BETA BASIC 4.0

FOR SPECTRUMS WITH 128K MEMORY

BETA SOFT

CONTENTS

INTRODUCTION

Beta Basic 4.0 has been introduced to take advantage of the extra facilities offered by the 128K versions of the Spectrum. All the commands and functions of Beta Basic 3.0 (explained in the Beta Basic 3.0 manual) are still available. Existing Beta Basic 3.0 programs should run without alteration. (To make use of the new functions of Beta Basic 4.0. in such a program, replace the version 3.0 line zero in the program with a copy of the Beta Basic 4.0 line zero - see page 2 of the Beta Basic 3.0 manual for the basic principle.)

Beta Basic 4.0 provides a large increase in useful memory space by allowing you to hold arrays up to 64K long on the RAM disc. These arrays can be searched or sorted alphabetically very quickly. RAM disc files can be moved to or from streams, or to the screen. Interrupt-driven sound has been provided, giving full use of the sound chip without stopping the program. The graphics have been enhanced, with faster or more flexible versions of `CIRCLE`, `DRAW`, `PLOT` and `FILL`. Printers normally usable only in 48K mode can still be used. If you are upgrading from Beta Basic 3.0, we hope you enjoy all the new features. If you are a new Beta Basic user, you should read at least pages, 4-5 of the Beta Basic 3.0 manual to learn about entering new keywords. Beta Basic's editor is rather different from the 128K mode editor; if you are used to typing all keywords in full, you might want to type in:

```
(space)KEYWORDS 4
```

Pages 2, 3, 12 and 13 of the Beta Basic 3.0 manual also provide useful general information. From then on, dip into things as you like - and don't expect to exhaust the possibilities of the program in an afternoon!

If you want to know what other people get up to with Beta Basic, you might like to subscribe to our thriving Newsletter, which is packed with interesting examples, ideas, letters and advice. It is published at approximately bi-monthly intervals, and costs £5.00 for a 6 issue subscription in the U.K. (£5.50 in Europe, £6.00 elsewhere). Six back issues totalling 90 pages, are available immediately at the same prices. A trial issue can be obtained for £1.00 (worldwide).

If you encounter any problems with this program or manual, please let us know, with details, so that we can take action. We are always interested in your comments or suggestions.

BETASOFT, 92 OXFORD ROAD, MOSELEY, BIRMINGHAM, B13 9SQ

LOADING BETA BASIC 4.0

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 Beta Basic's machine cade, which is in two sections. The first section is about 18.5K long; it is loaded above RAMTOP. The second section is 6K long; it is loaded in the first place to screen memory and moved from there to banked RAM. (Using the screen memory like this allows the second section of code to be loaded or saved without getting in the way of any Basic program you have in memory at the time.) This arrangement of memory usage maximizes the available RAM disc space; you will have 73K available, just 1K less than the normal 74K. Since you can keep arrays and program sections in the RAM disc, your combined program and variables length can be over 90K. Individual arrays can be up to 64K long.

Loading has finished when you see the Betasoft copyright message at the bottom of your screen. (If you have loading problems, try a different volume setting or try the second copy of the program on the tape.) Lines 1 and 2 will be automatically deleted by this stage, and just line zero will remain. Normally only the line number is visible.)

BACKUP AND MICRODRIVE COPIES

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 or disc. 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 provide 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 you make the copies:

```
POKE 64218,0: POKE 64219,0: POKE 64220,0
```

This ensures that programs can be `RUN` but not altered.

Line 1 of the Basic loader is a `SAVE` routine which will save the current Basic program, followed by the machine code parts of Beta Basic. If you wish to use it, `LOAD` Beta Basic 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 a Microdrive or other storage device, modify lines 1 and 2 to the appropriate form. On the next page is a breakdown of what lines 1 and 2 do, with each statement numbered to make the explanations easier.

THE LINE ONE SAVE ROUTINE.

```
 1:1 LET rt=PEEK 23730+256*PEEK 23731
   2 SAVE "Beta Basic" LINE 2
   3 LET ch=PEEK 23631+256*PEEK 23632
   4 POKE ch,27
   5 POKE ch+1,17
   6 POKE ch+2,27
   7 POKE ch+3,17
   8 SAVE "bbc1" CODE rt+1,65367-rt
   9 RANDOMIZE USR 63039
  10 RANDOMIZE USR 59910
  11 SAVE "bbc2" CODE 16384,6144
  12 CLS
  13 STOP
```

Statement 1 assigns the current value of RAMTOP to the variable `rt` before the Basic program is saved by statement 2. RT will be saved with the program.

Statement 3 assigns the address of the "K" channel to a variable, for convenience. Statements 4 and 5 `POKE` the channel to prevent the "Start tape.." message appearing, and 6 and 7 make the Spectrum think there is a key being pressed; statement 8 therefore causes an immediate `SAVE`.

Statement 9 sets the whole screen to white-on-white attributes. It is included only to disguise the fact that statement 10 moves part two of Beta Basic's `CODE` from paged RAM to the screen; omit it if you like. Statement 11 saves part two of Beta Basic's `CODE` from screen memory.

The `CLS` (statement 12) restores the normal screen attributes.

Statements 3 to 7 can be omitted if your system (e.g. Microdrive) does not wait for a keypress and does not print messages as it `SAVEs`. (Note: the `POKEs` are temporary in effect.)

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 if desired, then turn Beta Basic off again using `RANDOMIZE USR 59904` before using `RUN` to `SAVE` the program. (`RANDOMIZE USR 58419` will turn Beta Basic back on.)

NOTE: If Beta Basic is initialised with Interface 1 attached, it makes some modifications to its own code to suit different issues of the Interface 1 ROM. This modification is not performed by copies of Beta Basic, only the original, so if you change to a different version of Interface 1 you will have to make another copy from your original tape. (Version 2 of the Interface 1 ROM was introduced at a serial number of 87316.)

THE LINE TWO LOAD ROUTINE.

If you use the line 1 SAVE routine, the Basic program will auto-start at line 2 when it is loaded back. Beta Basic's machine code will be loaded automatically.

```
 2:1 CLEAR rt
   2 LOAD "bbc1"CODE
   3 RANDOMIZE USR 63039
   4 INK 7
   5 LOAD "bbc2"CODE 16384
   6 RANDOMIZE USR 16640
   7 INK 0
   8 CLS
   9 RANDOMIZE USR 58419
  10 DELETE 1 TO 2
```

Statement 1 uses the RAMTOP (rt) that was saved with the program to reserve enough space for the first block of code, which is loaded by statement 2. (The size of this block will change as you define DEF KEYs and WINDOWs.)

Statements 3 and 4 are cosmetic only; they set the attributes of the whole screen to white-an-white, and INK to white, to disguise the fact that the second block of Beta Basic's CODE is loaded temporarily into screen memory by statement 5. Statement 6 calls a USR routine 256 bytes from the start of this block of code, copying it to paged RAM. (The USR will still work if you have loaded the block elsewhere, so long as you call an address 256 bytes from the block start.) One K of this code is loaded to the RAM disc area; if you have some RAM disc files present before Beta Basic is loaded, the first 1024 bytes of the first file will be corrupted. You could create a "dummy" first file so that later files are not affected; if so, do not later ERASE the dummy file! Statements 7 and 8 restore the normal screen attributes.

Statement 9 turns Beta Basic on, and the last statement deletes lines 1 and 2. If there are any further lines in the program, they will now be executed.

USEFUL USR CALLS

RANDOMIZE USR 58419 - Turns Beta Basic ON.

RANDOMIZE USR 59904 - Turns Beta Basic OFF. Sets 48K mode. The RS232 print channel is still usable. Provided you do not corrupt the printer buffer (e.g. by using a parallel printer), RANDOMIZE USR 58419 will still allow you to get back to Beta Basic, and from there to normal 128K mode, if you wish. RANDOMIZE USR 58419 and 59904 can be included in programs.

RANDOMIZE USR 59907 - Turns Beta Basic OFF. Goes to main 128K mode: menu with the program intact. Part of Beta Basic's code is moved to the top of the RAM disc space. You will over-write this if you use more than 68K of RAM disc space while in this mode; avoid doing so if you wish to re-activate Beta Basic later on. Line zero is renumbered as line 1; do not edit this line.

RANDOMIZE USR 59910 - Copies part 2 of Beta Basic from paged RAM to the screen for saving. Do not allow this code to be over-written by e.g. "Start tape...". See the POKEs in line 1.

BEEP ! p<,d><,n><,e><,ep><,v>

p=TONE PERIOD
d=DURATION
n=NOISE PERIOD
e=ENVELOPE
ep=ENVELOPE PERIOD
v=VOLUME

See also: `BEEP CLEAR`

The `PLAY` command provided on the 128 has many facilities, and it is satisfactory for playing tunes. Its great disadvantage is that the program stops while the sound is made. In this respect, `PLAY` is no better than the original `BEEP` command. However, the sound chip on the 128 is quite capable of playing sounds with minimal supervision from the computer's "brain". The new `BEEP!` command puts sounds into a queue and lets the program continue. An interrupt routine in Beta Basic checks the queue and feeds data to the sound chip 50 times a second, without your own Basic program having to do anything. All the facilities provided by the chip can be controlled, which explains the large number of possible parameters after `BEEP!`. However, do not despair - most of the parameters are optional. We can start very simply:

```
BEEP !400
```

The 400 is the tone period - the smaller the number, the higher the pitch. You can use a number between 0 and 4094. The sound lasted for half a second, which is the initial duration setting. For a longer duration, we could use:

```
BEEP !400,50
```

The 50 is the duration of the sound in 50ths. of a second. You must use a number between 0 and 255; 0 has a special purpose (see later). Values between 1 and 255 give durations between 0.02 and 5.1 seconds. (These numbers are perhaps not as "user-friendly" as octave numbers and crotchets, but they can be interpreted much faster, since they correspond almost exactly to what the computer and the sound chip are up to.)

You may have noticed that the example above gave an "OK" report before the sound finished. The next example makes this more obvious:

```
10 FOR n=200 TO 500 STEP 20
20 BEEP !n,25
30 NEXT n
```

(After writing this command I had to educate myself out of a tendency to wait until the sound had stopped before touching the keyboard again - this is quite unnecessary!) If we had omitted the "`,25`" the value we last used (50) would have been assumed for the duration of the sound. The simpler the form of the `BEEP!` you use, the greater the number of sounds that can be queued. More than 120 sounds per channel are possible if you use just a tone period. However, even using all parameters, you can still queue over 30 sounds per channel. You can `RUN` the example above quite a few times in succession before you run out of queue space. If you do run out of queue space, `BEEP!` will simply wait for an available space in the queue.

Sometimes you might want a sound to last longer than 5 seconds - something like a rocket motor sound should be ON until you turn it OFF. In this case use a duration of zero; e.g.:

```
BEEP !400,0
```

(Annoying, isn't it?) The sound will continue until another sound is put into the queue; e.g. `BEEP !400,1`. You could also stop the sound with `BEEP CLEAR` (see `BEEP CLEAR` in this manual).

The following example should be helpful to those using `BEEP!` to generate music. The `DATA` statement gives the tone period numbers for the notes in the `REM` statement below; for good measure line 70 gives the number required by the standard Spectrum Basic `BEEP` command (as a second parameter). Double the numbers in the `DATA` statement to drop the notes by an octave, and halve them to raise the notes by an octave.

```
10 READ p
20 BEEP !p,20
30 G0 TO 10

50 DATA 212,200,189,178,168,159,150,141,133,126,119,112
60 REM   C   C#  D   D#  E   F   F#  G   G#  A   A#  B NOTES
70 REM   0   1   2   3   4   5   6   7   8   9  10  11 BEEP
```

The old `BEEP` numbers and the tone period are related like this:

Tone Period=INT (125.95/2^((Beep Number-9)/12)+.5)

NOISE

You may have thought that the computer has been generating noise, but technically, it has been generating tones so far. The word "noise" is used for semi-random hissing sounds, whereas tones contain just one frequency. The sound chip can generate noise of different frequency ranges, controllable by the third `BEEP!` parameter. A value of 1 gives a hissing sound, while the maximum of 32 gives more of a rumble. Zero has a special action - it turns the noise off. Although the sound chip has 3 channels and can generate 3 different tones at once, it only has one noise period register and thus can only generate noise of one frequency range at a time. Anyway, let's have some noise:

```
BEEP !0,50,1: BEEP !0,50,10: BEEP !300,50,30
```

The first two `BEEP!s` used a tone period of zero, which turns off tone generation. The last `BEEP!` shows that you can have tone and noise at the same time, if you like. You could also have used a mare concise form of the line above, namely:

```
BEEP !0,50,1;0,50,10;300,50,30
```

The second and third `BEEP!s` have been replaced by semicolons to keep the data for different sounds separate. You can do this with complete freedom, however many parameters you use.

If you add "`:BEEP !400`" or "`;400`" to the end of the example above, you will get the same effect as you would get by adding "`:BEEP !400,50,30`". This is because the last values used (a duration of 50 and a noise period of 30) are assumed.

The noise effect will be cancelled when the channel goes quiet, or you can turn off the noise effect on the channel using a third parameter of zero; e.g.:

```
BEEP !0,50,0
```

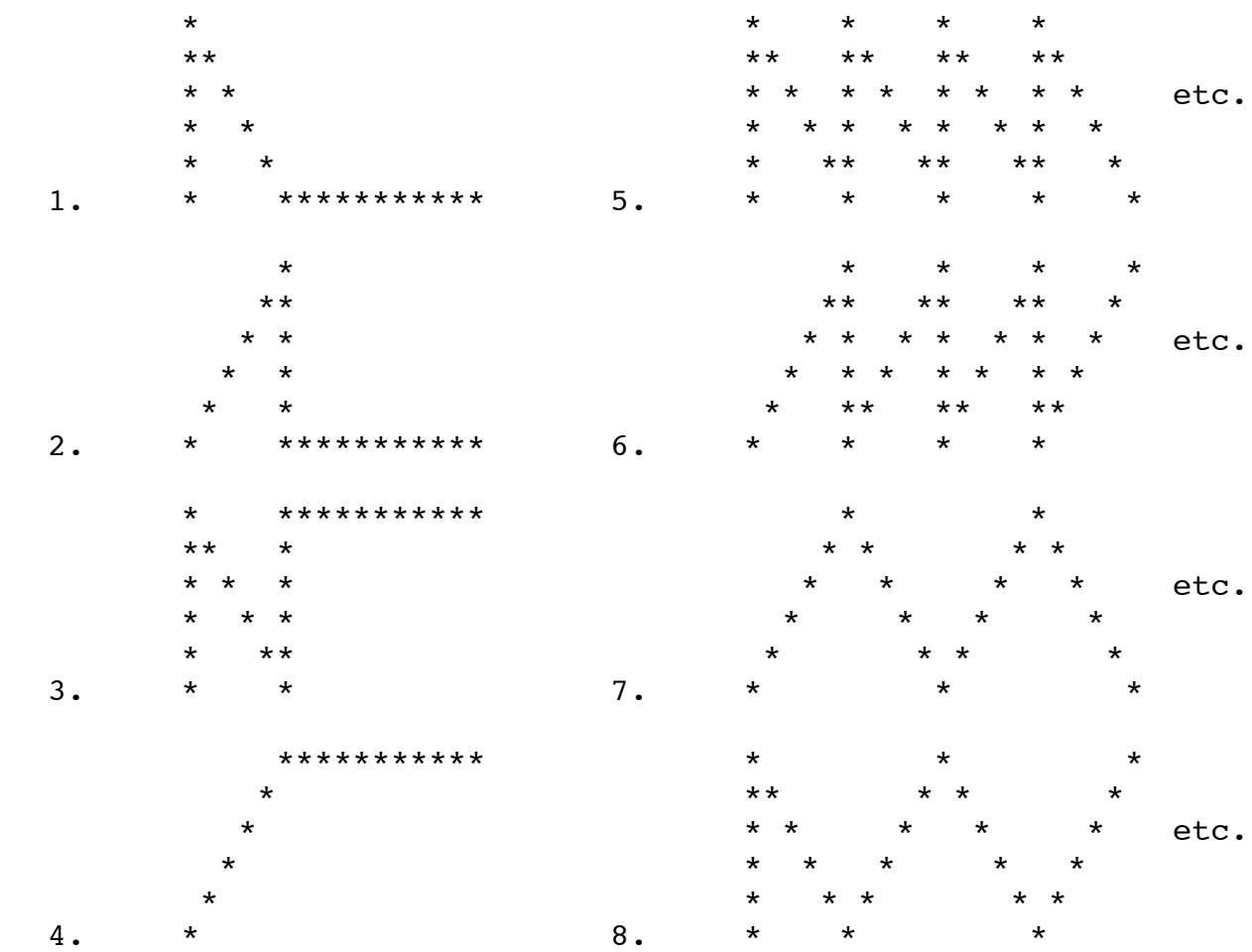
Ah yes, but suppose you want to turn off or alter the noise effect without specifying or altering the current duration? In that case you could use:

```
BEEP !0,-,noise period
```

Think of the minus sign as a dash, in this context. It means "do not change the current value". Any parameter can be replaced with a dash if you want to avoid altering it.

VOLUME ENVELOPES

So far, the various sounds we have produced have all been at a constant volume while they have lasted. However, the sound chip is capable of varying the volume while the sound is being produced. For example, a note can begin loudly and then die away, like a piano, or it can repeat rhythmically, like a machine gun. There are eight different patterns of volume change available. These are called volume "envelopes", and they are specified by a fourth parameter of from 1 to 8. (If you use zero, envelope control of the volume is turned off.) The eight envelopes are shown below:



You can see that envelope 1, for example, rises immediately to full volume and then slowly dies away, while envelope 2 increases volume slowly and then abruptly drops it to zero. Envelopes 3 and 4 end with maximum volume, and envelopes 5 to 8 repeat continuously. In all cases the sound will be turned off after the time given by the duration (second) parameter, although it may have died away already if the envelope dictates that.

As well as the basic form of the volume envelope, as given above, we need to specify how long it lasts. The fifth parameter of `BEEP!` is the envelope period, which determines how quickly the gradual "ramping" from minimum to maximum volume (or vice versa) is accomplished. The possible values are very wide - from 0 to 65535. A value of 7000 gives a ramp time of about a second. This sort of envelope period is useful for something like a collision sound (with envelope 1) or the "chuffing" of a steam train (with a repeating envelope such as 5). Much shorter envelope periods (less than 100) can be used with repeating envelopes to add the nasty background buzz expected of motorbikes, chainsaws, etc. Here are some examples:

```
10 BEEP !0,50,1,1,10000
20 BEEP !0,150,14,5,1000
30 BEEP !0,100,4,7,50
40 BEEP !700,50,0,1,10000
50 BEEP !0,0,1,7,20000
```

VOLUME

The final `BEEP!` parameter is the sixth, volume, which can vary from 0 to 15. When Beta Basic is loaded, it sets the volume on all channels to maximum. In the example below the volume is gradually reduced, then set back to maximum. As with the other parameters, the last-used value remains in effect until it is explicitly changed.

```
10 FOR n=15 TO 1 STEP –1
20 BEEP !400,20,0,0,0,n
30 NEXT n
40 BEEP !0,1,0,0,0,15
```

If you are using a volume envelope, the volume specified by parameter six will be over-ridden.

CHANNELS

So far we have generated only one sound at a time, using channel one, but you can use up to three channels at once by separating the data for the different channels with a "!". (Pronounced "shriek" if you want something shorter than "exclamation "mark".)

```
BEEP !400;25!200,50
BEEP !200,25;150;100!0,50,5;800,25,0!0,150;100,5
```

Things can get pretty complicated! The second example sends 3 ascending 0.5 second tones to channel one, a 1 second noise followed by a 0.5 second tone to channel two, and a 3 second *pause* (zero tone) to channel three, followed by a brief tone.

You can send data to a particular channel by omitting the data for earlier channels but using the exclamation marks. e.g:

```
BEEP !!200,40       - sound for channel two
BEEP !!!400,25      - sound for channel three
```

A TUNE

This tune, Frere Gustav, should provide a reassuringly familiar basis for experimentation. It featured in the original Spectrum manual.

```
 10 LET t=30,c=212,d=189,ds=178,f=159,g=141,gs=133
 20 BEEP !0,1,0,1,20000
 30 BEEP !c,t;d;ds,t/2;d;c,t
 40 BEEP !c,t;d;ds,t/2;d;c,t
 50 BEEP !ds,t;f;g,2*t
 60 BEEP !ds,t;f;g,2*t
 70 SEEP !g,3/4*t;gs,t/4;g,t/2;f;ds;d;c,t
 80 BEEP !g,3/4*t;gs,t/4;g,t/2;f;ds;d;c,t
 90 BEEP !c,t;2*g;c,2*t
100 BEEP !c,t;2*g;c,2*t
```

Line 10 sets the variable `t` which controls the speed of the music; all durations are specified in relation to this variable. The tone periods required in the music are assigned to convenient variables. Line 20 is used to set up an envelope which will be used throughout. Change this line to vary the effect; e.g:

```
20 BEEP !0,1,0,0
20 BEEP !0,1,0,5,10
20 BEEP !0,1,30
```

MONITORING THE SOUND QUEUES

Each of the three saund queues can hold up to 248 bytes, representing a variable number of sounds, according to how many parameters you have specified. You can check how many bytes are in use in the channel 1, 2 or 3 queues by `PEEKing` 49147, 49148 or 49149. You might want to do this if you are keeping say, a tune going in the "background" of a program. Periodically, you can run a, long `BEEP!` statement or series of statements, perhaps using `CLOCK` to activate a sound subroutine every 30 seconds or 50. If you do not know the state of the queue, you run the risk of either the queue running out, and (blissful) silence descending, or, of trying to put data into a full queue, leading to a pause in program execution. It is better in such cases to use something like:

```
IF PEEK 49147 < 150 THEN BEEP ! etc.
```

It may also be useful to know that a copy of the sound chip control register is kept at location 49150. It is bit-significant, so `PRINT BIN$(PEEK 49150)` is most informative. (Bit 7 is on the left, bit 0 is on the right.) The bits have the following significance:

BIT 7 - ALWAYS 1
BIT 6 - ALWAYS 1
BIT 5 - CHANNEL 3 NOISE
BIT 4 - CHANNEL 2 NOISE
BIT 3 - CHANNEL 1 NOISE
BIT 2 - CHANNEL 3 TONE
BIT 1 - CHANNEL 2 TONE
BIT 0 - CHANNEL 1 TONE

If a bit is zero, the specified property is ON.

Note: The sound queues occupy the upper part of the RAM disc space; if you have less than 2K of RAM disc space available, you may get an "Out of memory" report when you try to use `BEEP!`. If you `DIM!` or `SAVE!` while a sound is playing, the sound queues may be over-written, producing peculiar but harmless effects. `BEEP CLEAR` can be used to turn off all channels.

BEEP CLEAR <channel number>

See also: BEEP!

BEEP CLEAR is used to turn off a specified sound channel, or all three sound channels. When a channel is turned off, it is immediately silenced and its associated sound queue is cleared. The channel's volume is reset to maximum (15), and the channel's sound duration is reset to 0.5 seconds (25 interrupts).

```
   BEEP CLEAR
or BEEP CLEAR 0  - clears all 3 channels.

   BEEP CLEAR 1  - clears channel 1
   BEEP CLEAR 2  - clears channel 2
   BEEP CLEAR 3  - clears channel 3
```

CAT !

See also: CAT$ function

This command now gives the free RAM disc space in kilobytes after printing the list of files in the catalogue. The initial value is 73K, just 1K less than is normally available. If Beta Basic is not resident, or you prefer a more precise figure, use:

```
PRINT PEEK 23429+256*PEEK 23430+65536*PEEK 23431
```

CIRCLE x,y,radius

The CIRCLE command has been greatly speeded up compared to the Spectrum Basic version. The normal keyword, with a normal syntax, produces circles about 12 times faster than before. The circles are also more symmetrical and make better patterns when superimposed. It is possible for any or all points on the circle to be off-screen without the program stopping. With very large radii, a portion of a circle that "falls off" the bottom of the screen will wrap round and appear at the top, and vice versa. This feature is exploited 1n the example below:

```
10 FOR r=1 TO 255 STEP 2
20   CIRCLE 127,79,r
30 NEXT r
40 PAUSE 0
```

Note that circles can extend into the editing area of the screen - the PAUSE in line 40 keeps the pattern intact until you press a key.

You can, of course, use OVER, INK and PAPER as usual. Try the example above with OVER 1 and/or STEP 1.

DELETE line number TO line number

See also: DELETE in the Beta Basic 3.0 manual
MDP$ function in this manual

This command has been slightly modified from the form implemented in earlier versions of Beta Basic, so that now no error message is given if either of the two line numbers specified do not exist. The command deletes any program lines between the two line numbers, inclusively. This is convenient when using RAM disc "overlay" program sections.

DIM !name(dim1<,dim2><,dim3>....)
or DIM !file name,(dim1<,dim2><,dim3>...)

See also: INPUT!, LIST!, SORT!, INARRAY

DIM! creates arrays on the RAM disc which can be used in much the same way as arrays in normal memory. An individual array can be up to 64K in size. For an array of strings, the size is roughly the product of multiplying all the dimensions together; for an array of numbers, multiply this figure by five. RAM disc arrays can normally be considered as arrays, and the first form of the DIM! command reflects this. Sometimes it is convenient to think of a RAM disc array as a file, and the second form of DIM! might be appropriate. Let's deal with the "array form" first. Here are some examples:

```
DIM !fred$(100,10)  -  creates an array called "fred$" made up of 100 10-character strings.
DIM !long$(2000)    -  creates an array called "long$" made up of 2000 characters.
DIM !nums(100)      -  creates an array called "nums" made up of 100 numbers.
DIM !num#(5,5)      -  creates an array called "num#" made up of 5 rows of 5 numbers.
```

You must use names that start with a letter and consist of numbers or letters, apart from the last character. This must be a "$" for' a string array name, and it can be a "#" for a numeric array name (useful for showing "numeric array" in CAT! lists). Otherwise the last character must be a letter or a number.

This form of DIM! does not allow the name to be a variable, which might be desirable in some situations. (The program might do a CAT! and then ask "what do you want to call this array?" and ask for a string input, for example. This is where the second form of the command comes in. It allows the file name to be supplied as a variable, as it would be for SAVE!, LOAD!, ERASE!, LIST! or INPUT!. In the example below, note the comma after the name.

```
DIM !a$,(100,20)
```

If a$="name$", the string array "name$" will be created; if a$="abc" the numeric array "abc" will be created. You can also use a literal string if you like.

Now let's put some data in a RAM disc array:

```
10 DIM !fred$(20,10)
20 FOR n=1 TO 20
30  LET !fred$(n)=STR$ n+" hello!"
40 NEXT n
50 FOR n=1 TO 20: PRINT !fred$(n): NEXT n
```

Line 50 could be replaced by LIST !"fred$" (see LIST!).

RUN the example, then RUN 50. Notice that unlike a normal array, the RAM disc array is not erased by RUN (or CLEAR, or NEW).

However, if you run the whole example again, the DIM! will erase the old array before creating a new one. In general, RAM disc arrays can be used like normal arrays; however, there is a restriction on the makeup of expressions; references to RAM disc arrays must be at the start of the expression. For example:

`LET a$=!fred$(n,f TO g)+"qwerty"+x$` is allowed, but:
`LET a$="qwerty"+!fred$(n)` is not. You would have to use:
`LET t$=!fred$(n): LET a$="qwerty"+t$`

The RAM disc files created by `DIM!` are identical to the files created indirectly using `DIM` and `SAVE!`; i.e.

```
DIM !fred$(100,20)
```

gives the same as:

```
DIM a$(100,20): SAVE !"fred$" DATA a$()
```

This means that arrays created by `DIM!` can be loaded into the normal memory and saved from there to tape, Microdrive or disc, if memory allows. However, Microdrive users and most disc users can use `LIST!` and `INPUT!` to avoid any memory limitations during saving and loading. They can skip the next section if they like.

SAVING RAM DISC ARRAYS TO TAPE

If you are a tape user, and you run out of memory when you try to `LOAD!` a RAM disc array for saving to tape, here is one solution: Save the current Basic program, or most of it, to RAM disc, then `DELETE` most of the program, leaving a short control section near the start. You could also use `CLEAR`. An array up to about 20K can now be loaded from RAM disc and saved to tape. Use `LOAD!` or `MERGE!` to get the Basic program back. If you use `LOAD!` you must have saved the program `LINE` (something) if you want the program to keep running. If you use `MERGE!`, ensure that the line containing `MERGE!` and the next line after are not over-written; merge sections at higher line numbers. If you want to save a larger array (up to about 40K) you can still do it, but you will have to save the part of Beta Basic's `CODE` that is located in the normal memory to RAM disc, as well as your program. You could use something like this:

```
 10 LET rt=DPEEK(23730)
 20 SAVE !"program" LINE 100
 30 DELETE 100 TO
 40 RANDOMIZE USR 59907: REM BB off, goto 128K mode menu
 50 SAVE !"bbc1"CODE rt+1,65367-rt
 60 CLEAR 65535
 70 DIM a$(1): LOAD !"big$" DATA a$()
 80 SAVE "big$" DATA a$()
 90 LOAD !"program": REM Get rest of program back!
100 CLEAR rt
110 LOAD !"bbc1"CODE
120 RANDOMIZE USR 58419: REM Beta Basic ON
130 REM rest of program
```

When you see the menu, select 128K mode and `RUN 50`.

Note: The `DIM` in line 70 avoids a bug in 128K Basic which would otherwise become apparent here, with Beta Basic no longer ON.)

DRAW <TO> x,y

See also: DRAW and XOS in the Beta Basic 3.0 manual
PLOT in this manual

The DRAW command has been speeded up by about 2.5 times when drawing straight lines. It is possible to draw lines in the editing area of the screen, either during a normal DRAW (a relative DRAW) or during an absolute DRAW TO a specified x,y coordinate. You can use y coordinates ranging from 175 at the top of the screen to -16 at the bottom. It is also possible to add an offset to the y-coordinate (see XOS etc. in the Beta Basic 3.0 manual) so that the range is 191 to 0.

```
10 BORDER 6: INPUT ""
20 DO
30  DRAW TO RNDM(255),RNDM(191)-16
40 LOOP
```

Line 10 demonstrates a method of controlling the colour of the bottom two screen lines.

ERASE !

See also: SAVE!, CAT!, CAT$ function

This command works as it does under Spectrum Basic. A note on ERASE! speed may be useful: RAM disc files are stored one after another as they are saved; if you ERASE the first-saved file, all later files must be moved, which takes time. Try this:

```
10 SAVE !"one"CODE 0,30000
20 SAVE !"two"CODE 0,30000
30 PRINT "Erasing"
40 ERASE !"one"
50 ERASE !"two"
```

Now try it again with lines 40 and 50 reversed (which avoids having to move a later file).

The CAT$ function has the details of the most recently saved file at the start of the string it returns, so an efficient way to erase everything in the RAM disc is something like this:

```
100 DEF PROC rdclear
      DO UNTIL CAT$()=""
        ERASE !CAT$()(1 TO 10)
      LOOP
    END PROC
```

To use the procedure, just type:

```
(space) rdclear
```

FILL USING string;x,y

See also: GET and FILL commands, and MEMORY$ and STRING$ functions, in the Beta Basic 3.0 manual.

The FILL routine in Beta Basic 4.0 is much faster than in earlier versions; the whole screen can be filled in about a second. It is also possible to FILL USING any 16*16 pixel pattern, specified by a string. The string can be either a 2*2 character GET block (see GET), a normal string, or (using MEMORY$) any area of memory, including any set of four adjacent user-defined graphics. Here is a simple example:

```
10 PRINT CSIZE 16;"O"
20 GET a$,0,175,2,2
30 CIRCLE 127,87,60
40 FILL USING a$;127,87
```

If you PRINT LEN a$ you will see it is 39, characteristic of a 2*2 character, type-0 GET block. This is how FILL USING "knows" to treat the string as a GET block. To use a normal string instead, make up a 32-character string, such as the one below. (I have used BIN to make the relationship between the characters and the striped pattern more obvious. STRING$ produces a specified number of repeats of a string expression.)

```
50 LET a$=STRING$(32,CHR$ BIN 11001100)
60 FILL USING a$;0,0
```

To use the first four UDG's as a pattern source, use:

```
40 FILL USING MEMORY$()(USR "a");127,87
```

The MEMORY$ expression used here is actually just a 1 character string, but FILL USING assumes 32 bytes (four 8-byte UDG's) are to be used from the indicated area of memory. Thirty-two bytes are used no matter what the string length, unless it is exactly 39. Any area of memory can be used as a pattern source. FILL assumes it is organised in sets of eight bytes like the UDG graphics area. The first UDG or equivalent forms the top left of the pattern, the second forms the top right, the third forms the bottom left, and the fourth is the bottom right. Random patterns can be quite pleasing; try changing USR "a" above to, say, 1 (using the ROM as a pattern).

You can specify INK for a normal FILL, or PAPER for a FILL of an area that is already all INK (see Beta Basic 3.0 manual). Less usefully (due to attribute limitations) you can specify INK and PAPER, as well as a FILL pattern. INK *and* PAPER must come after USING in the command.

```
10 REM bricks
20 LET a$=""
30 FOR n=1 TO 8
     READ a
     LET a$=a$+CHR$ a
   NEXT n
40 LET a$=STRING$(4,a$)
50 DATA 16,16,16,255,1,1,1,255
60 CIRCLE 127,87,60
70 FILL USING a$; INK 1;127,87
```

This looks O.K., but specifying PAPER as well, as in:

```
70 FILL USING a$; INK 7; PAPER 2;127,87
```

shows up the Spectrum's attribute limitations. If you delete line 60 and fill the whole screen, or any other area that finishes at character square edges, there is no problem.

One advantage of pattern FILLs is that they can be used instead of colour, and without the associated attribute problems. Add the following lines to lines 10 to 50:

```
60 REM check
70 LET b$=STRING$(8,CHR$ BIN 11001100+CHR$ BIN 11001100
   +CHR$ BIN 00110011+CHR$ BIN 00110011)
80 REM horizontal stripe
90 LET c$=STRING$(16,CHR$ 255+CHR$ 0)
100 CIRCLE 88,87,70
110 CIRCLE 158,87,70
120 FILL USING a$;127,87
130 FILL USING b$;70,87
140 FILL USING c$;160,87
```

RUN to show the three different patterns. Try filling the outside of the circles. These simple patterns do not fully exploit the 16*16 pixel design area. You could use symbols such as trees, houses, products, etc.

Note: FILL uses part of the RAM disc area as a "scratch-pad". If you are using over 63K of RAM disc space, FILL will generate an "Out of memory" report.

FORMAT "p";printer mode

Beta Basic 4.0 uses the normal FORMAT syntax for altering the RS232 baud rate. For the benefit of those unfortunate enough to have the miserable Spectrum Plus 128K manual (all 16 pages of it!) you can set the baud rate by e.g.:

```
FORMAT "p";1200
```

Valid baud rates are 50, 110, 300, 600, 1200, 2400, 4800 or 9600 (the initial setting). Normally anything you LPRINT or LLIST is sent via the RS232 port.

Beta Basic 4.0 recognises three "baud rate" values (0, 1 and 2) as not being genuine baud rates at all, but printer mode settings. The characteristics of the "p" channel are altered; any streams that have been OPENed to this channel will be affected. (Stream 3, the LPRINT and LLIST stream, is attached to the "p" channel when the Spectrum is turned on.)

```
FORMAT "p";0
```

This gives normal RS232 output, as in 128K mode without Beta Basic present. This is the initial setting. Printer mode 0 gives "text" type RS232 output. A CHR$ 13 (carriage return) generates an automatic CHR$ 10 line feed). Other control codes CHR$ 0-31) are not sent. Keyword tokens are expanded (e.g. CHR$ 255, the COPY token, is sent as C,O,P,Y).

```
FORMAT "p";1
```

This sets RS232 output to "binary", type. All characters are sent unchanged, allowing control codes to be sent to the printer, or data to be sent to another computer. Thus a CHR$ 255, for example, is sent as CHR$ 255, not C,O,P,Y - this is what you want for bit-image data or control codes. You can, of course, send text in this mode, but you may have to send CHR$ 10's (line feeds) yourself after every carriage return, or set your printer to do an automatic line feed when it receives a carriage return. You can switch between "text" and "binary" mode even when Beta Basic is not present with some simple POKEs:

```
POKE 23349,36: POKE 23350,1: REM text mode
POKE 23349,39: POKE 23350,1: REM binary mode
```

```
FORMAT "p";2
```

This sets printer output to 48K type. A ZX printer should work correctly, as should many printers driven by interfaces plugged onto the edge connector. See PRINTERS in this manual for details. COPY should work with printers or interfaces that provide COPY in 48K mode.

If Beta Basic is not resident, you will find that most printers that work in 48K mode do not work in 128K mode. This is because many interfaces and printers use the 48K mode printer buffer, which is used in 128K mode to hold system variables, a temporary stack, a bank-switching routine, etc. Beta Basic also needs to use this area (in order to be compatible with ROM routines) but it swaps the contents with a "shadow" printer buffer before `LPRINTing` or `LLISTing`, provided printer mode 2 has been set. Everything is swapped back after the `LPRINT` or `LLIST` is complete. The "shadow" buffer can accumulate characters before a carriage return occurs (e.g. if you are using a ZX-type printer) or hold printer driving software. You might want to set the initial contents of this "shadow" buffer, perhaps loading such a printer driver. The `CODE` block "bbc2" (the second block of Beta Basic's `CODE`) begins with 256 zero bytes destined for the "shadow" printer buffer it can be modified by loading it somewhere in memory, overwriting the first 256 bytes (perhaps with a copy of the printer driving software you use in 48K mode) and saving it again. `SAVE` 6144 bytes of `CODE`. Use the modified block of code when you load Beta Basic. If you now use the line 1 `SAVE` routine, the modified "shadow" printer buffer will be saved as part of Beta Basic's code.

INPUT stream;!filename

See also: LIST!

INPUT! loads a file to the RAM disc from a stream. You need a Microdrive or a disc system that supports serial files for this command to be useful. The files that you INPUT! will normally have been created by LIST!. The example below reads the file "filename" from Microdrive and places it in the RAM disc with the name "fred$" (which would be appropriate if the file is a string array previously saved using LIST!).

```
10 OPEN #5;"m";1;"filename"
20 INPUT #5;!"fred$"
30 CLOSE #5
```

The advantage of doing the OPEN and CLOSE from Basic is, that this, allows various disc drive systems to make use of INPUT!; if Beta Basic did the OPEN and CLOSE automatically from machine code, the command would be machine-specific. Of course, you can simplify matters using a procedure or a DEF KEY.

LIST <stream;>!filename

LIST! lists a RAM disc file. If you do not specify a stream, the file will be listed to the screen; e.g.:

```
LIST !"fred$"
```

LIST! to any stream that is open to an "S", "P" or "T" channel (which normally deal with printable characters) causes arrays and programs to be translated slightly to make them more understandable, rather than output exactly as they are stored. If the file is a string array, the strings will be listed one at a time, with a carriage return after each one. Numeric arrays are listed with each number on a separate line; for a two dimensional array, the order is (1,1), (1,2), (1,3) etc. If the file is a program, it will be listed just as though it was present in the normal memory. RAM disc CODE files are likely to cause "Invalid colour" or other errors if you try to LIST! them to the screen. LLIST! acts similarly to LIST!, but uses stream three, which is normally directed to a printer.

LIST! allows RAM disc files to be saved to Microdrive or disc without passing through the normal memory; this is essential when dealing with 60K arrays! Here is an example which saves an array of blank strings:

```
10 DIM !name$(120,10)
20 OPEN #6;"m",1;"name$"
30 LIST #6;!"name$"
40 CLOSE #6
```

It is important that you CLOSE the stream, or you will lose part of the array.

There is no reason why you cannot LIST! and INPUT! non-array files to and from streams. Remember, though, that LIST! creates a *serial* file which (depending on your system) is probably not LOADable, only INPUT!able.

PLAY

See also BEEP! in this manual.

The PLAY command works normally. UDG "U" is the token for both PLAY and USING; context is used to determine which is shown on the screen or printer. This might cause otherwise puzzling effects. Enter:

```
10 PLAY "acd"
```

Now edit the line and add "PRINT" after the line number; context will change the display to PRINT USING (which requires a different syntax).

PLOT x,y

See also: DRAW in this manual, and XOS in the Beta Basic 3.0 manual.

PLOT now allows points to be plotted in the lower screen. The permitted y-values run from 175 at the top of the screen to -16 at the bottom. It is worth pointing out that Spectrum Basic also allows negative y values, but it simply uses the absolute value when the point is plotted. This could conceivably require a program to be altered slightly in order to run as before.

```
10 FOR y=-16 TO 175
20 PLOT 0,y
30 DRAW 255,0
40 NEXT y
```

It is possible to set a y-axis offset (see XOS etc.) so that the permitted y values run from 0 to 191.

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

See also: ERASE!, CAT$ and MDP$ functions in this manual.

SAVE! allows part of the program, or just the variables, to be saved to RAM disc (see below). (See page 59 of the Beta Basic 3.0 manual for related information and examples.) It also corrects a bug in 128K Spectrum Basic which can affect string and string array SAVE! and LOAD!

```
SAVE !1000 TO 2000;"part2"
SAVE ! DATA "vars4"
```

The RAM disc is very fast compared with tape, Microdrive or disc drive. This means that long programs can be loaded in sections from RAM disc without a serious speed penalty. The main program can be a fairly short control section which DELETEs the last lines of the program (see DELETE in the Beta Basic 3.0 manual) and then MERGEs a new section into the space made available. This process can be performed numerous times so that you have the possibility of Basic programs 60 or 70K long. The MERGEd sections would probably be created in the first place using SAVE with a "slicer". You will almost certainly want to do the SAVE to tape, Microdrive or disc at some stage - remember that the RAM disc loses its contents when you switch off! Assuming that you have 3 program sections on tape, with line numbers of 1000 or more, your auto-running program to LOAD Beta Basic and the sections after turning on the computer could be something like this:

```
   1 (SAVE routine - MERGE back Beta Basic's loader and modify as desired.
See INTRODUCTION.)
   2 (LOAD routine - see above. Program auto-runs from here.)

  10 MERGE "sect1": SAVE !1000 TO;"sect1": DELETE 1000 TO
  20 MERGE "sect2": SAVE !1000 TO;"sect2": DELETE 1000 TO
  30 MERGE "sect3": SAVE !1000 TO;"sect3": DELETE 1000 TO
```

The main program might be a "menu" system that MERGE!s the sections (often called "overlay files") as required, e.g.:

```
  40 CLS: DELETE 1000 TO
  50 PRINT "1. Input data"
  60 PRINT "2. Display data"
  70 PRINT "3. Calculate results"
  80 INPUT choice
  90 IF choice=1 THEN MERGE !"sect1": GOSUB 1000
 100 IF choice=2 THEN MERGE !"sect2": GOSUB 1000
 110 IF choice=3 THEN MERGE !"sect3": GOSUB 1000
 120 GO TO 40
```

This isn't very well structured, but I hope it is fairly understandable to those not familiar with Beta Basic. You may want to replace the GOSUBs with procedure calls and the GOTO with a DO loop. You can add sophistications like checking if the section already in memory is the one you need, before doing the DELETE and MERGE. (MERGE might take seconds, even from RAM disc, due to the extensive shuffling of memory required.) You could have slightly different copies of the same DEF FN in each section, e.g:

```
1000 DEF FN n$()="sect1": REM or "sect2" or "sect3"
```

This would allow your program to tell which section was resident by checking the value of FN n$(). Obviously you could achieve a similar object using a procedure, or a subroutine at a fixed line number. The function MDP$ provides a way of automatically loading non-resident procedures.

SORT !RAM disc string array
or SORT INVERSE !RAM disc string array

See also: SORT in Beta Basic 3.0 manual.
FP$ function in this manual.

The SORT command in Beta Basic 4.0 will sort RAM disc string arrays, as well as normal string and numeric arrays. RAM disc numeric arrays cannot be sorted, but Beta Basic provides several methods of coding numbers as sortable strings. See CHAR$, FP$ and USING$. To demonstrate a SORT, we need to set up an array:

```
10 DIM !rand$(20,8)
20 FOR n=1 TO 20
30   FOR c=1 TO 8
40     LET !rand$(n,c)=CHR$(RNDM(25)+65)
50   NEXT c
60 NEXT n
70 LIST !"rand$"
```

Now SORT and LIST the array:

```
SORT !rand$(): LIST !"rand$"
```

You can SORT in inverse order using INVERSE, and you can SORT just some of the strings in the array, or specify a substring to sort on, as you can for sorting a normal array; e.g.:

```
SORT INVERSE !rand$(1 TO 10)(2 TO)
```

This SORTs the first 10 strings into reverse order, using the second and later characters.

With the large arrays made possible by Beta Basic 4.0, you may find that sorting takes an appreciable time. An array of 1000 random 60-character strings will be sorted in about 14 seconds. (On the other hand, a much more up-market machine, with more expensive software, and a "real" disc drive, could probably do it in....14 minutes or so.) The time will increase a lot if the array is largely empty; it is a good idea to keep the number of used strings as a variable, and sort only that many. This keeps the blanks from being sorted to the top, too! If you use Beta Basic's improved BREAK to interrupt a SORT, your RAM disc array or normal string array will be partially sorted, but each string will be intact.

FUNCTIONS

See also: Beta Basic 3.0 manual, page 68.

CAT$()
FN Y$()

This function returns the entire RAM disc catalogue as a string. It is useful for finding out, from within a program, whether a file exists in the RAM disc. For example, if a file exists, you might want to do an `ERASE!` to allow you to `SAVE` a RAM disc file with the same name. Or you might want to divert the program from attempting a `LOAD!` of a non-existent file. `CAT$` can also be used as part of improved `CAT!` routines; as well as file names, it provides information on file type and file length.

The string returned by `CAT$()` contains 13 characters for each file in the RAM disc. The first 13 characters correspond to the file that was created most recently, the next 13 to the file that was created before that, and so on. The total number of files is given by `LEN CAT$()/13`

The simplest way to check if a file exists on the RAM disc is to use `INSTRING` on the `CAT$`; e.g:

```
10 INPUT "Name to SAVE under:";n$
20 IF INSTRING(1,CAT$(),a$)<>0 THEN ERASE !a$
30 SAVE !a$
```

For each file entry:

Characters 1 to 10 are the file name.
Character 11 gives the file type:
`CHR$ 0` = PROGRAM
`CHR$ 1` = NUMERIC ARRAY
`CHR$ 2` = STRING ARRAY
`CHR$ 3` = CODE

Characters 12 and 13 give the file length, coded as more significant and less significant bytes. These characters can be converted to a number using the `NUMBER` function.

Here is one of many possible approaches to an improved RAM disc catalogue command. Type: `(space)ncat` to run it.

```
 10 ncat
 20 STOP
100 DEF PROC ncat
    LET a$=CAT$ ()
    FOR f=1 TO LEN a$ STEP 13
      PRINT a$(f TO f+9);
      ON CODE a$(f+10)+1
        PRINT" Program";
        PRINT" Numbers";
        PRINT" Strings";
        PRINT" CODE ";
110   PRINT " Length:";NUMBER(a$(f+11 TO f+12))
    NEXT f
    END PROC
```

FP$(number,number of characters)
FN F$(number,number of characters)

See also: NUMBER function in this manual.

FP$ stands for Floating Point string. This function converts numbers into strings of 3, 4 or 5 characters. The NUMBER function performs the reverse conversion. FP$ has four main applications:

1. It provides a means of placing numeric information alongside associated string information in strings (probably in string arrays). This is convenient as it allows each string in an array (corresponding to an indiv1dual record in a database) to contain all relevant information. For different kinds of numeric information, CHAR$ or USING$ may be more appropriate - see the Beta Basic 3.0 manual. See also the discussion starting near the bottom of page 61 in that manual.
2. FP$ allows you to set up RAM disc arrays of string-coded numbers which can be SORTed. (SORT will not work with RAM disc arrays of actual numbers.)
3. String arrays can be SORTed faster than number arrays, and you might consider using FP$ for this reason.
4. FP$ allows you to set up arrays with a flexible balance between precision and memory usage. Each length of FP$ gives the complete *range* of possible numeric values (about 10E-39 to 10E38) but a different precision:
   Three characters: 5 digits are usually correct.
   Four characters: 7 digits are usually correct.
   Five characters: 9 digits are usually correct. This is the maximum precision available on the Spectrum. Not all the digits are displayed, although numbers are stored and manipulated at this level of accuracy.

A normal number occupies five bytes. Below is an example showing the precision of 4 and 3 character FP$'s. (Five-character FP$'s are just as precise as normal numbers.)

```
10 PRINT " Original    4 Chars    3 Chars"
20 FOR n=1 TO 20
30   LET x=RND*200000-5000
40   LET a$=FP$(x,4)
50   LET b$=FP$(x,3)
60   PRINT x;TAB 11;NUMBER(a$);TAB 22;NUMBER(b$)
70 NEXT n
```

INARRAY(RAM disc file, target string)
FN U(RAM disc file, target string)

See also: INARRAY function in Beta Basic 3.0 manual.

INARRAY has been modified to allow string arrays held on the RAM disc to be searched. Read about the Beta Basic 3.0 version first. INARRAY returns the number of the first string which contains the target string, or zero if it is not found. The example below looks for "SMITH" in the RAM disc file "name$", starting at the first string:

```
10 DIM !name$(100,10)
20 LET !name$(100)="SMITH"
30 PRINT "Found at string:";INARRAY("name$(1)","SMITH")
```

To start looking at the 10th string, you would use:

```
PRINT INARRAY("name$(10)","SMITH")
```

The required form of the RAM disc file name may look slightly odd, but it is necessary to get past the ROM syntax check for functions. If you wish to provide the RAM disc file name as a *variable*, something like:

```
30 LET a$="name$": PRINT INARRAY(a$+"(1)","SMITH")
```

will work. Do NOT use:

`INARRAY(a$(1),"SMITH")` Or: `INARRAY(a$,"SMITH")`

or you will be attempting to search the string `a$`.

In the example, the individual strings are quite short (10 characters) and little time is spent searching the entire length of each one. With longer strings, it is faster to specify the section which will contain the target string, if you can. (You will often want to do this anyway, for database applications.) For example, it would take 0.66 seconds to search the whole of an array of 1000 60-character strings, but if you knew the target string was at the start of a string, you could use:

```
PRINT INARRAY("name$(1,1 TO 5)","SMITH")
```

which would cut the time down to just 0.26 seconds. The `TO`, and any other keywords you want inside the quotes, have to be entered as keywords; either enter them using 48K-type shifts, or press `ENTER` before you put the quotes in, then add quotes after Beta Basic has "tokenised" the keywords.

MDP$()
FN X$()

See also: DELETE in this manual and ON ERROR in the Beta Basic 3.0 manual.

MDP$ stands for Missing DEF PROC string. Its major use is in error-handling routines which automatically LOAD procedures which are not present in memory when you try to use them. This will be fastest from RAM disc, but it is also possible from Microdrive or disc drive.

The value of MDP$() is altered every time you try to use a non-existent procedure. Type:

```
(space)testing
```

This will give you a "Missing DEF PROC" report (unless you really have a procedure called "testing"). Now:

```
PRINT MDP$()
```

and you will get "testing". We can put this to work in an error-handling routine. (See ON ERROR in the Beta Basic 3.0 manual.) First, create a procedure:

```
9000 DEF PROC testing n
       FOR x=1 TO n
         PRINT n,"Beta"
       NEXT x
9010 END PROC
```

SAVE it to RAM disc with:

```
SAVE !9000 TO 9010;"testing"
```

Enter these lines:

```
10 ON ERROR 100
20 testing 5
30 STOP

100 IF error=32 THEN MERGE !MDP$( )
      ON ERROR 100
      POP
      CONTINUE
    ELSE STOP
```

RUN this to see what happens without any errors. Then delete lines 9000 and 9010 and RUN again. The missing procedure being used at line 20 will cause line 100 to be GOSUBed. There, a check is made that we actually have a "Missing DEF PROC'" error (which is error 32 - see Appendix C in the Beta Basic 3.0 manual). If so, the missing procedure is MERGEd. Of course, we could have MERGEd a file called something other than MDP$, such as "p."+MDP$(), or "moreprocs".

After the MERGE, ON ERROR is turned on again, and the POP: CONTINUE causes line 20 to be re-executed and this time it will work! The whole thing is quite fast and allows programs to be modular, using the RAM disc to contain large sections of the program. DELETE (see DELETE in this manual) can be used to make room for the MERGEd sections.

NUMBER(string)
FN N(string)

See also: NUMBER function in the Beta Basic 3.0 manual.
FP$ function in this manual.

The NUMBER function in Beta Basic 4.0 has been modified to allow it to convert the 3, 4 or 5 character strings created by FP$ back into numbers. The 2-character strings created by CHAR$ are converted to integers, as before, but 3-5 character strings are instead converted to floating point numbers that can be positive or negative. The entire numeric range of the Spectrum (about 10E-39 to 10E38) can be handled.

ERROR CODES

See also: ON ERROR and Appendix C in Beta Basic 3.0 manual.

Beta Basic allows you to intercept errors using the ON ERROR command. The variable "error" is created with the following values for the various errors unique to the 128K Spectrum.

ERROR VALUE	MESSAGE
170	MERGE error
171	Wrong file type
172	CODE error
173	Too many brackets
174	File already exists
175	Invalid name
176	(not used)
177	File does not exist
178	Invalid device
179	Invalid baud rate
180	Invalid note name
181	Number too big
182	Note out of range
183	Out of range
184	Too many tied notes

DISC INTERFACES

OPUS DISCOVERY 1

There are special versions of Beta Basic 3.0 and 4.0 for this system. They support simplified `SAVE/LOAD` syntax, `DEFAULT`, `EOF`, correct `LLIST` via the built-in printer interface, and `ON ERROR` with disc errors. To obtain these versions of the program, send us £2.00 for a copy on tape, or £3.50 for a copy on disc. Quote the approximate date of purchase of Beta Basic if you bought direct from us, or return your original tape for exchange.

TRL BETA DISC

This disc drive is broadly compatible with Beta Basic. One problem is that Beta Basic is turned off after `LOAD` and `MERGE` from the disc. You can use `RANDOMIZE USR 58419` to turn Beta Basic on again. You will also need to modify the line 1 and 2 `SAVE` and `LOAD` routines to the Beta disc syntax, which requires splitting the lines up into shorter sections. See INTRODUCTION for an explanation of how these lines work. You will be able to simplify use of the disc commands by means of `DEF KEYs` and procedures. (There are some examples in our Newsletter.)

SPDOS Disc System

There is a memory conflict between Beta Basic and the disc operating system. You can get a cheap memory paging program to overcome this problem (for Beta Basic 3.0, at least) from the address below. Write to them for details.

SOUTHFIELD SOFTWARE, 64 SOUTHFIELD ROAD, OXFORD, 0X4 1PA

SPECTRUM PLUS 3 Disc System

This has not been released at the time of writing. Betasoft would almost certainly support the machine, and provide upgrades at reasonable prices. Send us an S.A.E. for details once the machine has been released.

DISCIPLE

The Disciple interface turns off Beta Basic when a disc error such as "File not found" occurs. It can be turned on again by `RANDOMIZE USR 58419`. You may need to turn Beta Basic off temporarily to avoid problems with some disc commands; this can be done with `RANDOMIZE USR 59904`. You can include these ON and OFF instructions in a program, perhaps as part of a SAVE procedure. Alternatively, Beta Basic can be `POKEd` so that it no longer deals with particular disc commands (see later section).

You need to make a modified copy of the Disciple's "system" file. Have ready a disc with at least 35K of free space. Initialise the disc system using `RUN`, then `LOAD` Beta Basic from tape. Use `RANDOMIZE USR 59904` to turn the program off. Then type in and `RUN` the program on the next page. Do NOT include a `CLEAR` statement. The program searches the system file in memory for particular bits of code and then alters them. The modified file is then saved as "`bbsys`".

```
10 LOAD d1"system"CODE 33000
20 LET X=FN i(1,FN m$()(33000 TO 39143),"!"+CHR$ 244+CHR$ 27)
30 POKE X+33001,252
40 LET X=FN i(1,FN m$()(33000 TO 39143),"!"+CHR$ 125+CHR$ 27)
50 POKE X+33001,252
60 LET X=FN i(1, FN m$()(33000 TO 39143),"!"+CHR$ 73+CHR$ 19)
70 POKE X+33000,75: POKE X+33001,251
80 SAVE d1"bbsys"CODE 33000,6144
```

You can now delete these lines. MERGE from tape a copy of Beta Basic's line 1 and 2. Modify them to the Disciple syntax (see INTRODUCTION for an explanation of how they work.) You might want to alter "Beta Basic" in line 1 to "Autoload", Delete the last statement in line 2. Replace it with:

```
LOAD d1"bbsys"CODE 0
```

Turn Beta Basic on with RANDOMIZE USR 58419, then add a line:

```
3 DELETE 1 TO 3
```

Turn if off again with RANDOMIZE USR 59904. Finally, RUN to save Beta Basic to disc.

Below are POKEs to remove control of certain commands from Beta Basic. This should make them work normally with the Disciple, but you will lose the Beta Basic features like slicer SAVEs and RAM disc access. The original values are given in brackets - they can be POKEd back to restore control.

```
SAVE: POKE 64844,223: POKE 64845,26: (45,237)
LOAD: POKE 64826,224: POKE 64827,26: (42,237)
CAT,FORMAT,MOVE,ERASE,OPEN,CLOSE,MERGE,VERIFY:
      POKE 64609,215: (207)
```

Send us an S.A.E. if you have problems - we may be able to work cut a more elegant solution.

PRINTERS

See also: FORMAT in this manual
APPENDIX E in the Beta Basic 3.0 manual.

Beta Basic controls the line length of all LPRINTed and LLISTed output according to the contents of location 57500. This starts at 80 but can be POKEd to a different value. The value should be less than or equal to that selected on your printer or interface, in order for program listings to be correctly indented. The first line may be the wrong length unless you also POKE 57545,line length+1 (This is a column position variable.) FORMAT selects between RS232 and parallel type printers. All these settings are saved with Beta Basic.

SERIAL PRINTERS

These should work normally with the program as supplied.

ZX-TYPE PRINTERS (ALPHACOM)

Enter: FORMAT "p";2: POKE 57500,32: POKE 57545,33

ZX LPRINT III

Use the printer interface in 48K mode to make sure the interface has copied its code to the printer buffer. Then:

```
SAVE "buffer"CODE 23296,256
```

LOAD Beta Basic normally, in 128K mode. Then (without using CLEAR) enter:

```
LOAD "bbc2"CODE 30000
LOAD "buffer"CODE 30000
RANDOMIZE USR 30256
```

This places the interface software in paged RAM as part of Beta Basic's code. Now enter:

```
DPOKE 61081,23296: FORMAT "p";2
```

See below for information common to ZX LPRINT III/Kempston.

KEMPSTON "E"

Load Beta Basic, then type in and RUN this program:

```
10 FORMAT "p";2
20 FOR n=57893 TO 57899
30 READ a: POKE n,a: NEXT n
40 DATA 87,219,251,243,195,146,15
50 DPOKE 61081,57893
```

You can delete the program once it has been run. It makes changes to Beta Basic's code; these changes will be saved with the program if you use the line 1 SAVE routine.

KEMPSTON/ZX LPRINT III: Common features.

MERGE the Basic loader (see INTRODUCTION) and add to line 2 a POKE 23679,100 statement anywhere before the DELETE statement. This sets characters-per-line to a high value, allowing Beta Basic's system variable at 57500 (which can be changed) to control the line length. Location 57500 should hold a lower value than 23679.

If you use the Kempston COPY: REM... system to change or check the interface status, or use the ZX LPRINT III method of selecting a graphics printer, or use COPY, Beta Basic will lose control of the "p" channel. This is not a problem, but you MUST enter: FORMAT "p";2 (which regains control) after doing this! If you do not, the printer buffer will be corrupted by the interface when you use the printer, and the computer will crash. You can reduce the inconvenience of this by setting up the interface characteristics by another method. When, the interface is configured (by the normal method) as you want it, PRINT PEEK 23658, PEEK 23728 and PEEK 23729. (These are the interface status variables. Location 23658 also sets the capitals/lower case mode of the computer.) Then include POKEs of these locations with those values in Beta Basic's line 2 loader, before the DELETE statement. Finally, RUN the line 1 SAVE routine to save the modified copy of Beta Basic.