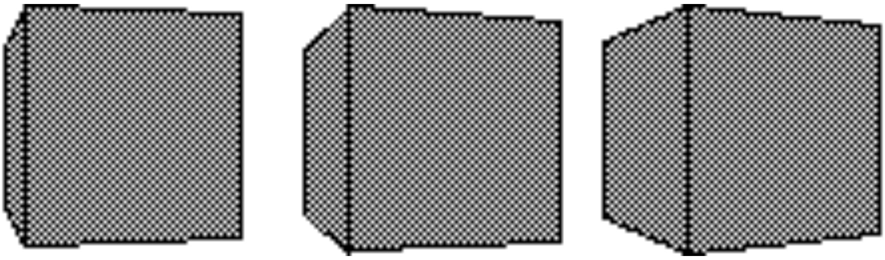
BETA BASIC NEWSLETTER No. 12



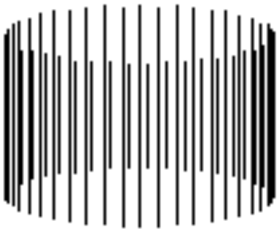
****
DRAWING 3-D CUBES

Back in issue 7 I provided an animation of a rotating ring, and said that I had really wanted to animate a cube, but failed. Some time after that, I got an excellent and well-documented contribution from Francesco Stajano (Rome, Italy) that drew complete "skeleton" cubes and animated them. Francesco asked if I could manage "hidden line" removal, which would help the illusion of depth. Another possible improvement would be to use perspective, so that parallel lines would converge into the distance. (Francesco used an "isometric" projection which keeps parallel lines parallel on the screen.) This prompted me to try to reconstruct my thought processes of 1983, when I had written such a cube-drawing program. (I had lost the tape and hadn't made a listing - no printer!) I had already failed at this while working on issue 7, but eventually I realised that the problem was that I knew more than I used to! I knew that such things are supposed to involve multiplying matrices and strange coordinate systems, which I had never heard of in 1983. I had used a less sophisticated method.

My reconstructed reasoning went something like this: Draw a cube - too hard! OK, draw one face of a cube - still too hard! OK, draw one upright of a cube's skeleton - that's just a vertical line. Now imagine that it is part of a cube rotating around a vertical axis. How should the position of the line on the screen vary as the angle of the cube changes? Suppose the cube is rotating anti-clockwise (seen from above) and the line starts in the middle of the screen, on the cube edge closest to us. Initially, the line should move quickly towards the right, because most of its motion is across our line of sight. As the line moves towards the right of the screen, more and more of its motion should be "into" the screen, and its rightward movement will slow and eventually reach zero. If the cube rotates clockwise, the pattern is similar but mirror-imaged.

So, successive views of the line need to be widely spaced in the centre of the screen, and closer together towards the sides. The SINE function fits this requirement; it changes rapidly for small angles, but changes more and more slowly as plus or minus 9111 degrees is approached. If you draw a vertical line at an x coordinate which varies according to the sine of the cube's angle (and also with the width of the cube) you get the right effect. However, the line should also be varying in height according to how fast it is approaching or receding. This will also be a SINE-type variation, but 90 degrees out of phase with the x coordinate change; the height will change fastest when the line is coming directly "out of" the screen, when the x position change is slowest. COSE gives the same as SINE(angle+90 degrees) so we can use that. The height of the line will not be zero, even when it is farthest away from us, so we need a constant, plus a varying component. The closer the cube is to our eyes, the larger the varying component will be. The program fragment below is for educational purposes only; it produces the output illustrated. This could be thought of as successive views of one upright of a cuboid. The variable q is related to nearness of the object.

```
10 LET wd=50,ht=60,q=0.18
20 FOR a=0 TO 2*PI STEP .14
30   PLOT 100+wd*SINE(a),60-ht*q*COSE(a)
40   DRAW 0,ht*(1+q*2*COSE(a))
50 NEXT a
```



Having got this far, it isn't hard to generate a cube face from two uprights with lines to join them. The second upright simply needs to be drawn 90 degrees (PI/2 radians) further round the cube. The procedure FACE does this, and al so incorporates another SINE-type term to make, horizontal lines on the rear of the cube shorter. A skeleton cube can now be produced by calling FACE four times with angles 90 degrees apart. In the example below, though, I have chosen to show just 2 faces to give a solid look to the cube. <The IF statement in line 100 is required in order that the first 2 animation frames show only a single cube face, as required in this case. This is a bit inelegant, as it varies according to how many frames you use.) For a skeleton cube, call FACE twice more with parameters of a+0.25*PI and a+0.75*PI. Lines 10, 251 and 252 are needed only if you are using BB 4.0 and want to FILL the cube faces. (As illustrated on the front cover.) At this stage, if you just want to view the cubes, you can omit lines related to animation if you like; these are 20, 30, 50, 120, 130 and 170. Animation is discussed in the next section.

```
 10 LET p$=STRING$(16,CHR$ BIN 10101010+CHR$ BIN 01010101)
 20 pokecd
 30 LET wx=88,wy=111,w=10,h=66
 40 LET frames=16
 50 DIM w$(frames,w*h)
 60 LET fx=128,fy=56
 70 LET wd=36,ht=48,q=.15
 80 LET f=1,st=PI/2/frames
 90 FOR a=0 TO PI/2-.01 STEP st
100   IF f>2 THEN face a+1.25*PI
110   face a+1.75*PI
120   nget a$,wx,wy,w,h
130   LET w$(f)=a$
140   CLS
150   LET f=f+1
160 NEXT a
170 anim w$,wx,wy,w,h
```

```
200 DEF PROC face a
210   PLOT fx+wd*SINE(a)*(1+q*COSE(a)),fy-ht*q*COSE(a)
220   DRAW 0,ht*(1+2*q*COSE(a))
230   DRAW TO fx+wd*SINE(a+(PI/2))*(1+q*COSE(a+(PI/2))),fy
      +ht*(1+q*COSE(a+(PI/2)))
240   DRAW 0,-ht*(1+2*q*COSE(a+(PI/2)))
250   DRAW TO fx+wd*SINE(a)*(1+q*COSE(a)),fy-ht*q*COSE(a)
251   LET xx=SINE(a+(PI/2))*wd+fx,yy=fy+ht*(1+q*COSE(a+(PI
      /2)))
252   FILL USING p$;xx-4,yy-15
260 END PROC
```

****
IMPROVED ANIMATION

Having managed to draw some nice cubes, I at first tried the animation routine from issue 7, but found some disadvantages. The screen mapping means that POKE loads data like LOAD "" SCREEN$, with lines scrambled; this makes vertical lines look momentarily ragged. Also, cubes do not fit conveniently into screen "thirds" and so waste memory, and 1, wanted to maximize the number of animation frames. Next I tried GET and PLOT (string), which are more flexible, but too slow. Basically this is because PLOT (string) works through a general-purpose printing routine that will handle pixel positioning and any CSIZE. A specialised routine would be simpler and much faster.

I wrote such a routine in assembly language; it handles copying of screen information into strings, and vice versa, in only 28 bytes. The procedures NGET and PUT modify the routine to suit the required direction of movement, and specify the top left-hand corner of the screen area (in pixel coordinates), its width (in characters) and length (in pixels). To keep things simple, no checking for valid values is done - add this to the procedures if you like. The machine code is poked into the UDG area, but could be placed elsewhere. As written, NGET and PUT need the code location to be in the variable ad, as set up by the POKECD procedure. NGET places the specified part of the screen memory into a designated string; PUT returns such an area to the screen. Note the similarity of the two procedures.

```
300 DEF PROC nget REF a$,x,y,w,h
310   DIM a$(w*h)
320   DPOKE ad+1,LENGTH (0,"a$")
      POKE ad+4,x
      POKE ad+5,y
      POKE ad+10,h
      POKE ad+13,w
330   POKE ad+16,0
      POKE ad+19,0
340   RANDOMIZE USR ad
    END PROC

400 DEF PROC put a$,x,y,w,h
420   DPOKE ad+1,LENGTH(0,"a$")
      POKE ad+4,x
      POKE ad+5,y
      POKE ad+10,h
      POKE ad+13,w
430   POKE ad+16,235
      POKE ad+19,235
440   RANDOMIZE USR ad
    END PROC
```

The procedure ANIM takes a whole array of strings and animates them, assuming that each string is a complete animation frame. ANIM needs to know where to put the animation - and the dimensions of the animated area. You can also specify the speed of animation with the final optional parameter. Animation continues until you press a key. To maximize the speed, the PUT procedure is called just once; this will alter the machine code suitably. After that, it is only necessary to vary the location of the source data to be put on the screen, as each frame is used. When used with the 16-frame cube example, an impressively smooth 24 frames per second are produced. (One advantage of a cube is that it has 4-fold symmetry - you only have to store views of 90 degrees of rotation, and show them 4 times, to imitate a full rotation. The ring in issue 7 was even better - it had 32-fold symmetry!)

```
500 DEF PROC anim REF f$,x,y,w,h,t
510   DEFAULT t=1
520   put f$(1),x,y,w,h
530   LET f=LENGTH(1,"f$")-1
540   LET e=LENGTH(2,"f$")
550   FOR n=1 TO 0
      NEXT n
560   LET b=LENGTH(0,"f$")
570   DO UNTIL INKEY$<>""
580     FOR n=0 TO f
590       DPOKE ad+1,b+n*e
600       RANDOMIZE USR ad
610       PAUSE t
620     NEXT n
630   LOOP
640 END PROC
700 DEF PROC pokecd
710   LET ad=USR "a"
720   IF PEEK ad<>17 THEN
        RESTORE 730
        FOR n=ad TO ad+27
          READ a
          POKE n,a
        NEXT n
730   DATA 17,0,0,1,0,0,205,170,34,6,0,197,1,0,0,229,0,237,
      176,0,225,205,0,232,193,16,240,201
740 END PROC
```

Here is the assembly-language version for those who may be interested - it should fill out the rest of this page, too!

```
        LD DE,string addr  ;POKED
        LD BC,coordinates  ;POKED
        CALL PIXEL-ADDRESS ;Call ROM to get screen address in HL
        LD B,length        ;POKED
NXSCAN: PUSH BC
        LD BC,width        ;POKED
        PUSH HL            ;save screen address
        NOP or EX DE,HL    ;POKED to suit direction of transfer
        LDIR               ;Copy BC bytes from HL to DE
        NOP or EX DE,HL    ;POKED to suit direction of transfer
        POP HL             ;restore screen address
        CALL NEXT-DOWN     ;Call BB to modify HL for next scan
        POP BC             ;fetch length count in B
        DJNZ NXSCAN        ;decrement B and LOOP WHILE B<>0
        RET
```

****
PROC ARROW - drawing arrowed lines

The next two procedures are based on ideas from Robert Dickson (London). I have re-written them to save space. PROC arrow is like DRAW TO but the line finishes with an arrow-head; the size of this can be altered by the third parameter of the procedure. Lines 130 and 140 find the x and y movement involved in drawing the lines, then the bit of nastiness in line 150 uses that data to calculate the angle of the line. The two arrow "barbs" can then be drawn at a suitable inclination to this line. The procedure would be useful in annotating diagrams.

```
 10 arrow 100,100
 20 PLOT 255,0
    arrow 105,100
 30 PLOT 128,170
    arrow 102,105

100 DEF PROC arrow x,y,hs
110   DEFAULT hs=10
120   LOCAL xd,yd,an
130   LET xd=x-PEEK 23677+xos
140   LET yd=y-PEEK 23678+yos
150   LET an=PI*(xd<0)+PI/2-ATN (yd/(xd+.001))
160   DRAW TO x,y
170   PLOT x-hs*SINE(an+.5),y-hs*COSE(an+.5)
180   DRAW TO x,y
190   PLOT x-hs*SINE(an-.5),y-hs*COSE(an-.5)
200   DRAW TO x,y
210 END PROC
```

****
PROC DOTDRAW - drawing dotted lines

This procedure is like DRAW TO, but the lines are drawn dotted. The third parameter determines how widely spaced the dots are; 1 gives a solid line, 4 gives fairly widely-spaced dots. The method used is to calculate the total x and y distances to be moved, and then add a part of these distances at each of a suitable number of PLOT steps. The most obvious application is to enable different lines to be told apart in diagrams, without the problems of using colour (such as attribute clash and difficulties with printer dumping).

```
 10 DO
 20   dotdraw RNDM(255),RNDM(175),3
 30 LOOP
100 DEF PROC dotdraw p,q,s
110   DEFAULT s=2
120   LOCAL x,y,xd,yd,d,f,a,b,c
130   LET x=PEEK 23677,y=PEEK 23678
140   LET xd=p-x,yd=q-y
150   LET d=SQR (xd*xd+yd*yd)
160   LET c=INT (d+.5)
170   LET a=s*xd/d,b=s*yd/d
180   FOR f=1 TO c-s STEP s
190     LET x=x+a,y=y+b
200     PLOT x,y
210   NEXT f
220   PLOT p,q
230 END PROC
```

****
'PLAY' TO 'BEEP" CONVERTER PROGRAM

Reader Martin Cleaver (Cuffley, Herts.) has sent me a very professional program that converts PLAY commands into BB 4.0's interrupt-driven BEEP! commands. Most PLAY options are covered. It is unfortunately too long to publish here, but I can send a listing or a tape copy (if you send a tape) to anyone interested. Stamps or postal reply coupons to cover return postage would be appreciated.

Martin's program threw up (yech!) some interesting points. An example he enclosed did not play all 3 channels exactly simultaneously, because each channel was set going by a large BEEP! command on a separate line. By the time the last BEEP! was encountered, the first channel was already playing. The solution in cases like this is to precede the main BEEP! lines with e.g.:

```
     9000 BEEP !0,30!0,30!0,30
```

This will set each channel to give a 0.6 second pause, allowing time for the rest of the BEEP! statements to be processed and put in the interrupt queue before any sound is produced.

Another problem was that KEYIN of the converted program lines sometimes failed for no obvious reason. I eventually discovered that a preceding EDIT (numeric variable) command left part of the computer's workspace untidy due to use of the ROM STR$ routine. (This particular routine seems to cause lots of problems!) Later on, the untidy workspace confuses KEYIN. The solution is to specifically clear the workspace; this can be done using RANDOMIZE USR 5808 before the KEYIN statement.

****
USING USING

I couldn't resist that title! Subscriber Robert Dickson (London) pointed out that Beta Basic's PRINT USING can look slightly inelegant when used with numbers less than 0.1. For example, if you use these lines:

```
     10 FOR n=0 TO .12 STEP .02
     20 PRINT USING "#.##";n;" ";
     30 NEXT n
```

```
You get:      0.00 .02 .04 .06 .08 0.10 0.12
```

This happens because BB uses the ROM print-a-number routine as a first step in producing a formatted string, and this routine does not use a leading zero with some values. The easy solution is to change the USING string to "0.##" (or "#0.##" or "##0.##" etc.) which will provide the missing zero.

Unfortunately, this method does not work with negative numbers in the same range, because the minus sign replaces the leading zero. You could try something like this for line 20:

```
     20 PRINT "-" AND n<0;" " AND n>=0; USING "0.##";ABS n
```

If n is negative, a leading minus is printed, otherwise a space 1S printed; the formatted number follows. A possible disadvantage is that the minus sign is always in the first column, whatever the form of the USING string.

****

SUPER-FAST DRAW AND PLOT

The fundamental idea of this section suggested itself just after I had written BB 4.0's faster DRAW command. This is up to 2.5 times faster than the normal DRAW, but in most real applications the speed-up is much more modest, because the speed of the program is limited by other factors, such as executing loops, or calculating the next line to draw. However, if a whole sequence of drawing instructions in a convenient form (for a computer) were to be given as data to a specialised graphics command, a very large increase in speed could be expected. (And in fact, I had already done program conversions on games that used something similar in special "scene description languages".) I considered adding a DRAW (string) command to BB 4.0, with the graphics instructions coded somehow into the string, but never got round to it. More recently, I've incorporated some of these ideas into the following set of procedures. They will work with either BB 3.0 or BB 4.0, and handle PLOT, DRAW and DRAW TO. The actual speed-up compared to normal Basic doing the same thing will depend very much on what you are doing; the examples given here will be speeded up 2 or 3 times. If you do lots of PLOTs or short DRAWs the speed increase could be much greater.

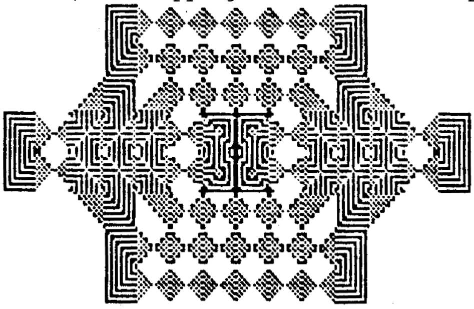
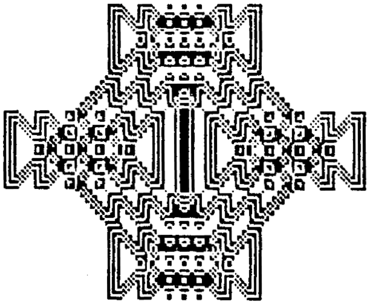
Five procedures are given below. POKECD sets up the required machine code in the UDG area, but the code could be placed elsewhere if references to USR "a" (the start of the UDG area) are altered. The DATA statements should add up to 8388. I have not provided a formal way to do this, but something simple like:
LET t=0: DO: READ a: LET t=t+a: LOOP followed by PRINT t when you run out of data, will work satisfactorily. The procedure automatically modifies the code if it decides that BB 4.0 is resident, by assuming that version number (PEEK 47272) will always be less than 10 for BB 4.0 (I will ensure that this is true!) and always 10 or more for BB 3.0. (If you have an earlier version of BB 3.0, please return it for replacement.)

NPLOT, NDRAW and NDRAWTO provide equivalents of the normal graphics commands. They add characters to the string A$ (which normally starts as a null string) to code for the graphic command requested, and also perform the actual operation, so that you can see what is being coded into the string. PLOT and DRAW TO are easy to code - CHR$ 1 and CHR$ 2 code for the commands, followed by the x and y coordinates. DRAW can involve negative numbers and is slightly more complex. The method chosen makes the job of the machine code easier. An advantage of using CHR$ 0 or CHR$ 255 to give the sign of the DRAW parameter is that we can avoid the need for a specific code for DRAW itself; anything starting with either character is a DRAW command.

The procedure SDRAW (for String DRAW) takes one of these coded strings as a parameter and executes the desired sequence of PLOT and DRAW commands much faster than would otherwise be possible.

Obviously there is no point in using these procedures if you only want to draw something once, but after the string has been created, it can be used many times to give fascinating fast-moving displays. SDRAW can be preceded by INK, PAPER or OVER commands to give variety. The position of the patterns can be altered by changing XOS and YOS; however, XRG and YRG have no effect, in order to keep the speed as high as possible.

Lines 10-180 are demo lines which create patterns such as those illustrated below. (If you can't be bothered to type in the procedures, you can still create these patterns, much more slowly, by altering all the NPLOT and NDRAW commands in lines 50 and 60 to PLOT and DRAW, and copying lines 40-70 to replace line 140.)

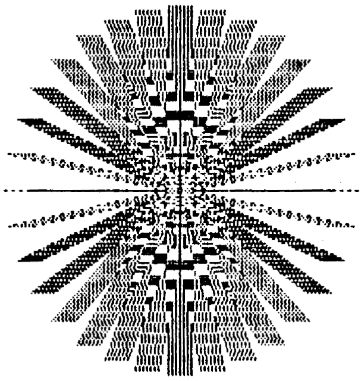
```
 10 OVER 1
 20 pokecd
 30 LET a$=""
 40 FOR n=74 TO 140 STEP 2
 50   nplot n,n
 60   ndraw 255-n*2,0
      ndraw 0,175-n*2
      ndraw -255+n*2,0
      ndraw 0,-175+n*2-SGN (-174+n*2)
 70 NEXT n
 80 CLS
 90 DO
100   FOR f=0 TO 24
110     LET xos=-f*4
120     FOR n=0 TO 6
130       LET xos=xos+f
140       sdraw a$
150     NEXT n
160     PAUSE 150
        CLS
170   NEXT f
180 LOOP

400 DEF PROC sdraw a$
      DPOKE USR "a"+1,LENGTH(0,"a$")
      DRAW 0,0
      LET z=USR USR "a"
    END PROC

410 DEF PROC ndraw x,y
      DRAW x,y
      IF x<-.5 THEN LET t$=CHR$ 255+CHR$ (256+x)
      ELSE LET t$=CHR$ 0+CHR$ x
420   IF y<-.5 THEN LET t$=t$+CHR$ 255+CHR$ (256+y)
      ELSE LET t$=t$+CHR$ 0+CHR$ y
430   JOIN t$ TO a$
    END PROC

440 DEF PROC nplot x,y
      PLOT x,y
      LET t$=CHR$ 1+CHR$ x+CHR$ y
      JOIN t$ TO a$
    END PROC
```

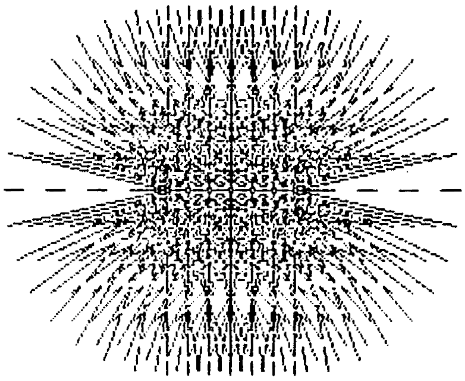
```
450 DEF PROC ndrawto x,y
      DRAW TO x,y
      LET t$=CHR$ 2+CHR$ x+CHR$ y
      JOIN t$ TO a$
    END PROC

500 DEF PROC pokecd
510   IF PEEK USR "a"<>33 THEN
        RESTORE 520
        FOR n=USR "a" TO USR "a"+72
        READ a
        POKE n,a
      NEXT n
520 DATA 33,0,0,126,95,35,86,35,70,35,60,254,2,56,29,229,2
     45,175,95,75,205,187,42,80,205,187,42,241,32,5,205
530 DATA 79,220,24,29,254,3,32,28,205
540 DATA 28,189,24,20,126,35,229,245,175,75,205,187,42,241
     ,88,75,87,175,205,187,42,205
550  DATA 183,36,225,24,192,225,33,88,39,217,201
560  IF PEEK 47272<10 THEN
       DPOKE USR "a"+31,63096
       POKE USR "a"+40,164
       DPOKE USR "a"+62,48610
570 END PROC
```

A completely different and rather pretty effect can be obtained by altering lines 40 to 70 to:

```
40 FOR a=0 TO 2*PI-0.01 STEP PI/16
50 nplot 128,87
60 ndraw 86*SINE(a),86*COSE(a)
70 NEXT a
```

Try altering the 16 in line 40 to other values.





****
PRINTERS - two types of output.

From time to time in this publication I have referred to sending output to a printer via "t" or "b" type channels. This should make sense to Interface 1 or Opus Discovery users, but for those unfamiliar with the idea I should have explained it. With an Interface 1, "t" and "b" channels send output to the RS232 port; on the Opus Discovery, they send output to the parallel port. The distinction between "t" and "b" is not related to the physical port, but to how the channel treats printed characters. Interfaces which do not use the terms "t" and "b" will generally provide something equivalent.

"B" stands for "binary" and any characters (from CHR$ 0 to CHR$ 255) printed to a "b" channel should be sent unchanged. This is what we want for sending control codes or bit-image data for printer dumps. We do not want keyword expansion, automatic pagination, extra line feeds or line length control, all of which could cause havoc. Provided our printer does a line feed after receiving a carriage return, "b" type channels are usable for sending normal text like "Hello World", though "t" would be more common. Where a "t" (for "Text") channel is really needed is in LLISTing. This is because Basic programs are stored in tokenised form; for example, COPY is stored as CHRS 255. To make a listing look sensible, a "t" channel will convert CHRS 255 to the characters (space), C, 0, P and Y and send them to the printer. This would be disastrous in a screen dump, where CHR$ 255 (binary 11111111) would represent a vertical line, to be sent unchanged. A "t" channel will also "filter out" Spectrum Basic colour control codes, such as INK, PAPER or INVERSE, as they will not be understood by the printer. TAB, AT and the print comma may be translated into a suitable sequence of spaces, depending on the interface. Carriage return may be translated into carriage return/line feed, perhaps as an option.

When an interface allows streams to be OPENed to "b" or "t" channels, switching between the two should be easy. However, many systems instead alter the "p" channel, which is the original ZX Printer channel. (This was "t" type in the way it handled characters.) An advantage with this method is that OPEN is not used; just LPRINT or LLIST and the printer should work. The 128K versions of the Spectrum make such an alteration, diverting LPRINT and LLIST to a "t" type RS232 channel. (This is flawed; any colour control codes are not handled correctly.) EPROM-based interfaces usually detect use of the old ZX printer code via hardware, and then alter the "p" channel.

Beta Basic needs control of the "p" channel if new keywords are to be LLISTed correctly through the channel, but this could cause problems if you want to send to control codes or printer dumps by the same route. Most interfaces provide some means of selecting "t" or "b" type output, whatever they call them. You may make a special POKE, or print a certain character, that tells your interface not to print keywords. Beta Basic will not understand such methods, and will still print e.g. CHR$ 130 as (space), P, R, 0, C. In other words, you are stuck with "t" type output. For this reason you might want to take control of the "p" channel away from Beta Basic. If you use: DPOKE DPEEK(23631)+15,2548 then the "p" channel will try to use the ZX Printer routine, which will cause some EPROM-based interfaces to re-initialise without BB in the way. You can use PRINT DPEEK(61081) to find the address BB is sending printer output to after expanding keywords; you might find this value is better than 2548. To restore "p" channel control to BB, DPOKE with 64423, instead of 2548.

****
PROC SEARCH - searching memory

This contribution is from Iain Rendall (Edinburqh), apart from lines 100 to 170, which I include as an alternative method of using the procedure. Iain writes:

"PROC search... comes up with all occurrences in a given area of memory of a given pattern of bytes. By default I have it searching the ROM. Simple but I have found it useful as INSTRING is so rapid. All occurrences give the decimal and hex addresses."

```
  10 DO
  20   INPUT "Number of search bytes? ";n
  30   FOR k=1 TO n
  40     INPUT "Byte "+STR$ k+"? ";x
         POKE USR "a"+k-1,x
  50   NEXT k
  60   CLS
       LET s$=MEMORY$()(USR "a" TO USR "a"+k-2)
  70   search s$
  80 LOOP

 100 DO
 110   LET s$=""
       INPUT a$
 120   FOR n=1 TO LEN a$ STEP 2
 130     LET s$=s$+CHR$ (DEC(a$(n TO n+1)))
 140   NEXT n
 150   CLS
 160   search s$
 170 LOOP

1000 DEF PROC search s$,start,final
1010   DEFAULT start=1,final=16384
1020   DO
1040     LET p=INSTRING(1,MEMORY$()(start TO final),s$),q=start+p-1
1050     EXIT IF p=0
1060     PRINT q,HEX$(q)
         LET start=q+1
1070   LOOP
1080 END PROC
```

The alternative calling routine at line 100 might be more suitable for hexadecimal-oriented users. It allows you to enter e.g. "CD0116". (= CALL 1601H = CALL Channel-Open.) I used something similar very frequently while I was writing BB. (Note: The "£" (accept anything) character is 23 in hex.)

****
MASTERFILE ADD-ONS FOR THE PLUS THREE

Regular contributor Robert Dickson (12 Coppelia Road, Blackheath, London SE3 9DB) has converted Masterfile (48/128) so that it will work with the Plus 3's printer interface, avoiding the need to buy Masterfile Plus Three. He will sell you the converter program for £3.00 if you provide a disc. Robert also sells a program to convert Masterfile data to Tasword +3 format; the cost is £2.50. You supply the disc. He has a 48K Spectrum, Interface 1, Microdrives, books etc. for sale - write to him about that. No reasonable offer refused, he says! Anyone know where he can get an ON/OFF switch for the Plus 3?

****
PROC PMENU - deluxe pull-down menu
PROC KTM - pausing a program and giving a message

These two procedures are from Martin Cleaver (Cuffley. Herts.). We have seen menu control programs before, and I have several unpublished contributions on the theme. It is clear that many users are interested in the subject. This version incorporates some new ideas, and is well explained by the author. (An advantage, in something of this length.) Now, over to Martin:

```
  Syntax:
  PMENU choice <,dataline><,max len><,no of options><,row><,col>
```

This calls the procedure and if no dataline is given it endeavours to calculate from which line the procedure was called. This is done by looking on the GOSUB/PROC/DO-loop stack for the address of the calling line and setting dataline to this value. Of course, this is not possible if the procedure was called as a direct command and in this case, default for dataline is 1.

The line addressed by variable *dataline* is RESTOREd and if either the maximum width or the number of options are not given then the program reads through the data calculating these until it finds the terminator CHR$ 255. (I have used this system of using a terminator of CHR$ 255 instead of the more conventional way of checking that ITEM()=0 so as to allow multiple entries into data statements. e.g.:

```
110 DATA "circle",square,pentagon,hexagon
120 DATA "change ink colour","change paper colour",CHR$ 255
```

If a dataline of 120 was used then only those items in line 120 would be used, but if 110 was used instead then not only would the items in line 110 be used, but those of 120 as well.) [BB allows you to omit quotes from strings in data statements if you use READ LINE. except where keywords would be created. Pity I didn't suppress "tokenization" in DATA statements, as I did for REMs! Ed.] String variable d$ is then dimensioned as the length of the longest data item (max) and then the dataline is restored again. This time the program goes through the data items again and shoves the whole lot in variable a$ together with a number or letter corresponding later to that option. (The first letter is capitalised and extra quotes removed with VAL$.)

Next, the window coords are calculated and window 10 defined to cover them. Then the area underneath the window is stored in p$ and n$ (pixel data and attributes respectively). The screen is then dulled (i.e. any bright signals removed) and the window opened together with a "frame" to give some impression of depth. This is then filled with the options (a$).

At this point a loop is programmed to accept keypresses to move the colour bar to the required option. This can be done in 2 ways; use up/down cursors or press the letter/number next to the required option. The bar will then move to that particular option, any other keys (except enter) will move the bar until it reaches the bottom when it will return to the top of the menu. Pressing enter will choose the option pointed by the bar, this option number will be passed back through by the REF. The screen is then restored to what it was before the procedure was called.

Back to The Editor again: I have modified this procedure at line 3040, because Martin's method of finding the address of the line that called PMENU was slightly flawed; it only worked if the Basic program started at a particular place. I have said something about how the procedure stack works before (issue 5, p.14) but I've never explained the format of its contents. Now I'd better! For speed, it is desirable to store the address of the calling line, not its actual line number; however, this address might change if a new channel or Microdrive buffer is created. Therefore, BB stacks the *difference* between PROG (the start of the Basic program - system variable 23635) and the line's address. The value 16384 is added to this to help show that the value is a procedure return address. Line 3040 reverses this process to obtain the real line address from the value on the top of the stack. Line 3060 can then read the line number from the Basic program. ("Underneath" each coded line address (e.g. at DPEEK(23613)+3) is a return statement number. These are one greater than the statement number of the procedure call.)

My monitor doesn't show BRIGHT, and I thought that storing the entire attribute file (768 bytes) in n$ so that the screen could be "dulled" and then restored was quite a high memory overhead. So for my own use, I wanted to remove this feature. This can be done by deleting lines 3355, 3365 and 3605, and adding ";1" to the end of line 3350 so that the GET string will contain attributes as well as pixel data. I added line 1 before doing my screen dump in order to show the "pull-down" nature of the PROC. Now, over to Martin again:

PROC KTM (Key Terminated Message) is an improvement of PROC KEY (issue 7). It accepts an optional (non-compulsory) message after the calling command, e.g.:

```
KTM or
KTM "Press any key to load"
```

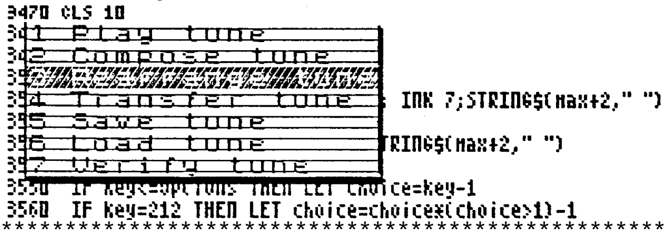
If no message is supplied then the default is "Press any key to continue". It uses an "alternative" way of making B$ exactly 32 characters long, as opposed to using DIM. INPUT "" clears the lower screen.

```
   1 CSIZE 4,8
     LIST 3469
  10 Pmenu c
     DATA "press any key to continue",end,CHR$ 255
  15 DATA "play tune",compose tune,rearrange tune,transfer tu
     ne,"save tune","load tune","verify tune",CHR$ 255
  20 KTM
  30 Pmenu c,15
  40 ON c
       KTM "play tune"
       KTM "compose"
       KTM
       KTM "transfer tune.Are you sure?"
       KTM "any key to save tune (a)bort"

1000 DEF PROC KTM B$
       DEFAULT B$="PRESS ANY KEY TO CONTINUE"
       LET B$=SHIFT$(1,(B$+STRING$(ABS (32-LEN B$)," "))( TO
       32))
       DO
         PRINT #0;AT 1,0; PAPER 6; INK 9;B$
         LET B$=B$(32)+B$( TO 31)
       LOOP UNTIL INKEY$<>""
       INPUT ""
     END PROC
```

```
3000 DEF PROC Pmenu REF choice,dataline,options,max,row,col
3010   LOCAL key,dep,number,wid,x,y,n,b$,d$,n$,p$,a$
3020   DEFAULT row=0,col=0,dataline=-1,max=0,options=0
3030   IF dataline>0 AND dataline<10000 THEN GO TO 3070
3040   LET n=DPEEK(DPEEK(23613)+2)+DPEEK(23635)-16384
3050   IF n>65535 THEN LET dataline=1
         GO TO 3070
3060   LET dataline=PEEK (n)*256+PEEK (n+1)
3070   RESTORE dataline
3080   LET a$="",number=1,n$="0"
3090   IF max<>0 AND options<>0 THEN STOP
         GO TO 3170
3100   DO
3110     READ LINE b$
3120     IF b$(1)=CHR$ 34 THEN LET b$=VAL$ b$
3130   EXIT IF INSTRING(1,b$,"CHR$ 255")
3140     IF LEN b$>max THEN LET max=LEN b$
3150     LET options=options+1
3160   LOOP
3170   LET col=INT (col/8)*8
3180   IF options=0 OR max>28 THEN STOP
         GO TO 3620
3190   DIM d$(max)
3200   RESTORE dataline
3210   DO
3220     READ LINE b$
3230   EXIT IF INSTRING(1,b$,"CHR$ 255")
3240     IF b$(1)=CHR$ 34 THEN LET b$=VAL$ b$
3250     LET b$(1)=SHIFT$(1,b$(1))
3260     LET d$()=b$
3270     IF number>9 THEN LET n$=CHR$ (55+number)
         ELSE LET n$=STR$ number
3280     LET a$=a$+n$+" "+d$+CHR$ 13
3290     LET number=number+1
3300   LOOP
3310   DEFAULT row=0,col=INT ((31-max)/16)*8
3320   LET x=col+8,y=167-(row*8)
       LET wid=max*8+16,dep=options*8
3330   IF (y-dep)<0 OR x+wid>255 THEN STOP
         GO TO 3610
3340   WINDOW 10;x,y,wid,dep
3350   GET p$,x-1,y,max+3,options+1
3355   LET n$=MEMORY$() (22528 TO 23296)
3360   WINDOW 10
3365   ALTER TO BRIGHT 0
3370   INK 0
3380   LET x=x-1
3390   PLOT x,y
       DRAW 0,-dep
3400   DRAW wid,0
       PLOT x+1,y-dep-1
3410   DRAW wid,0
       DRAW 0,dep
3420   PLOT x+2,y-dep-2
       DRAW wid,0
       DRAW 0,dep
3430   PAPER 5
3440   BRIGHT 1
3450   CSIZE 8
3460   OVER 2
3470   CLS 10
```

```
     3480   PRINT a$( TO LEN a$-1)
     3490   LET choice=1
     3500   DO
     3510     PRINT AT choice-1,0; PAPER 2; INK 7;STRING$(max+2," ")
     3520     GET key
     3530     PRINT OVER 1;AT choice-1,0;STRING$(max+2," ")
     3540   EXIT IF key=214
     3550     IF key<=options THEN LET choice=key-1
     3560     IF key=212 THEN LET choice=choice*(choice>1)-1
              ELSE
                LET choice=choice*(choice<options)+1
     3570     IF choice=-1 OR key=0 THEN LET choice=1
     3580   LOOP
     3590   BRIGHT 0
     3600   PLOT OVER 0;x,y;p$
     3605   POKE 22528,n$
     3610   WINDOW 0
     3620 END PROC
```

The Editor again: The use of STOP followed by GO TO when some problem is found, as used above, allows a simple CONTINUE to make the program GOTO somewhere useful, like a line to give WINDOW 0: END PROC. The dump below uses shading for colour.



```
3470 CLS 10
3 1 Play tune
3 2 Compose tune
3 [illegible]
3 4 Transfer tune ; INK 7;STRING$(max+2," ")
3 5 Save tune
3 6 Load tune       TRING$(max+2," ")
3 7 Verify tune
3550  IF key<=options THEN LET choice=key-1
3560  IF key=212 THEN LET choice=choice*(choice>1)-1
```

****
PROC PRINTAR - printing numeric arrays neatly

This contribution from Francesco Stajano (Rome, Italy) is used to neatly tabulate 1 or 2-dimensional numeric arrays, in a specified format. The default format prints each number with a decimal point; my example over-rides that, since all the numbers in my array a() are integers. Remember you can change CSIZE if the array is too wide for the screen. The procedure will work to a printer if you alter PRINT to LPRINT, or use PRINT #(stream) and vary the stream number.

```
  10 DIM a(12,6)
  20 FOR i=1 TO 12
       FOR j=1 TO 6
         LET a(i,j)=i*10+j
       NEXT j
     NEXT i
  30 printar a(),"####"

 100 DEF PROC printar REF q(),u$
       DEFAULT u$="##.#"
       LOCAL i,j
       FOR i=1 TO LENGTH(1,"q()")
         FOR j=1 TO LENGTH(2,"q()")
           PRINT USING u$=q(i,j);" ";
         NEXT j
         PRINT
       NEXT i
     END PROC
```

****

READERS' LETTERS

Dear Sir,

I must admit I subscribed to the BETA BASIC NEWSLETTER with the misconception that it would be full of useless rubbish. Boy was I wrong. The Newsletter is excellent. It's just the kind of thing I've been looking for. I saw in Newsletter 5 a routine to turn programs into Tasword files which I found fascinating. Is there a similar routine to reverse the process? I port Jupiter Ace software to the Spectrum and then store it on Microdrive. I would find it useful to be able to hold these programs as Tasword files but also to extract them from those files at a latter date.

Ian Jones, Pallion, Sunderland, Tyne & Wear

I think probably many people have the same idea of the Newsletter's contents as you used to have, but never had a chance to change their minds! It would be possible to convert Tasword files back into Basic programs, but it is probably easier to keep a copy in both program and document forms. FORTH programs would be harder, in any case, since I don't know anything about how such programs are stored.

Dear Andy,

...Why the hell is the display file so stupidly laid out? I can understand having lines of 32 bytes to make an horizontal line, which is "what the electron beam in the television needs", but why, then did they mess up the pixel lines?...

Francesco Stajano, Rome, Italy

As a programmer, I had always thought this was intended to simplify PRINT; since each pixel line in a character is 256 bytes below the previous one, you just need to increment the most significant byte of the screen address in machine code to find the location of the next place to put character data. If the next pixel line were 32 bytes lower instead, you would need to protect a register, load it with 32, add it to the address, and then restore the register, which is slower. But I learned recently from hardware designer Bruce Gordon that a more important advantage is that there is a simple relationship between each screen address and the corresponding attribute location. Both locations have to be read in quick succession by the display circuitry; this is a lot easier if the less significant byte of each screen address is the same as that of its related attribute.

****

-*-*-*- SUBSCRIPTIONS -*-*-*-

They are ALL due! So no more fascinating Newsletters unless you send me some money! To smooth out fluctuations in subscription income and encourage the hesitant, I have decided to change to 3-issue subscriptions. The price will be £3.00 (U.K.), £3.50 (Europe) or £4.00 (anywhere else - Air Mail). This covers some increased costs since issue 1. This is a good time to pass comment on the Newsletter's contents - if you want changes, please say so, and I will try to oblige. Also, PLEASE SEND IN YOUR CONTRIBUTIONS! All items will be gratefully received!

****