lines (like that produced by the DEF PROC) is only apparent if you list a whole screenful of lines at once, using LIST (or LLIST) or ENTER.

LIST PROC procedure name

PROC is key 2.

LIST (or LLIST) PROC will list a specified procedure from beginning to end. For example:

```
LIST PROC box
```

This is particularly convenient in long programs when you have forgotten where the procedure is. Sometimes you might want to use a variable for the procedure name; LIST PROC cannot deal with this directly, but you could use:

```
10 INPUT a$: KEYIN "list proc "+a$
```

Two system variables hold the first and last line number of a procedure after LIST PROC, and this can be useful in DELETEing or RENUMbering a procedure; they are locations 23625 and 57358 respectively. They can be conveniently examined with the function DPEEK.

LIST REF

REF is key shift 7

See also: REF

LIST REF gives a list of the line numbers in which a specified "reference" occurs. A "reference" can be a variable name, a number, or a sequence of characters. LIST REF is related to the REF command, which describes in more detail what a "reference" is - read about REF first.

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

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

then LIST REF n gives:

```
10
10
20
```

To direct the output to a printer, you can use LLIST REF, or

```
LIST #(stream number) REF reference
```

LOCAL variable <,variable> <,variable>...

Key shift 3

also: Section on PROCEDURES; DEF PROC

LOCAL allows the creation of special variables that only exist inside defined procedures. (The parameters of the procedure are automatically "local" and do not need to be put in LOCAL statements.) This allows procedures to be used without the possibility that they will interfere with the operation of the program they are used in by changing the values of "main program" (or "global") variables which happen to have the same name. A variable such as x or a$ can exist within the procedure, while at the same time the main program has a different value for x or a$,

LOCAL should only be used inside a defined procedure, or you will get a "missing DEF PROC" report. The best place to use it is just after the DEF PROC, so that it is immediately clear which variables are to be used. You can have several LOCAL statements within a single procedure if you wish. If the variables specified in the LOCAL statement exist, their current values will be "hidden". (Actually, they are still present in the variables area, so CLEAR will destroy them, but they are invisible as far as the procedure is concerned.) The procedure can then re-use the variable names. At the end of the procedure, the values which have been assigned to these "local" variables during the procedure will be erased, and the original values (if any) will be made available again. Here is a simple example:

```
10  LET n=1
20  test: REM use a leading space if you need to
     cancel K mode
30  PRINT n
40  STOP
100 DEF PROC test
110 LOCAL n
120 LET n=999
130 PRINT n
140 END PROC
```

Run the above to demonstrate that n=999 inside the procedure, and n=1 outside it. If you omit line 120, n will not exist inside the procedure. If you omit line 10, n will not exist outside the procedure.

You can skip this paragraph if you are using procedures for the first time - it is potentially confusing! If the procedure "test" calls another procedure (say, at line 125) then the variable n will appear to be a global variable with value 999 as far as this "sub procedure" is concerned. The procedure "test" is the sub-procedure's "main-program". All the variables available inside "test" will considered "global" by sub-procedures it calls (unless you make them LOCAL to the sub-procedures by using them in a LOCAL statement or as parameters).

Unlike most Basics, Beta Basic supports LOCAL arrays. If you use, for example, LOCAL a$ then any current string or array called a$ will be "hidden", and it will be possible to use he name a$ inside the procedure either as a string name or in a DIM statement without any effect on the main program. To make a numeric array LOCAL use brackets after the letter: e.g. LOCAL b(). This will "hide" any existing array b(). Then use a separate DIM statement to create a local array b() of the size you want.

LOOP
or **LOOP WHILE** condition
or **LOOP UNTIL** condition

Key: L (WHILE has key J; UNTIL has key K)

See also: DO, EXIT IF

Part of the DO - LOOP structure - read about DO first. LOOP on its own simply causes program execution to jump back to the matching DO statement. The qualifiers WHILE and UNTIL allow a conditional loop back - LOOP WHILE (condition) loops only if the specified condition is true; otherwise the statements and lines after LOOP are executed. LOOP UNTIL is the opposite - the loop is not made if the specified condition is true, and is made if it is false. You get report T, "LOOP without DO" if you use LOOP without a matching DO statement.

MERGE

See also: DEFAULT (LOAD/SAVE device)

Normally, trying to MERGE an auto-running program from the Microdrives will cause an error message. Beta Basic will MERGE such programs successfully, although auto-running will be prevented. This can often be convenient.

MOVE

If you have a Microdrive, the MOVE command will now let you MOVE programs, machine code or arrays, as well as data files. Use the normal syntax and avoid the use of the keywords CODE and DATA, even if they seem appropriate. You would transfer a file called "test" from drive 1 to drive 2 with the following:

```
MOVE "m";1;"test" TO "m"; 2; "test"
```

This will work whether "test" is a program, CODE, array or data file. Although MOVE is often convenient, you will find that large files are more rapidly transferred by using LOAD followed by SAVE.

ON

Key: O

Beta Basic supports two distinct forms of ON statement. The first form allows a particular line number in a list of line numbers to be GOSUBed or GOTOed, according to the value of the numeric expression immediately after ON. This is the conventional use of ON in most Basics, but its use of a list of line numbers is rather old-fashioned in an era of procedures. The second form allows a particular statement to be selected according to the value of the numeric expression after ON.

First form:

GO TO or GOSUB ON number; line no.,line no.,line no....

(The more usual syntax is ON number GO TO line number, line number,... but the Spectrum's keyboard system makes this difficult.) This form of ON is more flexible than the usual Spectrum alternative, such as:

```
10 INPUT choice: GO TO choice * 100 + 100
```

since the line numbers don't have to be in any particular sequence:

```
10 INPUT choice : GO TO ON choice;90,135,60,40
20 PRINT "Enter 1 to 4!": GO TO 10
```

In the above example, line 90 will be GOTOed if choice = 1, line 135 if choice = 2, etc. (If choice is negative, its absolute value will be used.) If it is not in the range 1 to 4, none of the lines is GOTOed; instead, the program continues with the next statement or line, which in this case forces the user to try again. INPUT could be replaced by GET in the above example for an elegant way of implementing menu-driven programs.

Second form:

ON number: statement: statement: statement......

This form of ON allows one statement from the following part of the line to be executed. Only the specified statement will be executed before the program continues with the next line. If the numeric expression after ON is greater than the number of statements following, the program will continue with the next line. Here is a simple example which is designed to respond to 1, 2 and 4. No response is made to an input of 3 because the third statement after ON is "empty".

```
10 INPUT x
20 ON x: PRINT "one": PRINT "two":: PRINT "four"
30 GO TO 10
```

There is no need for the statements to be of the same type. The following (non-functional) example mixes a procedure, GOSUBs and PRINT, and is listed in LIST FORMAT 2.

```
10 DO
20    GET number
      ON number
      GO SUB 100
      sound
      GO SUB 200
      PRINT "bye"
30 LOOP
```

ON ERROR line number
or ON ERROR: statement: statement:...

Key: N

Two forms of the ON ERROR statement are possible. The first form specifies a line number which will be GOSUBed if an error occurs. The rest of the statements in the same line as ON ERROR have no special relationship to it. The second form does not specify a line number; instead, the remainder of the line (the statements after ON ERROR) will be GOSUBed when an error occurs.

Any of the reports listed in the Sinclair manual or the Beta Basic manual apart from report 0, "OK" and report 9, "STOP statement" will cause an error GOSUB. The facility is turned off by ON ERROR 0, but it is also turned off automatically during the error handling subroutine, and turned on again when you RETURN to the main program. (An error handling routine that GOSUBed itself would be confusing!) Three special variables are available to the subroutine; these are not keywords - they must be entered in full. LINO and STAT are the line number and statement number where the error occurred, and ERROR is derived from the report code of the offending error - see Appendix C for details. You can use these variable names yourself - but the values will be over-written if ON ERROR or TRACE are active. It is usually desirable to have all but one or two reports handled normally, or you may become confused as to what is happening. In any case, your programs should not suffer from multiple errors! An example (which would be only part of a program) is given in which a series of points is plotted and the error routine simply skips any which are off-screen. The first form would be:

```
100 ON ERROR 5000
110 FOR n=1 TO 10: INPUT "x coord ";x;"y coord ";y
120 PLOT x,y: NEXT n

4990 STOP
5000 IF error=11 AND lino=120 THEN RETURN : ELSE POP :
     CONTINUE
```

Points to note: The STOP statement is intended to prevent the subroutine being executed accidentally. The value of both lino and error are tested because "Integer out of range" is a fairly common report. The RETURN from the error handling sub-routine at line 5000 will be to the statement after the one that caused the error - so RETURN is to "NEXT n". If the line or the error are not as specified, CONTINUE is used instead. This results in the offending statement being re-executed (unless the report was "BREAK into program" - see Spectrum manual Appendix B) and since CONTINUE does not re-enable ON ERROR, this time the normal response to the error is made. (The POP gets rid of the return address in the main program - this would otherwise clutter up the computer's stack, if RUN or CLEAR are not used.)

If we use the second form of ON ERROR we can discard lines 4990 and 5000. The new line 100 (using LIST FORMAT 2) will be:

```
100 ON ERROR
     IF error=11 AND lino=120 THEN
        RETURN
     ELSE
        POP
        CONTINUE
```

One "error" which has to be handled a little differently from the rest is "BREAK into program". Since ON ERROR is turned off when the error-handling subroutine is GOSUBed, the usual result of pressing BREAK is that the program stops after the first statement of the subroutine (because you are still pressing BREAK). Therefore, if you want to take some action when BREAK is pressed, you will have to introduce a delay in the first statement of your subroutine so that you have time to stop pressing BREAK. BEEP serves quite well - you can use an almost inaudible frequency. For example:

```
100 ON ERROR 5000
110 PRINT "round and ";: PAUSE 10: GO TO 110

4990 STOP
5000 IF error = 21 THEN BEEP 1,69: BORDER
     RND * 7: RETURN: ELSE POP: CONTINUE
```

Here is an example of ON ERROR used in Microdrive operations:

```
100 INPUT "FILE TO LOAD FROM? ";f$
110 ON ERROR 130
120 LOAD 1;f$: GO TO 140
130 POP: IF error=61 THEN PRINT "File ";f$;
    " not found": GO TO 100: ELSE STOP
140 ON ERROR 0
150 REM rest of program
```

Notice that this error routine is only used by this part of the program and has a very specific action - it prevents the program from crashing if an attempt is made to load a non-existent file. ON ERROR is turned on just before the LOAD operation, and is turned off again afterwards.

OVER 2

See also: GET screen blocks, PLOT strings.

OVER will now accept a value of 2, as well as the normal 0 and 1. OVER 2 causes letters or shapes placed by PLOT (string) to be added to what is already on the screen, rather than over-writing it or inverting it at the points of overlap. Because a pixel will be INK if it was already INK or if it is being set to INK by the new shape, this is often called "ORing" whereas OVER 1 is XORing (the functions section explains the difference). OVER 2 will make no difference to the plotting of points or to DRAW, or to printing if you are using CSIZE 0. However, it works with printing at any other CSIZE.

PLOT X coord, Y coord <; string>

(Normal keyword)

See also: GET screen blocks, OVER 2, CSIZE, CONTROL CODES.

As you can see from the syntax above, Beta BASIC allows you to PLOT a string, as well as the usual pixels. These can be normal strings or the screen blocks stored in strings by GET. The coordinates of the PLOT statement refer to the location that the upper left-hand corner of the first character of the string will be plotted to. All the usual PLOT qualifiers can be used, such as INVERSE, OVER, INK, etc. If the string extends off the right-hand side of the screen, it will "wrap round" onto the left-hand side. If the top of a plotted character extends off the top or bottom of the screen, an "Integer out of range" report will be produced. (It is however possible for the bottom seven pixels of a character to extend into the editing area at the bottom of the screen.) The plot position that DRAW uses as its starting point is not altered when a string is plotted.

By changing the coordinates to which you PLOT a character appropriately, you can achieve much smoother movement than is possible with PRINT AT:

```
100 FOR X = 16 TO 224: PLOT X, X/2;"<>": NEXT X
```

Try adding STEP 2, 3 or more to the FOR statement for faster speed and a different effect. Since "0" has a PAPER margin at least one pixel wide around it, it will automatically erase its previous position if it is moved by one pixel at a time. Some letters, such as "T", have INK pixels that extend to the edge of the character square. This means that they may leave a "trail" if moved in certain directions unless you explicitly erase it by overplotting the previous location. If you are designing your own characters, it is often a good idea to leave a one-pixel PAPER margin round them.

You can magnify or shrink the size of the plotted characters, graphics or blocks of screen (see GET from the screen) by using CSIZE (see CSIZE for more details). The CSIZE keyword must immediately follow PLOT - for example:

```
PLOT CSIZE 32;INK 2;100,88;"HI!"
```

The cursor control codes CHR$ 8-CHR$ 11 can be included in plotted strings to give them complex shapes. See CONTROL CODES for more details.

The ability to PLOT strings is useful when labelling graphs and diagrams since in addition to greater precision, the use of common coordinate system is convenient. It is simple to label the axes of a graph directly opposite the "tic" marks.

POKE address, string

(Normal keyword)

Beta Basic allows you to POKE strings as well as numbers, which in combination with the function MEMORY$ allows rapid manipulation of large areas of memory. (It should perhaps be said, that if you can crash a computer with an ill-judged POKE, this goes many times over if you are POKEing long strings!)

Let's POKE a string where we can see some effect:

```
10 LET screen = 16384
20 POKE screen, STRING$(6144,"U")
```

(STRING$ is explained in the functions section.) We have filled screen memory with "U"s, which because of their location, are no longer really "U"s, but pattern data. Since "U" is "01010101" in binary, the pattern is stripey. The next example copies the start of ROM to the attributes file, where it produces some colourful effects:

```
30 LET attributes = 22528
40 POKE attributes, MEMORY$()(1 TO 704)
```

Now we will write something simple to put a shape on the screen, store it in a string, and then POKE it back:

```
10 CIRCLE 128,88,70
20 FILL 128,88
30 LET a$ = MEMORY$()(16384 TO 23295): REM whole screen
40 CLS: PRINT "Hit any key": PAUSE 0
50 POKE 16384, a$
```

The Spectrum's memory will hold several such pictures, allowing you to swop them or show them in sequence. For more pictures, a third of the screen can conveniently be read into a string, if all that is wanted is the pattern information (you could still add a third of the attributes file to the same string or another string, of course.) To store the three thirds of the screen in strings use one of these:

```
TOP:    LET a$  = MEMORY$()(16384 TO 18431)
MIDDLE: LET a$  = MEMORY$()(18432 TO 20479)
BOTTOM: LET a$  = MEMORY$()(20480 TO 22527)
```

There is enough memory to make a reasonable cartoon using sequential POKEing of such strings. You could use an array (e.g. DIM a$(10,2048)) to hold the data.

Of course, there are many other potential uses besides playing with screen memory. We can CLEAR a large area of high memory, and store a whole program there:

First, CLEAR 33900, then RUN this:

```
10 POKE 34000, MEMORY$()(23552 TO 33800)
20 REM rest of long program
```

You can now use NEW, then get the program back with:

```
POKE 23552, MEMORY$()(34000 TO 44248)
```

The above could hide from NEW on a user-defined key which is defined after the CLEAR statement and before the program is RUN:

```
DEF KEY "j": POKE 23552, MEMORY$()(34000 TO 44248)
```

When the program is POKEd back, it will continue running where it left off, since all the system variables were saved, as well as the normal variables. It is slightly harder to arrange to swop programs; you will need to have two storage areas, as well as room to run a program.

One last idea for POKE: to SAVE machine code with a Basic program, assign it to a string using MEMORY$, and SAVE the Basic program including the string in such a way that an auto-running POKE will move the code back into place on LOADing. This is considerably faster than loading code separately. You can use Beta Basic's CLEAR command to make room for machine code without losing the variables (see CLEAR).

POP <numeric variable>

Key: Q

POP removes an address from the GOSUB/DO-LOOP/PROC stack. The indicated line number is assigned to a variable if one is used. This word makes it possible to jump out of subroutines, DO-LOOPs and DEF PROCs without cluttering up the stack with unused addresses. POP used alone simply junks the value, but POP loc (for example) makes the value available as the variable "loc". The subroutine or procedure then "knows" where it was called from. If you regret the POP it is still possible to do something similar to a RETURN by GO TO loc+l (this is not identical to RETURN since you cannot specify the statement number).

```
100 GOSUB 500
110 STOP
500 POP loc
510 PRINT "Subroutine called from line ";loc
520 GO TO loc+1
```

If you replace line 520 with:

```
520 RETURN
```

you will get "RETURN without GOSUB" since the required return address is no longer present on the stack.

Use of POP when there is no data on the stack to POP gives report V, "No POP data".

PROC name< parameter><,parameter><parameter>...
or name< parameter><,parameter><,parameter>...

Key: 2

See also: Section on PROCEDURES; DEF PROC

The PROC keyword is no longer needed to call procedures in Beta Basic 3.0. It has been retained largely to maintain compatibility with earlier versions of the program. A procedure name can be typed in on its own by cancelling K-mode with a leading space, or by entering your programs in KEYWORDS mode 4 (which turns K-mode off). The PROC keyword is however used to list a procedure, as in LIST PROC ellipse (see LIST PROC).

READ LINE string variable <,string variable>...

See also: Section on PROCEDURES.

Use of the two existing keywords in this way allows READ to work with DATA which would normally need quote marks. For example:

```
100 DATA dog,rat,fish,frog,z$,12,"?*+"
110 READ LINE a$
120 PRINT a$: GO TO 110
```

The DATA statement limits the kind of things which can be stored in this way, since only valid expressions are allowed. In line 100, "cat" could be a numeric variable, but "?*+" needed quote marks to be accepted. You could mix both types by adding:

```
115 IF a$(1)=CHR$ 34 THEN LET a$=VAL$ a$
```

READ LINE makes string DATA statements easier to write; however, its main use is in allowing procedures to handle strings without the use of quote marks.

REF reference

Key: Shift 7

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

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  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 used are in capitals or not.

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 "counts". 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.

REF variable name

Key: Shift 7

(We recommend that you read the section on PROCEDURES before continuing.) REF can be used to specify that a parameter in a procedure definition is to be passed by reference; that is, during the procedure, the corresponding variable in the procedure call will be temporarily re-named with the name after REF. This allows values to be passed out of procedures, as well as into them. Arrays must be passed by reference. In the example below, REF a$ refers to a string or array, REF b() refers to a numeric array.

```
100 DEF PROC crunch REF a$, REF b(), REF output
```

RENUM <*><start TO finish> <LINE new start> <STEP step >

Key: 4

RENUM provides very powerful renumber, block move and block copy facilities. RENUM used by itself renumbers the entire program so that line 10 is the new first line and the interval between lines is 10. A block of lines to be renumbered can be specified by a "slicer" type expression following RENUM:

RENUM 130 TO 220 - renumbers the specified block

RENUM 130 TO - renumbers line 130 and all following lines.

RENUM TO 100 - renumbers all lines up to and including line 100, except line 0.

A renumbered block will be moved to the appropriate position in the program, provided there is room for it at the destination. If there is not, you will get report G, "No room for line" (which in this context doesn't have the meaning given in the Sinclair manual). RENUM * will act as a block copy - a copy of the specified block is moved and renumbered, but the original lines still remain. A new first line number other than the assumed value of 10 can be set by using LINE (the normal keyword). An increment value between lines other than 10 can be set by STEP (the keyword). The block-specifying slicer, LINE and STEP can all be included or omitted as required, but they must be used in the specified order. Some examples:

```
RENUM
RENUM LINE 100 STEP 20
RENUM 100 LINE 300: REM one line only
RENUM 1540 TO LINE 2000
RENUM 100 TO 176 LINE 230 STEP 5
RENUM * 10 TO 100 LINE 500: REM block copy
```

Note: Early versions of Beta Basic required the slicer to be in brackets. This format will still be accepted.

References to the renumbered block anywhere in the program are altered as required, including GOTO, GOSUB, RESTORE, RUN, ON, ON ERROR, TRACE, LIST, LLIST, LINE and DELETE. CLOCK is not taken care of - you will have to do this yourself! (This is because CLOCK can be followed by a line number or a mode setting.) RENUM will be unable to renumber a statement such as GO TO In*10. The location of any such failures will be printed out when the renumbering is complete; for example:

```
Failed at 100:2
Failed at 230:4
```

Any GOTO, GOSUB, etc. followed by an expression will always be reported, (even with RENUM of a small program section) since it could refer to the renumbered section. If you wish to direct the "failed" messages to a printer, use:

```
OPEN #2,"p": RENUM: OPEN #2,"s"
```

Note: RENUM creates a data table in screen memory - this is cleared as soon as the renumber is completed, and it is no cause for alarm!

ROLL direction code<,pixels><;x,y; width,length>

Key: R

See also: SCROLL

ROLL moves the entire screen or a defined screen window up, down, left or right. Anything moved off the edge of the ROLL window will reappear on the opposite side of the window. In other words, the command does not destroy anything on the screen - it simply re-arranges it (unlike SCROLL).

As you can see at the top of the page, ROLL can have a fairly complex syntax. However, most of this is needed only in order to define the area to be ROLLed. If you want to move the whole of the current WINDOW (which will be the whole screen unless you have changed things) by one pixel, you can simply use:

```
10 ROLL direction code
```

For a ROLL of the screen pattern information only, the direction of ROLL is specified by using 5, 6, 7 or 8 after the command. The direction arrows are printed on the keyboard (unless you have a Spectrum+); 5 is left, 6 is down, 7 is up and 8 is right. Because each use of ROLL only causes a small movement, the command is best used in a loop. DRAW a large graphic, or LIST a program, then try:

```
100 FOR d = 5 TO 8: FOR p = 1 TO 100
110 ROLL d
120 NEXT p: NEXT d: STOP
```

Diagonal ROLLing can be obtained by, for example, repeatedly ROLLing one pixel left, then one up. For faster (if less smooth) motion, change line 110 to:

```
110 ROLL d,4
```

which will move 4 pixels at a time. This "pixels to move" number should be less than 256 for horizontal movement, or less than 177 for vertical movement. If not specified, it is assumed to be one. Obviously, the bigger the number, the further the FOR-NEXT loop will move the screen. In the vertical direction, speed of movement is roughly proportional to the number of pixels moved at once. In the horizontal direction, 4 and 8 pixel moves give the best speeds, since they exploit half-byte (nibble) and byte moving instructions possessed by the Z-80 chip.

In order to move the attributes (colour) information with the pattern information, add 4 to the direction code; to move <u>only</u> the attributes, subtract 4.

DIRECTION CODE	DIRECTION	APPLIES TO:
1	LEFT	ATTRIBUTES
2	DOWN	ATTRIBUTES
3	UP	ATTRIBUTES
4	RIGHT	ATTRIBUTES
5	LEFT	PATTERN DATA
6	DOWN	PATTERN DATA
7	UP	PATTERN DATA
8	RIGHT	PATTERN DATA
9	LEFT	BOTH
10	DOWN	BOTH
11	UP	BOTH
12	RIGHT	BOTH

The attributes can only be moved by 8 pixels at a time (the pixels-to-move number is ignored). When direction codes 9-12 are used to move the attributes and the pattern data at the same time, the best alignment possible will be maintained. You can see this happening by using:

```
10 PRINT AT 10,10; PAPER 2;"DEMO"
20 ROLL 9: PAUSE 10: GO TO 20
```

The attributes are kept within 4 pixels of their corresponding pattern data. Now change line 10 to:

```
10 PRINT AT 10,10; INK 2;" DEMO "
```

The extra spaces surrounding "DEMO" ensure that it is always covered by the 6-character strip of red INK. A similar method will work for other shapes. The example below creates filled circles in different colours; each is surrounded by a protective ring of the appropriate attributes by drawing an invisible circle just outside it:

```
10 LET y=88,r=15
20 FOR n=1 TO 4
30   LET x=n*48+8
40   CIRCLE x,y,r
50   FILL INK n;x,y
60   CIRCLE INK n; INVERSE 1; OVER 1;x,y,r+5
70 NEXT n
80 ROLL 9
90 GO TO 80
```

A specific screen window can be ROLLed by following the direction code with four parameters: the X and Y coordinates of the top left hand corner of the window (the same coordinate system as for PLOT and DRAW), then the window width in characters (not pixels - this is possible but it would cost memory or speed) then the window length in pixels. Width can be 1 - 32, length can be 1 - 176. (The window specification for attribute moves can only be obeyed to a precision of individual character squares, not pixels). Alternatively, you can use the WINDOW command to reduce the size of the current window; the simpler form of ROLL will move only the area of the current window.

ROLL can be very effective in games for achieving smooth movement of players or background - "Frogger" would be easy! Extremely interesting - if slightly nauseating - effects can be obtained by setting up several overlapping ROLL windows so that the screen is apparently twisted and shredded into complex patterns. The routines below, which shred their own listings (or anything else on the screen) create some interesting effects.

```
100 LIST: LIST: LIST
110 LET pixels=4
120 ROLL 5,pixels;0,175;32,88
130 ROLL 6,pixels;0,175;16,176
140 ROLL 8,pixels;0,87;32,88
150 ROLL 7,pixels;128,175;16,176
160 GO TO 120
```

Try using pixels = 1 or other values, or change line 100 to:

```
100 KEYWORDS 0: PRINT STRING$(704," END PROC "):  KEYWORDS 1
```

(Type the "END PROC" as Graphic 3.) Another example:

```
200 FOR N = 1 TO 7: LIST: NEXT N
210 FOR L = 1 TO 175: ROLL 5;0,175;32,L: NEXT L
```

SAVE <line number TO line number;><drive number;>name
SAVE DATA <drive number;>name

Also discussed here: VERIFY (same syntax)

See also: DEFAULT (SAVE/LOAD device)

Beta Basic allows part of a program, or a program's variables to be SAVEd or VERIFIYed. A "slicer" selects which lines are to be saved, or DATA shows that only the variables are to be saved. If the drive number is omitted, the SAVE will be to tape, unless DEFAULT has been used to select another output device. If a drive number is given, the SAVE will be to the specified Micro-drive, unless DEFAULT has been used to select the Net or RS232 link as the default output device. If no slicer is given, the whole program is saved, along with its variables. Some examples:

```
SAVE 10 TO 200;"fragment"       SAVE lines 10 to 200 as "fragment"

VERIFY 10 TO 200;"fragment"     VERIFY that the program "fragment"
                                matches lines 10 to 200 of the
                                current program.

SAVE 900 TO,"box"               SAVE line 900 onwards as "box".

SAVE DATA "vars3"               SAVE the variables as "vars3".

VERIFY DATA "vars3"             VERIFY that the "program" "vars3"
                                matches the variables in memory.

SAVE 20 TO 70;2;"bit"           SAVE lines 20 to 70 to Microdrive
                                2 as "bit".
```

When you come to re-load part of a program, or some variables, remember that LOAD will erase your current program, including line zero. (Even the saved variables are considered to be a "program" with no lines!) You will usually want to use MERGE instead.

One of the main uses of the "SAVE a slicer" feature is to allow you to save procedures separately, so that you can build up a library of program modules for later use. LIST PROC gives a method for finding the first and last line numbers of a procedure. You may wish to move the procedure to a high line number before saving it - then you can later use MERGE to make it part of a program, followed by RENUM to move it and allow more procedures to be safely MERGEd at high line numbers.

SCROLL <direction><,pixels><;X,Y; width, length>

Key: S

See also: ROLL

SCROLL has a very similar syntax to ROLL, which should be read about first. One difference is that SCROLL can be used alone - in which case it simply moves the whole screen up by one character line (just like the ZX81). If SCROLL is followed by 5, 6, 7 or 8 the whole of the current WINDOW (normally the whole screen) will be moved by one pixel in the direction indicated by the arrow over the key. Anything pushed off the edge of the screen is destroyed; the new screen which is being SCROLLed in at the other side is blank. A given area of the screen can be SCROLLed by following the direction code with the X and Y coordinates of the top left hand corner of the desired window (in the same format as for PLOT and DRAW) and the width of the window (in character positions) and its length (in pixels). Both SCROLL and ROLL are very helpful in creating games and interesting graphic displays. Try the examples given for ROLL with SCROLL instead and contrast the results. Here is a program that takes a string and SCROLLS it across the screen:

```
100 LET A$="A NICE LONG STRING..."
110 FOR C = 1 TO LEN A$
120 PRINT AT 10,31; INK 7; A$(C)
130 FOR P = 1 TO 8: SCROLL 5; 0,95; 32,8: NEXT P
140 NEXT C
150 FOR P = 1 TO 255: SCROLL 5; 0,95; 32,8: NEXT P
```

The string is printed one character at a time to the same screen location in white INK (or INK the same colour as the paper). This makes each letter appear smoothly as it is SCROLLed out of the position which has "invisible INK" attributes. The inner FOR-NEXT loop moves the characters left by one position before the next one is printed, using a window covering one row of characters only. Line 150 moves the string off-screen once it has all been printed; alternatively, you could add 32 trailing blanks to the string. (Incidentally, if you find the mixing of PRINT AT and PLOT coordinate systems confusing, you could use Beta BASIC's enhanced PLOT and replace line 120 with:

```
120 PLOT 248,95; A$(C)
```

The same principle can be use to attractively present text (e.g. program instructions):

```
200 DATA "LONG AGO, IN A DISTANT GALAXY,"
210 DATA "LOTS MORE TEXT, LOTS MORE...."
300 FOR L = 1 TO 2: READ A$
310 PRINT AT 21,0; INK 7; A$
320 FOR P = 1 TO 8: SCROLL 7: NEXT P: NEXT L
330 FOR P = 1 TO 176: SCROLL 7: NEXT P
```

SORT
or **SORT INVERSE** string array or numeric array or string

Key: M

SORT arranges strings, numbers or letters in ascending or descending order. Its use with string arrays will be discussed first. Here is a program to generate an array of 100 random 10 letter strings. (You could speed it up by using Beta Basic's RNDM function rather than RND.)

```
100 DIM A$(100,10)
110 FOR S = 1 TO 100: FOR L = 1 TO 10
120 LET A$(S,L) = CHR$(RND * 25 + 65)
130 NEXT L: NEXT S: GO TO 200
140 SORT A$
200 FOR S = 1 TO 100: PRINT A$(S): NEXT S
```

As soon as the array has been created - which will take a little while - it is printed for you. Now GO TO 140 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 one fifth 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 0.7 sec, and 400 about 3 seconds.

Strings are sorted according to their CODEs - the normal sort has "space" before "A", which is before "a" - see the list in Appendix A of the Sinclair manual. If line 140 is changed to SORT INVERSE A$ this order will be reversed - try it! We can select any given block of strings to sort with a slicer:

```
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 take 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.)

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$(S,41) = CHR$ age
```

if we wish to place the age data into record S. 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$(S,41 TO 46) = STR$ balance
```

the information will be stored in the string, but it will be left-justified (the "9" of a £9 balance will be in position 41, as will the "1" of a £100 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$(S,41 TO 46) = USING$("000.00",balance)

(See USING$ and USING for a complete description of this function, and CHAR$ for information on string-coded integers.) 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 - eg 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 " BFdegglors" 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$(position) = CHR$(number)
```

It also works on conventional numeric arrays of one or two dimensions, using the same syntax as for string arrays. A two dimensional numeric array can be considered as a table in which the first dimension is the rows and the second is the columns.

```
SORT B(1 TO 20)(2)
```

will sort the first 20 rows of the array B on the basis of the numbers in the second column. (Entire rows will be moved around.) Note that we always need to use at least one slicer with a numeric SORT to distinguish array B() from simple variable B, even if the slicer is empty, as in:

```
SORT B()
```

The second slicer, if used, must always have a length of one - unlike the case with strings. Sorting of numbers is about four times slower than with strings, because SORT has to check for two different number formats (see page 170 of the original Spectrum manual) and for the possibility of mixed positive and negative numbers. Since this is not an alphabetic sort but a numerical one, larger numbers come first unless SORT INVERSE is used.

SPLIT - not a keyword. You actually enter "< >"

Key: symbol-shift W (not Graphics mode)

If a line that you have edited or have just written is entered with "< >" as the first non-space character in any statement, only the part of the line before the "< >" will be put into the listing. The rest will remain in the edit line at the bottom of the screen. The "< >" will be removed and replaced be a copy of the original line number. The cursor will be just to the right of the line number, ready for you to alter it before pressing enter (unless you wish 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. You could move this part of the line elsewhere in the program by changing the line number, then perhaps add it to an existing line using JOIN.

TAB (with LPRINT) - see Appendix E.

TRACE line number
or **TRACE**: statement: statement:...

Key: T

This is a facility which allows debugging of Basic programs with printing of current line, statement and selected variables, and reduced speed or single stepping. There are two forms of TRACE. The first form causes the line specified after TRACE to be GOSUBed immediately before every statement is executed. Any following statements in the same line have no special significance. The second form causes the following statements in the line to be GOSUBed before every statement is executed.

The TRACE subroutine (of either form) has available to it the special variables (not keywords) LINO and STAT, which are the line number and statement number of the part of the program that is about to be executed. TRACE is turned off during the execution of the trace subroutine, but is re-enabled by the RETURN which terminates it. The contents of the subroutine are up to you - a simple example using the first form would be:

```
9000 PRINT INVERSE 1;lino;":";stat: RETURN
```

Add this routine to a program, and insert the statement:

```
TRACE 9000
```

into the program at the point where you wish to start debugging. Using the second form, we can discard line 9000 and instead add to the program at the desired point:

```
TRACE: PRINT INVERSE 1;lino;":";stat: RETURN
```

If you want to turn off TRACE at some point in the program, insert:

TRACE 0

Use of the routine above will result in statements being listed as they are executed, in the format used by the "Reports". INVERSE is used to distinguish this list from any output the program proper may have. In order to see the text of the line being executed, include in the routine:

```
        LIST lino TO lino
or      LIST lino-1 TO lino
```

(The latter version will avoid inverse current line cursors everywhere, provided you have used line numbers separated by more than 1.) RUN, CLEAR or TRACE 0 turns the facility off. If you wish to add single stepping or slowing, use PAUSE or other means of your choice. Variables could also be printed - if you do this it is best to declare all the variables early in the program, or use the DEFAULT (variable value) statement, or you may get a "Variable not found" report. To avoid confusion of TRACE's output with that of the program, PRINT AT may be used, but it will be necessary to save the current print position or the printing of the main program will still be interfered with. The routine below should be activated by inserting TRACE 9000 into your program:

```
9000 LET pos=DPEEK(23688)
9010 PRINT AT 0,0;lino;":";stat;" ","A$ = ";A$;"  "
9020 DPOKE 23688,pos: RETURN
```

The program location and the variable A$ are printed after every statement, at the top of the screen. (Trailing spaces are to make sure previous values are over-printed.) The subroutine saves the relevant system variables so that the print position is only temporarily altered by the PRINT AT statement.

Your favourite TRACE routine could conveniently be assigned to a user-defined key (see DEF KEY) and saved with Beta Basic. Then you can simply type a line number and use the DEF KEY.

UNTIL condition

Key: K

Allows conditional execution of DO and LOOP - see these commands for more details.

USING

Key: U

As in: PRINT USING format string; number

Also discussed here: USING$ function

Both USING and USING$ allow specification of the format of numbers which are to be printed. USING can only be used as a modifier of PRINTed or LPRINTed numbers, while USING$ is a function. All applications of USING can be replaced by USING$ if desired. USING$ returns the string that PRINT USING would print; this allows formatting of numbers for use with LET, PRINT and any other command that can be applied to a string. With either USING or USING$, 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:

```
100 FOR n=1 TO 20: LET x=RND*100
110 PRINT x,USING "###.##";x: NEXT n
```

Note how much neater the formatted numbers are. By comparing the two columns, you will also 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
"000.00"        012.35
"00"            12
"£00.00"        £12.35
"0.00"          %..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 demonstrates an output with a "%" sign which indicates overflow of the specified format. (Note: USING does not at present work with scientific notation.) Trailing spaces in the format string are ignored. The function USING$ is similar to USING but instead of:

```
PRINT USING a$;number
```

we would use:

```
PRINT USING$(a$,number) or: LET b$=USING$(a$,number)
```

VERIFY <line number TO line number;><drive number;>name
or **VERIFY DATA** <drive number;>name

VERIFY will work with a "slicer" of line numbers, or, if DATA is used, the variables. The syntax for use with the Microdrive is also considerably simplified. See SAVE and DEFAULT (LOAD/SAVE device) in this manual for details.

WHILE condition

Key: J

WHILE allows conditional execution of DO and LOOP -- see those commands for details.

WINDOW window number <,x,y,width,length>

Key: 5.

See also: CLS, CSIZE

The WINDOW command allows you to set up separate areas of the screen which you can print and list to. Each area has its own print position, OVER, BRIGHT and FLASH status, INK, PAPER and CSIZE. You can use any number from 1 to 127 as a window number. These numbers have no special significance, but simply serve as a way of referring to a particular window. The details of any windows you define are stored above RAMTOP (which is lowered as needed) and so are safe from NEW, and can be saved with Beta Basic's CODE. Window 0 is permanently defined to cover the entire screen, and this window is the "current window" when Beta Basic is first loaded. This means that listings and printed characters act normally. To set up another window, you could use something like this:

```
WINDOW 1,0,175,128,176
```

Window 1 will have its top left-hand corner at point 0,175. It will be 128 pixels wide (half the screen) and 176 pixels long. The character size will be 8*8 pixels (normal) and the INK and PAPER and other attributes will be the current values for window 0 (you will see how to change these in a moment). Defining a window like this does not cause it to be the current window. To do that, enter WINDOW with just a window number, as in:

```
WINDOW 1
```

The specified window will be made the current window, provided you have already defined that window, as we did above. If you have not, you will get an "Invalid I/O device" report. The only window which is already defined is window 0. At the moment, with window 1 as the current window, your listing and printing will be restricted to the left half of the screen. To try different window shapes and sizes, you can enter a new definition for window l, or define other windows. In either case, you must use a WINDOW (number) command to see any effect, even if the number you select is the same as the current window number. Changes in the window definition only become effective when a WINDOW (number) command is used.

When a window is de-selected (i.e. when another window is selected) the CSIZE, INK, PAPER and other attributes, and the print position that were in effect are stored above RAMTOP. These values will be restored if the window is selected again. This means that you can switch printed output back and forth between windows that have different CSIZE, INK and PAPER. Enter the following and see how it changes the listing and the action of PRINT (print some long strings):

```
CSIZE 4,8: INK 7: PAPER 1
```

Now change to another window - WINDOW 0 will do - and the CSIZE, INK and PAPER will change. However, as soon as you re-select the previous window, the stored characteristics will be put back. The example below defines two windows and prints text to them alternately:

```
10 WINDOW 1,0,175,128,176
20 WINDOW 2,128,151,128,80
30 WINDOW 1: INK 1: PAPER 6
40 WINDOW 2: INK 7: PAPER 1: CSIZE 4,8
50 WINDOW 0
60 PRINT WINDOW 1;"one "; WINDOW 2;"two ";: GO TO 60
```

To clear a specific window, use CLS followed by the window number e.g. CLS 1. CLS on its own will clear the current window. Since RUN will only clear the current window, you may wish to use CLS 0 near the start of a program in order to clear the whole screen,

perhaps before using it for graphics. To erase all the window definitions, use WINDOW ERASE.

XOS, XRG, YOS and YRG

See also: [Appendix D]

These four words are not keywords - they are a special kind of variable which allows changing the scale and the origin used by the PLOT, DRAW, DRAW TO, CIRCLE, GET and FILL commands. XOS stands for X-axis off-set (of the origin), YOS is the same for the Y-axis, and XRG and YRG stand for X-axis range and Y-axis range. One way in which these variables are special is that they are set to particular values by CLEAR and RUN, rather than erased. CLEAR, then PRINT XOS (or xos) and you get "0", not "Variable not found". The two off-sets have the value zero (i.e. nothing is added to the normal origin) unless you use LET to set a different value. RUN or CLEAR will set them back to zero. CLS will reset the current graphics position to the origin (0,0) according to the current values of XOS and YOS. Changing the location of the origin is more convenient than altering several PLOT statements - for example:

```
LET XOS = 128,YOS = 88
```

will move the origin to the centre of the screen and allow you to PLOT any X values between -128 and 127, and any Y values between -88 and 87.

XRG normally has the value 256 (you can PLOT to 256 different places on the X-axis) and YRG is normally 176. Changing XRG and YRG alters the scale to which PLOT and DRAW work:

```
 10 GOSUB 100: REM normal
 20 LET XRG = 128: GOSUB 100
 30 LET YRG = 88: GOSUB 100
 40 LET XRG = 256: GOSUB 100: STOP

100 CLS: PLOT 0,0: DRAW 50,0: DRAW 0,50
110 DRAW -50,0: DRAW 0,-50: PAUSE 100: RETURN
```

The square drawn by the subroutine is first normal, then distorted along the X-axis, then along both X and Y-axes, and finally along just the Y-axis as the DRAW statements are scaled according to the specified ranges. The following example shows a sine wave being drawn using both off-set and range changes. (Use the Beta Basic function SINE for greater speed.)

```
100 LET XRG = 2*PI: REM 360 degrees
110 LET YRG = 2.2: REM sine varies between -1 and 1
120 LET YOS = 1.1: REM off-set origin half way up screen
130 FOR n = 0 TO 2*PI STEP 2*PI/256
140 PLOT n, SIN n: NEXT n
```

Note that the off-set value is also scaled according to the given range.

The end-point to which a curve is drawn will be accurately placed on the altered coordinate system created by the use of these special variables; however, the curve itself will still be part of a circle - it is not distorted to match the new coordinate system. Similarly, although the centre of a CIRCLE will be accurately placed, the radius is not scaled by either XRG or YRG; it is not possible to produce distorted CIRCLEs by changing XRG or YRG.

You may want to use the special graphics variables in order to make normal values in graphics commands produce output on only part of the screen. Combined with a WINDOW to limit your listings to another part of the screen, you can create a QL-style split screen.

FUNCTIONS

GENERAL

More than 20 new functions have been added to Beta BASIC using function definitions in line 0 (which is not usually visible in the listing) to point to machine code higher in memory which does the actual work. The functions exist in the program as normal user-defined functions, but they act like keywords in a listing; for example, if a line contains FN S$, it will appear as STRING$, and the cursor will pass it with a single key-press. User-defined functions which are not part of Beta Basic act normally. It may happen that you have some programs that already use some of the same user-defined functions that Beta Basic does. If so, you will have to change the function letters to avoid a conflict. This is easily done with ALTER (program references).

The names of Beta Basic's functions can be entered either in full (provided you are in KEYWORDS mode 3 or 4) or by entering FN, a letter, and "(" or "$". ("FN" can be obtained as Graphic-Y, or by entering F, then N.)

The new functions will not work (in fact, they will crash) if an attempt is made to use them when the main (machine code) part of Beta BASIC is not resident in memory. On the other hand, if line 0 is not present, you will simply get "FN without DEF" if you try to use them. In this case the rest of Beta BASIC will still work, but you will have lost the ability to use the new functions. SAVEing a program will also save line 0; therefore, if you LOAD a program written under Beta Basic, line 0 will be present. However, LOADing a program that was not written under Beta BASIC will erase line 0 - to avoid this, use NEW to get rid of any program currently in the computer (this will leave line 0 intact) and then MERGE the new program (retaining line 0). It is possible to remove line 0 from a program using DELETE 0 TO 0.

Below is a summary of the functions as they are displayed and as they exist in the program. They are given in alphabetical order of the displayed form. In the following pages, every function is described in detail. The two possible methods of entry are shown at the start of each description.

DISPLAYED AS:	EXISTS AS:	DISPLAYED AS:	EXISTS AS:
AND	FN A(	MEM	FN M(
BIN$	FN B$	MEMORY$	FN M$
CHAR$	FN C$	MOD	FN V(
LOSE	FN C(	NUMBER	FN N(
DEC	FN D(	OR	FN O(
DPEEK	FN P(	RNDM	FN R(
EOF	FN E(	SCRN$	FN K$
FILLED	FN F(	SHIFT$	FN Z$
HEX$	FN H$	SINE	FN S(
INARRAY	FN U(	STRING$	FN S$
INSTRING	FN I(	TIME$	FN T$
ITEM	FN T(	USING$	FN U$
LENGTH	FN L(	XOR	FN X(

AND (number, number)
FN A (number, number)

This function is spelled the same way as the normal keyword, but can be distinguished from it in a program listing by the different syntax used. It gives a bit-by-bit AND of two numbers, which must be between 0 and 65535. Only if a particular bit is a "1" in both the first number AND the second number will the same bit of the result be "1". A "0" in either number will give a "0" at that position in the result. The new function BIN$ is a great help in understanding what is going on.

```
          BIN$(254)  = "11111110"
          BIN$(120)  = "01111000"
BIN$(AND(254,120))  = "01111000"
```

You can use AND to "mask off" unwanted bits. For example:

```
PRINT AND(BIN 00000111, ATTR(line, column))
```

will give the INK colour for position "line, column" by masking off the other bits. (We could have used "7" instead of "BIN 00000111".) The example following will print "Bang!" if "F" is pressed, whatever other keys are being pressed at the same time. (See page 160 of the original Spectrum manual for information on the keyboard considered as a series of ports.)

```
10 IF AND(BIN 00001000, IN 65022)=0 THEN PRINT "Bang! ";
20 GO TO 10
```

BIN$ (number)
FN B$ (number)

Gives the binary equivalent of "number" as an eight character string (if "number" is less than 256) or as a sixteen character string (if "number" is between 256 and 65535).

This function is useful in understanding machine code and the operation of the bit-by-bit functions AND, OR and XOR.

It can also be very useful when examining the ROM character generator, user-defined graphics area, the attribute file, the system variables or the keyboard. The latter is demonstrated below (see page 160 in the original Spectrum manual).

```
10 PRINT AT 10,10; BIN$(IN 65022): GOTO 10
```

If you would like something other than "1" and "0" in the string, POKE 62865 or 62869 with the desired character. This can be useful when decoding character patterns (such as those starting at address 15616) - you could POKE "X" and " ", for example.

CHAR$ (number)
FN C$ (number)

See also: NUMBER(string) function

This function converts integers (whole numbers between 0 and 65535) into two-character strings, allowing storage of much numeric data with good memory economy. The equivalent in Basic is:

```
LET A = INT(number/256): LET B = number - A * 256
LET C$ = CHR$ A + CHR$ B
```

The string result will often give report K, "Invalid colour", if an attempt is made to print it, since it may contain print control codes. NUMBER(two-character string) would usually be used to translate the character-coded integers back into numbers. Since only two bytes are used per number, versus five in the normal format, it is well worth considering this function if you have a lot of numeric data to crunch. The data need not be an integer in the first place - you can multiply up something like 87.643 to give 8764.3. CHAR$ would use the 8764 part of this to produce a two-character string; subsequent use of NUMBER and division by 100 would give 87.64, which is acceptable accuracy for many purposes. The following example shows implementation of what is essentially an integer array:

```
100 DIM A$(500,2)
110 FOR E = 1 TO 500: LET A$(E) = CHAR$(E * 10): NEXT E
120 PRINT "Array created - press any key to print it"
130 PAUSE 0
140 FOR E = 1 TO 500: PRINT E, NUMBER(A$(E)): NEXT E
```

Note that this array uses just 1K bytes, versus 2.5K with standard array of the same number of elements. SORT will work correctly on such arrays.

COSE (number)
FN C (number)

Gives the cosine of "number", to a lesser degree of accuracy than COS (although still usually accurate to 4 decimal places), but about six times faster.

DEC (string)
FN D (string)

See also: HEX$(number) function

This function gives the decimal equivalent of a 1-4 character string which represents a valid hexadecimal number. The case of any letters is not significant.

```
DEC("FF")     = 255
DEC( "10")    = 16
DEC("4000")   = 16384
DEC("e")      = 14
```

To POKE memory, using hexadecimal input, you could use:

```
INPUT A$: POKE address, DEC(A$)
```

Use with the null string or a string of more than four characters, or using a string containing characters other than 0-9, A-F or a-f gives an "Invalid argument" report.

DPEEK (address)
FN P (address)

See also: DPOKE command

DPEEK is a double PEEK of the specified address and the next address up. The Basic equivalent is:

```
LET address=PEEK (address)+256* PEEK (address+1)
```

Note that the least significant byte is assumed to come first, which is the usual practice for system variables and machine code. For example:

```
100 LET nxt = DPEEK(23637): POKE nxt+5, 65
110 REM xxxxxx
```

This reads the address of line 110 from the system variable "NXTLIN". The first character after the REM is then altered to an "A" by POKE. (POKE nxt + 5,"Ha!" will also work.) The "+5" is needed to skip the line number and line pointer bytes, and the REM keyword.

DPOKE allows double POKE in the same way that DPEEK allows double PEEK.

EOF (stream number)
FN E (stream number)

EOF stands for End Of File. This function tells you if the last item in a Microdrive file has been read or not.

The stream specified must have been previously OPENed to a Microdrive file, or you will get an "Invalid stream" report. The file must already exist (it must be open for reading) or you will get a "Reading a "write" file" report. The function returns 1 if the last item has been read, or 0 if not. This makes it easy to detect end-of-file and avoid trying to read more items than there are in the file, which would cause an error.

Let's assume you have a Microdrive file called "data". An elegant way to use the function would be:

```
10 OPEN #5,"m",1,"data"
20  DO UNTIL EOF(5)=1
30    INPUT #5;a$: PRINT a$
40 LOOP
50 STOP
```

Line 20 could also be entered as:

```
 20 DO UNTIL EOF(5)
```

If DO-LOOPs confuse you, you could use instead:

```
10 OPEN #5,"m",1,"data"
30 INPUT #5;a$: PRINT a$
40 IF EOF(5)=0 THEN GO TO 30
50 STOP
```

FILLED ()
FN F()

See also: FILL command

Gives the number of pixels filled by the last FILL command. For example:

```
10 PLOT 0,0: DRAW 9,0: DRAW 0,9
20 DRAW -9,0: DRAW 0,-9
30 FILL 5,5
40 PRINT FILLED()
```

The side length of the box is 10 pixels. (Remember, if we had used 1 rather than 9 in the DRAW statements, the side length would have been 2 pixels.) This is because although lines may theoretically be infinitely thin, real lines on a computer are one pixel wide - you can imagine the theoretical lines running down the middle of the real pixel lines. This gives a half-pixel border on either side, so the external dimensions of our box in the example are increased by two half-pixels in both width and length. Thus it has a side length of 10 rather than 9 pixels. Internally, width and length are reduced to 8 pixels, so FILLED() gave us 64. If we use FILL PAPER;5,5 to remove the box, PRINT FILLED( ) will give 100. (The difference between 64 and 100 is the 36 pixels used to form the perimeter.)

HEX$(number)
FN H$(number)

See also: DEC (string) function

The numeric argument is converted to a hexadecimal string. This will be two characters long if the number was between -255 and +255, and four characters long if the absolute value of the number was greater than this. Values over 65535 give an "Integer out of range" report.

```
HEX$(32)      = "20"
HEX$(255)     = "FF"
HEX$(512)     = "0200"
HEX$(-64)     = "C0"
HEX$(-1024)   = "FC00"
```

The ability to deal with negative numbers should prove useful to machine code users calculating backwards relative jumps.

To display memory in hexadecimal you could use:

```
100 INPUT "Start address? "; address
110 PRINT HEX$(address);" ";HEX$(PEEK address)
120 LET address = address+1: GOTO 110
```

To specify the start address in hex, change line 100 to:

```
100 INPUT "Start address? ";A$: LET address = DEC(A$)
```

INARRAY (string array(start element<,slicer>),target string)
FN U (string array(start element<,slicer>),target string)

See also: INSTRING function

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 INSTRING; we recommend you read about INSTRING first. The examplebelow 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 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,l TO 5).) It is 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 INSTRING. 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.

```
10 LET namelen=20,townlen=15
20 DIM d$(10,namelen+townlen)
30 FOR n=1 TO 10
40    INPUT "name? ";n$
50    INPUT "town? ";t$
60    LET d$(n,1 TO namelen)=n$
70    LET d$(n,namelen+1 TO )=t$
80 NEXT n
90 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.

```
100  INPUT "name? ";n$
110  INPUT "town? ";t$
120  LET s$=n$+STRING$(namelen-LEN n$,"#")+t$
130  LET loc=INARRAY(d$(1),s$)
140  IF loc=0 THEN
        PRINT "not found"
     ELSE
        PRINT loc;" ";d$(loc)
150  GO TO 100
```

Note: INARRAY will not work with arrays of more than two dimensions.

INSTRING(start, stringl, string2)
FN I(start, stringl, string2)

See also: MEMORY$ function, INARRAY function

INSTRING searches stringl for string2, beginning at character "start" in stringl. If the string is found, the result is the position in stringl of the first character of string2; otherwise the result is zero.

Stringl may be any length, but string2 must be less than 256 characters long (or you will get an "Invalid argument" report). If "start" is zero, you will get a "Subscript wrong" report. Zero is returned if string2 is longer than stringl or "start" is greater than LEN stringl, or if either of the two strings has LEN zero.

It is possible to replace some of the characters in the string being searched for with "#" (hash) which means "don't care". For example:

```
PRINT INSTRING(1,A$,"SM#TH")
```

will find the position of "SMITH", "SMYTH", "SMATH", etc. anywhere in A$. The only time # (literally) is looked for is when it is the first character of the string being looked for. The ability to specify the start position for the search is useful when you expect to find more than one occurrence of the target string. The following example will search A$ for every use of "TEST".

```
100 DIM A$(1000)
110 FOR n = 1 TO RND * 10 + 3
120 LET pos = RND * 995
130 LET A$(pos TO pos + 3) = "TEST"
140 NEXT n
150 PRINT "TESTs hidden in A$ - press any key
     to find them"
160 PAUSE 0
170 LET loc = 1
180 LET loc = INSTRING(loc, A$, "TEST")
190 IF loc <> 0 THEN PRINT "Target found at
     position ";loc: LET loc = loc + 1: GO TO 180
200 PRINT "That's the lot!"
```

The string A$ is first searched from position 1 (loc=1) and subsequently from just past each "TEST" that it finds. When INSTRING returns "0", all occurrences have been found.

Note: line 190 could read "190 IF loc THEN PRINT..." etc. since "0" is equivalent to "not true". If you feel the above example is too easy (after all, the string is mostly blanks) change line 100 to: LET A$ = STRING$(250,"TESS") which will require each "TESS" to be checked as far as the fourth character before it can be distinguished from "TEST".

The whole of memory or a particular part of it can be searched by INSTRING using the function MEMORY$.

INSTRING can be used to check an input string in educational or games programs. For example, suppose the Spectrum has asked some question to which the required answer, stored in C$, is "NAPOLEON". People who input "NAPOLEON " (note the space) or "NAPOLEON BONAPARTE" and are told they are wrong tend to become annoyed - but of course this often happens when the input string and C$ are compared in any simple way. You can solve the problem in most cases with something like the following, which prints "Correct" if the correct answer is contained anywhere within the input string:

```
INPUT A$: IF INSTRING(1,A$,C$)<>0 THEN PRINT "Correct!"
```

(You might also like to force A$ to capitals using the SHIFT$ function; insert as the second statement: LET A$=SHIFT$(1,A$).)

Yet another application for the versatile INSTRING is to implement packing of multiple variable-length strings or records into one long string. One approach would be to reserve all characters in a particular range (e.g. CHR$ 1 - CHR$ 31) to be "marker" characters. CHR$ 1 could mark the start of the first substring in a long string, CHR$ 2 the second, etc. Then:

```
PRINT A$(INSTRING(1,A$,CHR$ n),+1 TO INSTRING(1,A$,CHR$(n+1))-1)
```

will print the "n"th substring from A$. Many alternative schemes are possible, but all have the advantage that space is saved if the strings are irregular in length.

ITEM ()
FN T ()

See also: Section on PROCEDURES

This function gives information concerning the next item to be READ. Its main use is with procedures, but it can also be used with conventional READ and DATA techniques. The function has values as follows:

0 if all the items in the current DATA statement have been READ. The "current DATA statement" may be in fact a list of parameters following a procedure call.

1 if the next item is a string.

2 if the next item is numeric.

When used with procedures, ITEM( ) can tell you the nature of the first item in the list, but in other cases ITEM() will return zero until at least one item has been READ from the current DATA statement. The example below therefore tests ITEM( ) after READ is used:

```
100 DO
110  READ x
     PRINT x
120 LOOP UNTIL ITEM()=0
130 DATA 1,2,3,4,5,6
```

Lines 100 to 120 can be used to read a DATA statement of any length

LENGTH (n,"array name")
FN L (n,"array name")

LENGTH tells you the size of an array, which is particularly useful in Beta Basic since arrays can change their dimensions without data loss. LENGTH can also be used to find the location of an array or string in memory. The array name is enclosed in quotes to allow both numeric and string names to be used. If n=1, the first dimension is returned. If n=2, the second dimension is returned, or 1 is returned if the array is actually only one dimensional. Arrays of more than two dimensions are not dealt with.

Only the first two characters of the array name are significant; for example, a$, b$, c(, d(), a$qwert are all acceptable. If a simple string name is used instead of a string array name, the function will treat it as though it was an array with LEN (string) elements, each one character long. Here are some examples:

```
10 DIM a$(10,20)
20 PRINT LENGTH(1,"a$"): REM prints 10
30 PRINT LENGTH(2,"a$"): REM prints 20
40 DIM b(5)
50 PRINT LENGTH(1,"b("): REM prints 5
60 PRINT LENGTH(2,"b("): REM prints 1
```

An extra feature is provided by LENGTH(0,array name), which returns the address of the first element of the array (or string). This could be of use to machine code users; it would be possible for a machine code routine to work on the array (or string), or machine code in the array could be executed. For Basic users, there are a number of applications made possible by Beta Basic's string POKE and MEMORY$ features. To duplicate an array a$ as b$ (which could be one-dimensional if you wish to change the arrangement of the data):

```
20 LET e=LENGTH (1,"a$")
30 LET l=LENGTH (2,"a$")
40 DIM b$(e,l): REM or DIM b$(e*l)
50 LET source=LENGTH (0,"a$")
60 POKE LENGTH (0,"b$"), MEMORY$()(source TO source+e*l-1)
```

(A numeric array would need e*1*5 bytes to be moved, since each number is stored as 5 bytes.) The above differs from Beta Basic's array COPY command in many ways - it is much less versatile. However, the routine makes a good basis for a simpler, faster array duplication command. Add the lines:

```
10 DEF PROC dup REF a$, REF b$
70 END PROC
```

Now a command such as: dup r$, t$ will duplicate r$ as t$.

Experienced users: If you already have an array which you want to use with Beta Basic but which is too big to load, you can split it up by first saving it in sections as CODE, then using LENGTH to reload the CODE into an array. When Beta Basic is not present you can find the start of an array or string by making sure it is the first variable in the variables area. (Assign it first, or use only one variable.) Then PEEK 23627+256* PEEK 23628+d (where d=3 for a string, 6 for a one-dimensional array or 8 for a two-dimensional array) gives the start of the first data byte of the variable. The characters, strings or rows of numbers follow one another in memory, so you can work out the start address and length of a section quite easily. Characters take 1 byte each, numbers take 5.

Note: If LET creates or alters string variables the location of an array or string may change, making previous values returned by LENGTH (0,"name") incorrect.

MEM()
FN M()

MEM() returns the number of bytes of free memory available. Nothing should go in the brackets. Try:

```
PRINT MEMO(): DIM A$(100): PRINT MEM()
```

This is a simple function which consists mostly of a call to ROM - it can be duplicated in the absence of Beta BASIC by PRINT 65535 - USR 7962.

MEMORY$()
FN M$()

See also: POKE strings

Returns the whole of memory as a string! Actually, the first byte in the computer, location 0, is not included, so that CODE MEMORY$()(1) is the same as PEEK 1. The last 3 bytes in memory are also excluded (for technical reasons) so the function has LEN 65532. You will obviously run out of space if you try:

```
LET a$=MEMORY$()
```

but you could use, for example:

```
LET a$=MEMORY$()(16384 TO 22527)
```

In combination with Beta Basic's ability to POKE strings, this function gives the Basic programmer the power to move large areas of memory about very rapidly. For a fuller description of this aspect, see POKE in this manual.

Another use of MEMORY$ is to allow quick searches of memory using INSTRING. Although a specified section of memory can easily be searched by using the "slicer" notation (e.g MEMORY$()(23759 TO)) INSTRING is so fast that it may often be simpler to the search whole of memory.

```
10 REM asdfg
20 PRINT INSTRING (1,MEMORY$(),"asdfg")
```

will find the location of "asdfg" in the REM statement. If you omit line 10, the "asdfg" will be found in line 20. If you LET a$="asdfg" and search for a$, you will find it in the variables area. You will always find at least one occurrence of the string somewhere in memory.

In place of "1" we could have used "DPEEK(23635)", which is the PROG system variable, in order to start the search at the start of the program rather than at address 1 in the ROM.

To find all occurrences of a string, you could use:

```
10 LET adr=1
20 LET adr=INSTRING(adr,MEMORY$(),a$)
30 IF adr<>0 THEN PRINT adr: LET adr=adr+1: GO TO 20
```

Since Beta Basic allows strings to be POKEd, searching for one string and replacing it with another is easy, provided that you know what you are doing. Often, you may wish to avoid the replacement string being longer than the first.

MOD(number, number)
FN V(number, number)

Gives the amount left over when the first number is divided by the second number. This is called taking the first number "modulo" the second number.

```
MOD(10,3)     = 1
MOD(66,16)    = 2
MOD(125,35.5)= 18.5
```

The example below prevents PLOT being off-screen.

```
10 FOR n = 0 TO 400
20   PLOT MOD(n,256), MOD(n,176)
30 NEXT n
```

NUMBER(string)
FN N(string)

See also: CHAR$ (number)

Converts a two-character string to an integer (a whole number) between 0 and 65535. The BASIC equivalent is:

```
LET number = 256 * CODE C$(1) + CODE C$(2)
```

You will get an "Invalid argument" report if the string is not two characters long. With the complementary function CHAR$, NUMBER makes "integer arrays" easy to implement (see CHAR$ for an example).

OR(number, number)
FNO(number, number)

This function is spelled the same way as the normal keyword, but can be distinguished from it in a program listing or during program entry by the different syntax used. It gives a bit-by- bit OR of two numbers, which must be between 0 and 65535. If a bit is a "1" in the first number OR the second number, then it will be "1" in the result. The bit must be "0" in both numbers for it to be "0" in the result.

RNDM (number)
FN R (number)

If "number" is 0, RNDM gives a random number between 0 and 1, like RND. It is about two and a half times faster, however. If "number" is non-zero, RNDM gives a random whole number between 0 and "number" inclusive. This also is about two and a half times faster than RND * "number" would be.

```
10 PLOT RNDM(255), RNDM(175)
20 GO TO 10
```

RANDOMIZE (number) will set RNDM to a particular place in its pseudo-random number sequence, just as it does for RND.

SCRN$ (row, column)
FN K$ (row, column)

Works similarly to the normal SCREEN$, except that it recognises user-defined graphics as well as normal characters. A bug that afflicts Spectrum Basic's SCREEN$ has also been avoided. First, enter KEYWORDS 0. Then try the example below, which makes the user-defined graphics into random patterns, and then reads some from the screen.

```
10 FOR a = USR "a" TO USR "u" + 7
20 POKE a, RND * 255: REM RNDM (255) is faster!
30 NEXT a
40 PRINT "some user-defined graphics"
50 LET a$ = ""
60 FOR c = 0 TO 31
70 LET a$ = a$ + SCRN$(0,c)
80 NEXT c
90 PRINT a$
```

The Spectrum's block graphics are not recognised. If you need to do this, program some of the user-defined graphics to look like the block graphics. Characters are only recognised when CSIZE is normal - i.e. the characters are 8*8 pixels.

SHIFT$ (number, string)
Z$ (number, string)

SHIFT$ is a general-purpose string manipulation function. Depending on the value of "number", it can change the case of letters, suppress control codes in various ways, or convert between keywords and their fully spelled-out form. Here is a brief summary:

number

1 Force all letters to upper case (capitals).
2 Force all letters to lower case.
2 Reverse the case of all letters.

4 Change all control codes except CHR$ 13 to "."
5 Change CHR$ 128-255 to CHR$ 0-127 (suppress keywords).
Change all control codes except CHR$ 13 to "."
6 Change CHR$ 128-255 to CHR$ 0-127 (suppress keywords).
Change all control codes to "."

7 Change keywords to fully spelled-out equivalent.
8 Change fully spelled-out keywords to compact form.
Ignore case, insist on a non-letter character after the end of a keyword.
9 Same as 8, but any character can follow a keyword
10 Same as 9. but keywords must be in capitals.
11 Same as 8. but keywords must be in capitals.

CASE CONVERSION SHIFT$ 1-3

Some examples:

SHIFT$(1,"Beta Basic 3.0") ="BETA BASIC 3.0"
SHIFT$(2,"Beta BASIC 3.0") ="beta basic 3.0"
SHIFT$(3,"Beta Basic 3.0") ="bETA bASIC 3.0"

A common use would be to convert input to a particular case before a comparison, for example:

```
100 INPUT i$: IF SHIFT$(1,i$)="Y" THEN GO TO 200
```

This can save you a whole series of comparisons such as IF i$="Y" OR i$="y". The function was actually first used to convert all our customer records to capitals before using SORT on them.

CONTROL CODE AND KEYWORD SUPPRESSION: SHIFT$ 4-6

This use of SHIFT$ is probably of most use to machine code users. If you are examining memory, you may often wish to print CHR$(PEEK address), but before long you get an "Invalid colour" report because you have tried to print something like CHR$ 17; CHR$ 200 which is PAPER 200. You may also have problems getting a neat format because characters over 127 may be printed as keywords. SHIFT$ lets you suppress these effects. In a few minutes you can scan the entire memory for data tables, messages and keyword lists, using a program such as:

```
100 FOR n=1 TO 65535 STEP 704
110 PRINT SHIFT$(6,MEMORY$()(n TO n+703))
120 PAUSE 0: CLS
130 NEXT n
```

You might prefer to use SHIFT$ 4 if you are looking at the area of memory containing a program - this will allow keywords to be printed, but remove most control codes.

CHANGING KEYWORDS TO SPELLED-OUT FORM: SHIFT$ 7

For example:

```
10 LET a$=" THEN NOT": REM the keywords
20 PRINT a$, LEN a$: REM LEN=2
30 LET t$=SHIFT$(7,a$)
40 PRINT t$, LEN t$: REM LEN=9
```

The main reason for providing this function is to ensure that users with incompatible printer driving software can still obtain a LLISTing of Beta Basic programs - providing they have a Microdrive. (Additionally, we have software that relocates the Tasman, Morex or Kempston printer drivers, and "patches" for use with the Euroelectronics and other EPROM-based interfaces. Send an S.A.E. and specify your interface if you wish a copy.)

The procedure below is a little complex, but it works:

```
9000 OPEN #5;"m";1;"temp"
9010 LIST #5; TO 8999
9020 CLOSE #5
9030 OPEN #5;"m";1;"temp"
9040 LET a$=""
9050 DO UNTIL EOF(5)
9060    INPUT #5; LINE t$
9070    LET a$=a$ + SHIFT$(7,t$)+CHR$ 13
9080 LOOP
9090 CLOSE #5
9100 ERASE 1;"temp"
9110 OPEN #5;"m";1;"listing"
9120 PRINT #5;a$
9130 CLOSE #5
```

This routine can be MERGEd with the program you want to list. First it lists the program to a Microdrive file, then it reads it back into a$, expanding all keywords (including Beta Basic's keywords) as it does so. This string is then printed to another file. If your problem was due to the printer refusing to print Beta Basic's keywords, there should now be no problem printing the file containing the program listing, because it does not contain any keywords as single characters. If your problem was due to a memory conflict between Beta Basic and your printer driving software, you can now get rid of Beta Basic and load the driver. Now print the Microdrive file:

```
10 OPEN #5;"m";1;"listing"
20 INPUT #5;LINE a$
30 LPRINT a$
40 GO TO 20
```

CHANGING SPELLED-OUT KEYWORDS TO TOKENS: SHIFT$ 8-11

Any sequence of characters that is recognised as a fully spelled-out keyword is changed into the compressed form (either a single character, or FN + letter + "(" or "$"). A keyword will not be recognised if there is a letter immediately preceding it. The other requirements for recognition vary according to the number used:

number

8 Keywords are recognised in upper and lower case. they must by followed by a character that is not a letter. (Beta Basic uses this routine if you are typing in your programs letter by letter.)

9 Keywords are recognised in upper and lower case. Any character can follow a keyword.

10 Keywords are only recognised in upper case (capitals). Any character can follow a keyword.

11 Keywords are only recognised in upper case. They must be followed by a character that is not a letter.

This function could be used as part of a Basic full screen editor, or it could be used to convert programs sent from a different make of computer (e.g. BBC) over the RS232 link into Spectrum tokens. Either use would involve KEYIN to actually enter the strings into a program.

SINE (number)
FN S (number)

Gives the sine of "number", to a lesser degree accuracy than SIN (although usually still accurate to 4 decimal places), but about six times faster.

STRING$(number, string)
FN S$ (number, string)

This function gives "number" repeats of "string":

```
STRING$(32,"-")          =     32 minus signs
STRING$(4,"AB")          =     "ABABABAB"
PRINT STRING$(704,"X")   =     a screenful of "X" PRINT
STRING$(3,"A"+CHR$ 13)   =     A
```

STRING$ is faster than a FOR-NEXT loop and shorter than entering the string explicitly, provided the string would be longer than about 14 characters. It can be useful in combination with Beta Basic's string POKE for filling an area of memory (such as the attributes file) with desired values.

TIME$()
FN T$()

See also: CLOCK

This function returns the current time, as held by Beta Basic's digital clock (see CLOCK). If you enter: PRINT TIME$( ) a few times, you will see that you get continually changing values. It is often a good idea to pass the result of TIME$ to a variable, to "freeze" it at one moment in time:

```
100 CLOCK 1
110 LET N$ = TIME$(): PRINT N$
120 PRINT "Hours = ";N$(1 TO 2);"Mins = ";N$(4 TO 5)
130 GO TO 110
```

See CLOCK for details on how to speed up the clock rate for use as a stopwatch.

USING$ (format string, number)
FN U$ (format string, number)

See also: USING

Returns the string equivalent of "number", formatted as specified by "format string". The number of leading or trailing digits can be specified, and rounding is done to the nearest displayed digit. The keyword USING on the "U" key provides an equivalent function which can only be applied to PRINT, whereas USING$ can be used with any command that allows the use of a string - eg LET. See USING for a fuller explanation of number formatting with USING/USING$.

XOR (number, number)
FN X (number, number)

Gives a bit-by-bit exclusive OR of two numbers, which must be between 0 and 65535. If a bit is a "1" in both numbers, or a "0" in both numbers, it becomes "0" in the result. If a bit

is a "1" in only one of the numbers, it will be "1" in the result. AND shows an example that you can modify for use with XOR.

APPENDIX A

THE CHARACTER SET

The following amendments to Appendix A of the Spectrum handbook are in force unless KEYWORDS 0 has been used to turn off the new keywords.

CODE	KEY	CHARACTER
128	8	KEYWORDS
129	1	DEF PROC
130	2	PROC
131	3	END PROC
132	4	RENUN
133	5	WINDOW
134	6	AUTO
135	7	DELETE
136	shift-7	REF
137	shift-6	JOIN
138	shift-5	EDIT
139	shift-4	KEYIN
140	shift-3	LOCAL
141	shift-2	DEFAULT
142	shift-1	DEF KEY
143	shift-8	CSIZE
144	A	ALTER
145	B	----
146	C	CLOCK
147	D	DO
148	E	ELSE
149	F	FILL
150	G	GET
151	H	----
152	I	EXIT IF
153	J	WHILE
154	K	UNTIL
155	L	LOOP
156	M	SORT
157	N	ON ERROR
158	O	ON
159	P	DPOKE
160	Q	POP
161	R	ROLL
162	S	SCROLL
163	T	TRACE
164	U	USING

APPENDIX B

REPORTS

Report G is used differently by Beta BASIC and Spectrum BASIC. The other reports that follow are new reports used in addition to those of Spectrum BASIC.

CODE	MEANING	SITUATIONS
G	No room for line Renumbering the program as specified would result in the new line numbers being mixed up with a block which is not being renumbered, or greater than 9999.	RENUM
S	Missing LOOP EXIT IF or a conditional DO (one followed by WHILE or UNTIL) tried to go to the end of a DO-LOOP and failed to find a LOOP statement.	DO, EXIT IF
T	LOOP without DO LOOP was used without a matching DO statement.	LOOP
U	No such line DELETE was used with a line number that does not exist in the program.	DELETE
V	No POP data An attempt was made to remove data from the GOSUB/DO-LOOP/PROC stack when the stack was empty; i.e. no GOSUBs, DO-LOOPS or PROCs were in operation.	POP
W	Missing DEF PROC A procedure name was used without there being a procedure definition of the same name in the program, or END PROC was used without DEF PROC.	procedure calls, END PROC LOCAL
X	No END PROC In trying to jump over a procedure definition, no END PROC could be found.	DEF PROC

APPENDIX C

ERROR CODES

The following list gives the value of the variable ERROR (produced by use of the ON ERROR command) for the various error states. The first section is for the standard error reports, the second is for Beta Basic's reports, and the third is for "Interface errors" and only applies if Interface 1 is connected. Reports "0" and "9" are not intercepted by ON ERROR.

ERROR VALUE	CODE	MESSAGE
0	0	OK (Not intercepted)
1	1	NEXT without FOR
2	2	Variable not found
3	3	Subscript wrong
4	4	Out of memory
5	5	Out of screen
6	6	Number too big
7	7	RETURN without GOSUB
8	8	End of file
9	9	STOP statement (not intercepted)
10	A	Invalid argument
11	B	Integer out of range
12	C	Nonsense in BASIC (see report 44)
13	D	BREAK - CONT repeats
14	E	Out of DATA
15	F	Invalid file name
16	G	No room for line
17	H	STOP in INPUT
18	I	FOR without NEXT
19	J	Invalid I/O device
20	K	Invalid colour
21	L	BREAK into program
22	M	RAMTOP no good
23	N	Statement lost
24	O	Invalid stream
25	P	FN without DEF
26	Q	Parameter error
27	R	Tape loading error

BETA BASIC REPORTS

ERROR VALUE	CODE	MESSAGE
28	S	Missing LOOP
29	T	LOOP without DO
30	U	No such line
31	V	No POP data
32	W	Missing DEF PROs
33	X	No END PROC

INTERFACE 1 REPORTS

These are given in the order in which they exist in the Interface 1 ROM. Two reports, "m Header mismatch error" and "u Hook code error", are not documented in the Interface 1 manual. Report "c Nonsense in BASIC" is a duplicate of the normal report, but caused by Interface operations. The lower case code letters are only produced when Beta Basic is present.

ERROR VALUE	CODE	MESSAGE
43	b	Program finished
44	c	Nonsense in BASIC
45	d	Invalid stream number
46	e	Invalid device expression
47	f	Invalid name
48	g	Invalid drive number
49	h	Invalid station number
50	i	Missing name
51	j	Missing station number
52	k	Missing drive number
53	l	Missing baud rate
54	m	Header mismatch error
55	n	Stream already open
56	o	Writing to a "read" file
57	p	Reading a "write" file
58	q	Drive "write" protected
59	r	Microdrive full
60	s	Microdrive not present
61	t	File not found
62	u	Hook code error
63	v	CODE error
64	w	MERGE error
65	x	Verification has failed
66	y	Wrong file type

APPENDIX D

SPECIAL VARIABLES

Beta Basic possesses a number of special variables, which do not behave quite the same way as ordinary ones. Their names and properties are given below.

GRAPHICS VARIABLES

These variables always exist - they are set to a particular value by RUN or CLEAR, rather than destroyed. Using LET to change their values will affect the operation of DRAW, DRAW TO. PLOT. CIRCLE. GET (from the screen) and FILL.

NAME	VALUE SET BY RUN/CLEAR	MEANING
XOS	0	X-AXIS OFFSET
XRG	256	X-AXIS RANGE
YOS	0	Y-AXIS OFFSET
YRG	176	Y-AXIS RANGE

TRACE and ON ERROR VARIABLES

These variables do not normally exist - they are created by ON ERROR if an error occurs while it is in use, or by TRACE before every statement is executed, if TRACE is operating.

NAME		**MEANING**
ERROR	-	Code value for last error (See Appendix C).
LINO	-	For TRACE, line number about to be executed. For ON ERROR, line number where error occurred.
STAT	-	For TRACE, statement number about to be executed. For ON ERROR, statement number where error occurred.

APPENDIX E

USE OF PRINTERS

Beta Basic programs will LLIST correctly to the ZX printer or to a serial printer connected to Interface 1's 't' channel. If you have an Interface 1 with a serial number less than 87316, Beta Basic provides a TAB facility on the 't' channel. The line length of output on this channel can be set by POKEing 57500 with the desired value. (The value as supplied is 80 characters.) If you have an Interface 1 with a serial number of 87316 or more, the Interface provides the TAB feature itself, and the line length is set by POKEing 23729.

In other cases, the many different types of printer interface and driving software can create compatibility problems. Beta Basic needs control of the 'p' channel if it is to successfully LLIST the new keywords. Before taking control (during initialisation) Beta Basic stores the address that it finds in the LPRINT channel, and later routes all printer output to this address after modifying it. You should therefore load and initialise your printer driving software before loading Beta Basic. Providing that there is no conflict of memory usage, and providing that the printer driving software does not subsequently force its own address back into the LPRINT channel, all should be well. If you have a problem, but own a Microdrive, you will be able to obtain a listing by a devious method using the function SHIFT$ 7 - see SHIFT$.

Alternatively, if your printer driving software occupies the same locations as Beta Basic, we may be able to help by supplying a relocator program to move the driver. These are available for the Tasman, Morex, or Kempston interfaces - just tell us where you bought Beta Basic, specify your interface and send an S.A.E. We will usually be able to write a relocator for any other interface if we are sent a tape of the driving software.

One common problem is caused by control codes. The routine to handle them in many printer driving programs involves temporarily changing the address held in the 'p' channel - this may cause Beta Basic to lose control, preventing the new keywords being printed out. Therefore, as supplied, Beta Basic handles most control codes itself - for example, TAB is turned into a string of spaces before being sent to the printer. However, this could prove inconvenient at times, so a 'switch' is provided. The address 60921 controls what happens: if it is 1, control codes are sent to the printer; if it is 0, most control codes are handled by Beta Basic. Do not POKE any other values!

You can read the address of the routine that controls LPRINT from the 'p' channel; it should be 64423 if Beta Basic is in control:

```
PRINT DPEEK(DPEEK(23631)+15)
```

The address to which printer output is sent after Beta Basic has modified it can be got by:

```
PRINT DPEEK(61081)
```

It is normally 2548, an address in ROM. It should contain the address of your printer driving software if you initialised your printer before loading Beta Basic. You can get Beta Basic to take over the 'p' channel as it does during initialisation by:

```
RANDOMIZE USR 58419
```

SYNTAX SUMMARY

Anything enclosed like <this> can be omitted. Exp=expression, ln=line number, prm=parameter, stat=statement, var=variable.

ALTER <attrs> TO attrs
ALTER reference TO reference
AUTO <start line><,step>
BREAK (improved) normal key(s)
CAT <drive>
CLEAR up to +/- 767 bytes
CLOCK number or string
CLOCK: stat: stat:...: RETURN
CLS <window number>
COPY a$<slicer> TO b$<position>
COPY a<slicer> TO b<position>
CSIZE width<,height>
DEFAULT var=exp<,var=exp>...
DEFAULT =m/t/n/b number
DEF KEY char: stats <:>
DEF KEY char; string
DEF PROC name <prm><,REF prm>...
DEF PROC name DATA
DELETE <ln> TO <ln>
DELETE a$<slicer> or b<slicer>
DO or DO WHILE or DO UNTIL
DPOKE address, number (0-65535)
DRAW TO x,y<,z>
EDIT <In>
EDIT a$ or EDIT ;b
ELSE stat
END PROC
ERASE <drive;>name
EXIT IF condition
FILL <INK or PAPER colour;> x,y
GET a$ or b
GET a$,x,y<,width,length><;type>
JOIN <In>
JOIN a$<slicer> TO b$<position>
JOIN a<slicer> TO b<position>
KEYIN strinq
KEYWORDS number
LET var=exp<,var=exp>...
LIST or LLIST <ln> TO <ln>
LIST DATA
LIST VAL
LIST VAL$
LIST DEF KEY
LIST FORMAT number
LIST PROC name
LIST REF reference
LOAD <drive;>name
LOCAL var<,var>...
LOOP or LOOP WHILE or LOOP UNTIL
OVER 0, 1 or 2
PLOT x,y<;string>
POKE address,string
POP <numeric variable>
<PROC> name <prm><,prm>...
READ LINE a$<,b$><,c$>...
REF reference
REF variable
RENUM <*><slicer><LINE 1><STEP s>
ROLL dir<,pixels><;x,y;wid,len>
SAVE <slicer;><drive;>name
SAVE DATA <drive;>name
SCROLL <dir><,pixels><;x,y;w,l>
SORT a$ or b<slicer><slicer>
SPLIT enter as "<>"
TRACE ln
TRACE:stat: stat:...: RETURN
UNTIL as in DO UNTIL/LOOP UNTIL
USING as in PRINT USING a$,number
VERIFY <slicer;><drive;>name
VERIFY DATA <drive;>name
WHILE as in DO WHILE/LOOP WHILE
WINDOW number<,x,y,wid,len>
XOS, XRG, YOS and YRG
as in LET XOS=number

FUNCTIONS

AND(number,number)
BIN$(number)
CHAR$(number)
COSE(number)
DEC(hex$)
DPEEK(address)
EOF(stream number)
FILLED()
HEX$(number)
INARRAY(a$(st<,slicer>),b$)
INSTRING(start,a$,b$)
ITEM()
LENGTH(number,string)
MEM()
MEMORY$()
MOD(number,number)
NUMBER(two-char string)
OR(number,number)
RNDM(number)
SCRN$(line,column)
SHIFT$(number,string)

ON as in GO TO ON x;ln,ln,ln....	SINE(number)
or GO SUB ON x;ln,ln,ln....	STRING$(repeats,string)
or ON x: stat: stat:...	TIME$()
ON ERROR In	USING$(format$,number)
ON ERROR: stat: stat:..: RETURN	XOR(number)

BETA BASIC TURTLE GRAPHICS

This large group of PROCEDURES written in Beta Basic will draw shapes on the screen in response to direct commands in a very similar way to LOGO commands. Commands can also be combined to make new PROCEDURES which can be called directly just by using their names.

Although LOGO has been used as a model, this program is not an implementation of that language and should not be compared to it other than superficially. It runs quite slowly as no attempt has been made to include machine code routines, but it illustrates the way a large program can be subdivided into simple modules using procedures. This Structural Approach to programming is necessary in most modern languages such as 'Pascal', 'C', 'LOGO', 'Forth' and 'Lisp'.

To use TURTLE load it from Tape with LOAD "TURTLE". If you wish to make a copy on Microdrive, type SAVETURTLE. This will save a copy to drive 1 with the name "TURTLE", and it can be reloaded with LOAD 1;"TURTLE".

Once loaded the program occupies lines above 9899 and can be restarted by typing TURTLE to call the initiating procedure. Line 1 consists of the one word TURTLE, so you can use RUN instead. This has the advantage of clearing the procedures stack as well.

Commands can now be entered in direct mode. ('K' mode has been turned off with the KEYWORDS command.) Take care not to press ENTER when a command is completed and the "OK" report is shown, or you will get an automatic LISTing that will wipe out anything you have drawn.

Remember that commands call Beta Basic procedures, so they follow the same syntax requirements. The simplest way to combine commands is to write a new procedure which incorporates them. However, to come closer to LOGO, there is a procedure TO (the zero is deliberate to distinguish it from the Basic word TO) which will write a new procedure with a given name.

Beta Basic's error handling routine is used to check when the turtle goes off the edge of the screen. It also traps other errors, reports their numbers and then waits for a key to be pressed before returning to the Turtle program. If you find that this causes looping, BREAK, then restart TURTLE or call the procedure SETERR before continuing.

Recursion, the calling of a procedure from within itself, is a very powerful feature of LOGO which allows very compact routines to be written. Beta Basic's procedures are recursive and you can adapt any LOGO programs easily. However, this is not really the most suitable method, firstly because they will run very slowly and secondly because the procedure data must be stored for each new call and memory can be quickly used up. It is recommended that repetitive programs are written as loops, which are more suited to Basic.

The FOR-NEXT and DO-LOOP structures are both suitable for this, but to keep to the style of LOGO there is a procedure REPEAT which will execute a fragment a number of times. This is intended for direct commands and should be used in new procedures with care. It cannot be used in recursive procedures.

To illustrate the power of turtle Graphics type in these commands, which call two procedures included in the package:

```
INSPIRAL 10,20,35
```

OUTSPIRAL 10,117,10

Now LISTP INSPIRAL and OUTSPIRAL to look at the very simple programs that produce the curves. Try changing some of the parameters.

You may feel that these programs are rather slow,. but a lot of checking is taking place to make TURTLE very robust so that it can be used by children and give a lot of flexibility. If you just want to play around with curves like these and are happy to do all your own checking, type QUICKTURTLE. This will delete most of the program leaving a remainder of the following commands, optimised for speed:

TURTLE, FD, BK, RT, LT, SETPOS, SETH, PD, PU, HOME, CS, INSPIRAL, OUTSPIRAL

You will lose the turtle, scale changes and type of boundary and you will have to manage colour changes etc. from Basic, but the increase in speed is gratifying.

To get the best from any Turtle Graphics you should read Seymour Papert's book 'Windstorms' (Harvester Press 1982, in paperback). A good, cheap guide to LOGO itself is 'Pocket Guide to LOGO' by Boris Allan (Pitman 1984) and 'Turtle Geometry' by Abelson and de Sessa is useful for the theory.

THE COMMANDS

(Many commands can be entered either in full or in abbreviated form, as shown.)

Command	Description
BACK D BK D	Moves the turtle back D turtle steps and draws a line if the pen is down.
CLEAN	Clears the screen, leaving the turtle in the same position.
CLEARSCREEN CS	Clears the screen, putting the turtle at the centre with heading 0.
DOT X,Y	Puts a dot at X,Y leaving the turtle in the same position.
FENCE	Prevents any line being drawn off the screen. Gives OFFSCREEN warning if used when the turtle is off-screen or when the end point of a line to be drawn would be off-screen.
FORWARD D FD D	Moves the turtle forward D turtle steps and draws a line if the pen is down.
HIDETURTLE HT	Prevents the turtle shape being shown, which speeds up drawing.
HOME	Moves the turtle to the centre of the screen, drawing a line if the pen is down.
LEFT A LT A	Turns the turtle left through an angle of A degrees.
PENDOWN PD	Puts the turtle's pen down, so that lines will be drawn when it moves.
PENERASE PE	Changes the turtle's pen into an eraser so that it rubs out any ink that it crosses. Note: It is not always possible to erase a line which has just been drawn by going over it backwards with PENERASE set.
PENREVERSE PX	Changes the turtle's pen so that it rubs out any ink that it crosses, but draws everywhere else (like OVER 1).

PENUP PU	Lifts the turtle's pen so that no lines are drawn when it moves.

REPEAT N ,"command: command:..."

	Repeats the list of commands in the string N times. This uses KEYIN and cannot be nested.
RIGHT A RT A	Turns the turtle right through an angle of A degrees.
SETBACKGROUND C SETBG C	Makes the background colour C. C must lie between 0 and 7.
SETBORDER C SETBR C	Makes the border colour C. C must lie between 0 and 7.
SETHEADING A SETH A	Makes the turtle point to A degrees. 0 is straight up and angles increase clockwise to 359 and then to 0 again. 90 degrees is a right angle
SETPENCOLOUR C SETPC C	Makes the pen colour C. C must lie between 0 and 8. The pen will change the ink in any attribute square it passes through, or if C=8 it will draw in the colour already there.
SETPOS X, Y	Moves the turtle to position X,Y and draws a line if the pen is down.
SETSCRUNCH X,Y SETSC X,Y	Changes the scale of the screen. This starts as 1 pixel to 1 turtle step and X and Y are each 100. Reducing X or Y will mean fewer turtle steps across the screen along that axis. For example, SETSCR 50,50 will double the size of each step along both axes, but SETSCR 50,100 will double the size of the steps in the X direction only.
SETX X SETY Y	Moves the turtle horizontally to the coordinate X without changing Y. Moves. the turtle vertically to the coordinate Y without changing X.
SHOWTURTLE ST	Makes the turtle visible so that it can be seen while it draws.
TØ name	Sets up the EDIT line to accept a sequence of commands which will be included in the program as a procedure with the given name. Each new procedure is given line number 1001 and all previous procedures are moved. up by 1 line.
TOWARDSØ X,Y	This is a pseudo function which calculates the direction in degrees from the turtle's position to X,Y and puts it in the variable TOWARDS.
WINDØW	(Zero is correct and distinguishes WINDOW from the Beta Basic command WINDOW.) Allows the turtle to leave the screen. The position is updated and the turtle will be shown again when it returns. Note: No line will be drawn when the turtle moves from one position which is offscreen to another which is also offscreen, even if it would be expected to cross the screen area.
WRAP	Makes the turtle wrap around the screen so that when it goes off one side it reappears at the opposite side. This is the initial condition. Gives OFFSCREEN warning if called when the turtle is offscreen.

DELETEP name	Deletes the procedure named from the program, as long as it occupies only one line.
ERASEP name,md	Erases the procedure named from the Microdrives, as long as it was saved by SAVEP. Defaults to Microdrive 1.
EDITP name	Brings the first line of the procedure named into the EDIT line.
LISTP name	Lists the procedure named.
MERGEP name, ln,md	Merges the procedure named from Microdrive to the line number given, as long as it was saved by SAVEP. Defaults to the line following the current line (with the '>' cursor) and to Microdrive 1.
RENUMP name, In	Renumbers the procedure named, as long as it occupies only one line, to the line given. Defaults to the line following the current line.
SAVEP name,md	Saves the procedure named, as long as it occupies only one line, as line 9999. Defaults to Microdrive 1.

THE VARIABLES

The following variables are part of turtle graphics and may be used in procedures:

XCOR YCOR	Coordinates of the turtle in the current scrunch.
XSCR YSCR	Scrunch, or scale, along each axis.
SCRUNCH	Ratio of XSCR to YSCR.
SHOWN	1 if turtle is shown, 0 otherwise.
PENCOLOUR, PC	Hold current 'pen colour.
BACKGROUND, BG	Hold current background colour.
BØRDER, BR	Hold current border colour.

BETA BASIC 3.0 / 4.0 FOR THE DISCIPLE AND PLUS D INTERFACES

If you have bought the program on tape:

The enclosed tape has been modified from the standard version of Beta Basic to suit the Disciple or PLUS D disc interfaces. To make a copy to disc, initialise the disc system, LOAD the program from tape, then MERGE the first part of the program once more. The Basic lines 1 and 2 must be modified to include "d1" before every LOAD and SAVE. Now RUN to SAVE the program to disc as "Autoload".

If you come to SAVE Beta Basic 4.0 twice to the same disc, ERASE d1"bbc2" first. If you don't do this, the "Overwrite ?" message will corrupt "bbc2" when it is saved via screen memory.

If you wish Beta Basic to control output sent via the parallel printer port (needed for correct LLISTing of Beta Basic keywords) then enter POKE @11,1: RANDOMIZE USR 58419. (Note: you can make a modified version of the interface's system file by making the POKE @11,1 and then saving the system file as described in your interface manual. In this case Beta Basic will be ready to LLIST as soon as it loads.) Line length can be set by poking address 57500 with the required number of characters per line. Beta Basic will LPRINT a line feed automatically after every carriage return. Depending how your printer is set, this may give double-spaced listings. If so, POKE 54989,24 to turn off the automatic line feed. POKE @11,0 will return control of the parallel port to the Disciple again.

The EOF (End-Of- -File) function will work correctly with Disciple serial file.

If required, Beta Basic can be turned off temporarily using RANDOMIZE USR 59904 and on again with RANDOMIZE USR 58419.

Some lines with Beta Basic keywords directly after Disciple commands cannot be entered; e.g:

```
10 CAT 1: ALTER TO PAPER 1
```

(Press BREAK for a second or two to escape such luck-ups.) This could be entered on two separate lines, or as:

```
10 CAT 1: PRINT; : ALTER TO PAPER 1
```

The following Beta Basic features do nut work with the Disciple: DEFAULT (device), "slicer" SAVEs to disc and ON ERROR with Disciple errors. (Disciple errors can be intercepted by another method - see FORMAT (the Disciple User Group Newsletter) issue 4