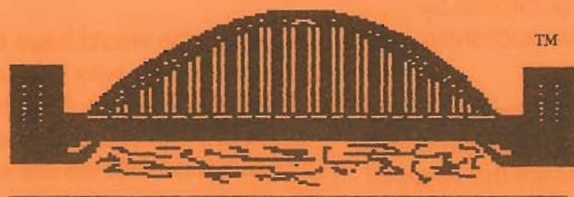
TYNE & WEAR



ATARI 8-BIT

USER GROUP

Issue 25

January/February 1997

Published by TWAUG

Edited & Printed by AMG

TWAUG NEWSLETTER

Publishing

TWAUG NEWSLETTER is published bi-monthly, around mid-month of (Jan, Mar, May, July, Sept and Nov.)

It is printed and published by TWAUG, no other publishing company is involved.

Opinion expressed by authors, in this newsletter, is their own opinion and do not represent the views of TWAUG.

The Atari Fuji symbol and Atari name are the trademarks of Atari Corporation. The Fuji symbol on the front cover , is for informational purpose only.

TWAUG is entirely independent and is in no way connected with Atari Corporation or any associate company.

Do you need to **Contact** anyone at TWAUG for a chat then phone

Alan Turnbull on: 01670 - 822492

or Max on: 0191 - 586 6795

Our Postal address:

TWAUG

c/o J.Matthewson

80 George Road

Wallsend, Tyne & Wear,

NE28 6BU

1997

Well who would have thought that T.W.A.U.G. was still going strong after four years. Yes this is the start of the fifth year for us and we are delighted that we are still publishing our mag. for our subscribers.

Well I wonder what the next four years has in store for us, it would be nice if we could just look into a crystal ball and see what the future will bring. But that would be cheating, wouldn't it?

We haven't had a lot of mail in the last year, but hopefully our readers will have made writing to TWAUG a priority in their New Years resolution.

Please make sure you address all your mail for TWAUG to the address shown in the opposite column.

Thank you for your help.

John's address

PUBLISHING!

This new look newsletter is set up with the Desktop Publishing program "TIMEWORKS 2", on the Mega 1 ST with 4 meg memory. Files are converted to ASCII and transferred to the ST with TARI-TALK. Those files are then imported into the DTP and printed with the Canon BJ-30 Bubble Jet Printer at 360 dpi, with excellent result.

TWAUG

TWAUG subscriptions

Home	..... 1 Copy	£2.50
- DO -	..... 6 Copies..	£12.50
Europe	..... 1 Copy	£2.50
- DO -	..... 6 Copies..	£13.50
Overseas...	1 Copy	£3.50
-- DO --	..... 6 Copies..	£16.00

Issue 26 is due mid-March 97

ISSUE CONTENT

REMINDER & NEWS 2
CONTRIBUTIONS & CONTENT 3
DON'T LET BASIC BUG YOU ....
Tutorial in Basic by Mike Bibby 4
GRAPHICS DISPLAY LIST ....
by Mike Rowe 11
GAMES REVIEW ....
by Kevin Cooke 19
BIT WISE ....
by Mike Bibby 24
NEW YEAR GREATINGS ....
from TWAUG to members 31
SURVEY & SALES 32
ARTISTIC IMPRESSION ....
WOT? NO MAIL? by Max 33
DISK CONTENT 34
USER GROUPS ADVERTS ....
for LACE & OHAUG 35
ADVERTISING.......................................
MICRO DISCOUNT 36

DON'T LET BASIC BUG YOU

Part VIII of MIKE BIBBY's guide through the micro jungle considers the question of nested loops

As I promised last issue, we've more **FOR...NEXT** loops this time, so if you're not too sure of them perhaps you'd better find a few minutes for a bit of quick revision.

Actually Program I should be fairly straightforward. All it does is to print out

A BLACK BOX

three times. The loop variable **COUNTER** keeps track of how many times lines 50 to 80 - the body of the loop, between the **FOR** and **NEXT** - are repeated. Since line 30 reads:

30 FOR COUNTER=1 TO 3

this will be three times. Notice that lines 50 and 60 have semi-colons to "glue" the words together. Line 70 omits this, though, as we want to move on to a new line:

```
10 REM PROGRAM I
20 PRINT CHR$(125)
30 FOR COUNTER=1 TO 3
50 PRINT "A";
60 PRINT " BLACK";
70 PRINT " BOX"
80 PRINT
90 NEXT COUNTER
```

Program I

So why the message "A black box"? Well, the idea is to stress that it doesn't really matter what's inside the "box" formed by the **FOR** and **NEXT**, it will be done as many times as is specified in the **FOR** Statement.

Solving the

Admittedly our knowledge of Basic isn't yet so encyclopaedic that we could think of many other things to go inside the box, but we can see the possibility.

The point is, given lines 30 and 90, whatever lies in the box between them will be done three times and you don't have to know what's inside the box to be aware of this. There are stupid exceptions to this which we'll meet, but they involve bad

DON'T LET BASIC BUG YOU continued

programming, which, of course, you won't be interested in...

```
10 REM PROGRAM II
20 PRINT CHR$(125)
50 FOR LOOP=1 TO 4
60 PRINT "DOING LOOP"
70 NEXT LOOP
```

Program II

Now take a look at Program II. Again, a simple loop. Nothing there to trouble you - it just prints out:

DOING LOOP

four times. Lines 50 to 70 form the chunk of program that prints this message out four times.

The only odd thing about this program - and Program I, come to that - is that our line numbers haven't gone up in consecutive tens. You'll see why in a minute.

secret of the black box

Returning to Program I, as I've stressed it doesn't matter what went inside the loop formed by lines 30 and 90 - it would be done three times.

So in a wheels within wheels manner, let's put a loop inside the loop of Program I. We'll take the loop of Program II -lines 50 to 70 - and put them in place of the lines that give the "A Black Box" message in Program I - also lines 50 to 70 (now you see one of the reasons for the line numbers).

```
10REM PROGRAM III
20 PRINT CHR$(125)
30 FOR COUNTER=1 TO 3
50 FOR LOOP=1 TO 4
60 PRINT "DOING LOOP"
70 NEXT LOOP
80 PRINT
90 NEXT COUNTER
```

Program III

Program III is the result. We now have two loops, one nested inside the other like those Russian dolls. In fact we call them nested loops. And you won't be surprised to learn that we call the loop that goes round the outside the outer loop, and the one on the inside the inner loop.

' Wheels within wheels

.... loops within loops '

Before you run it, see if you can think through what happens. Lines 30 to 90 ensure that we do the

DON'T LET BASIC BUG YOU continued

intermediate lines three times. Of these lines 50 to 90 form a loop printing out "**Doing loop**" four times, followed by a blank line (line 80).

So the outcome is that we get the message "**Doing loop**" 12 times in all, in three sets of four, each separated by a blank line.

To help you see what's going on more clearly, Program IV gives another version. I've changed the loop variable in lines 30 and 90 to *SET* to reflect the fact we're doing things in sets, and added:

40 PRINT "SET ";SET

to mark off each set. Note this line is inside the outer loop but outside the inner loop, so it only appears each time the outer loop is done. I've also altered line 60 sa that the variable *LOOP* is printed out as it cycles through its various values.

To get a feel for nested loops, try changing the limits of the loops in lines 30 and 50, predicting what you'll get *before* you run the altered program.

After your experiments restore the original Program IV, swap lines 70 and 90, then **RUN** the result. You should be able to work out what's going wrong. Remember, they're nested loops - the start and finish of the inner loop must fit neatly inside the start and finish of the outer.

Anyway, untangle yourself from this mess by swapping the lines back and changing line 50 to:

50 FOR LOOP=1 TO SET

then run it. You should get:

DOING LOOP 1

DOING LOOP 1
DOING LOOP 2

DOING LOOP 1
DOING LOOP 2
DOING LOOP 3

We're still doing the outer loop three times, so we still get three sets of output from the inner loop. Now though, because of the change to line 50, the number of times the inner loop is done varies, depending on the value of *SET*. That is, the number of times the inner loop's done depends on the value of the outer loop's variable!

In this case the longer in the tooth the outer loop is the more often the inner loop is done. The effect is that there's one more "**Doing loop**" in each successive set.

(As we've already seen, we refer to the loops as outer and inner. Some people like to use these words as

DON'T LET BASIC BUG YOU continued

labels for their loops to help them keep track. Program V reinterprets Program IV in this way. Personally, I prefer more meaningful labels - its up to you.)

```
10 REM PROGRAM IV
20 PRINT CHR$(125)
30 FOR SET=1 TO 3
40 PRINT "SET ";SET
50 FOR LOOP=1 TO 4
60 PRINT "DOING LOOP ";LOOP
70 NEXT LOOP
80 PRINT
90 NEXT SET
```

Program IV

Program VI uses the idea of making the number of times we do the inner loop dependent on the outer loops variable to print out a triangle of asterisks.

When deciphering what's going on with nested loops it's helpful to have a quick look at the line defining the outer loop - in this case line 30 - to get an idea of the range of its variable. Then concentrate on the inner loop - here lines 40 to 60.

The effect of this inner loop is to print out **LENGTH** number of asterisks on a line: Our inner loop goes from one to **LENGTH** and a semicolon follows the asterisk in the **PRINT** Statement of line 50, which forms the body of the loop. After printing the required number of asterisks, line 70 moves us on to the next line of the display.

So looked at as a black box, what's inside the outer loop (lines 40 to 70) simply prints out a separate line of **LENGTH** asterisks.

```
10 REM PROGRAM V
20 PRINT CHR$(125)
30 FOR OUTER=1 TO 3
40 PRINT "SET";OUTER
50 FOR INNER=1 TO 4
60 PRINT "DOING LOOP ";INNER
70 NEXT INNER
80 PRINT
90 NEXT OUTER
```

Program V

We repeat this outer loop 10 times, with the value of **LENGTH** varying from one to ten. So the first time round the outer loop we get one asterisk on a line, the second time two asterisks,

' The start and finish of the inner loop must fit neatly inside the start and finish of the outer '

DON'T LET BASIC BUG YOU continued

and so on.

I use a similar technique in Program VII to produce a triangle of letters. Here the outer loop variable, *FINISH,* ranges from one to *LEN-(STRING$).* Since *STRING$* is ABCDEFGHIJ, this boils down to our familiar from one to ten.

```
10 REM PROGRAM VI
20 PRINT CHR$(125)
30 FOR LENGTH=1 TO 10
40 FOR ASTERISK=1 TO LENGTH
50 PRINT "*";
60 NEXT ASTERISK
70 PRINT
80 NEXT LENGTH
```

Program VI

I've chosen *FINISH* as a label because its value determines where we end our printing of characters from *STRING$* in the inner loop.

The inner loop prints out successive characters from STRING$ by picking them out with:

70 PRINT STRING$(LETTER,LETTER);

as *LETTER* varies from one to *LENGTH.* Remember: *STRING$(1,1)* picks up the first letter of *STRING$, STRING$(2,2)* the second and so on.

The semicolon of line 70 ensures they all appear on the same line.

```
10 REM PROGRAM VII
20 PRINT CHR$(125)
30 DIM STRING$(10)
40 STRING$="ABCDEFGHIJ"
50 FOR FINISH=1 TO LEN(STIRNG$)
60 FOR LETTER=1 TO FINISH
70 PRINTSTRING$(LETTER,LET-
TER);
80 NEXT LETTER
90 PRINT
100 NEXT FINISH
```

Program VII

Once the inner loop is complete and the line finished, line 90 moves to a fresh line of the display.

The outer loop is then repeated, *FINISH* being increased by one, so that this time our inner loop will print out one extra character from *STRING$* and so on.

Actually we could accomplish all this with far less effort, as we saw from Program IV last issue. However it illustrates the techniques of nested loops quite well.

Now take a look at Program VIII. Before you start looking, it hasn't got nested loops - that will come later! The idea of the program is to add together all the whole numbers (integers) between one and a number you've input, then print out

DON'T LET BASIC BUG YOU continued

the result.

```
10 REM PROGRAM VIII
20 PRINT CHR$(125)
30 PRINT "Number ";
40 INPUT NUMBER
50 PRINT
60 TOTAL=0
70 FOR INTEGER=1 TO NUMBER
80 TOTAL=TOTAL+INTEGER
90 NEXT INTEGER
100 PRINT "Total is ";TOTAL
```

Program VIII

For instance, if you input 5, the program would do the sum 1+2+3+4+5 and print out the answer, 15.

As you can see, the numbers we add go up in steps of one, so it's a natural for a loop.

The number we're going to total up to is called *NUMBER*. Lines 30 and 40 get its value for us. Our answer is going to be stored in the appropriately named *TOTAL* which we set to zero with line 60.

For a moment, think about how you do a sum like 1+2+3+4. The answer doesn't just leap into your head all at once. You do it by adding two numbers, then adding the answer to the next number, then adding that new answer to the next number and so on.

In other words you think "One and two gives three. Three and three gives six. Six and four gives me ten. No more to add, that's the answer". We call it keeping a running total. This is how the micro does it, adding each new number to the answer arrived at so far.

See how Program VIII works, assume you've input 4, so we're asking the micro to do the sum we've just worked through. The actual work of adding is done in line 80, the body of the loop. This adds the integer we're considering to the total so far.

INTEGER goes from 1 to 4 successively. Since *TOTAL* is initially zero, the first time through the loop line 80 boils down to **TOTAL=0+1**, so our total so far is one - correct.

We don't actually do this first 0+1 step when we do it in our heads, but the micro is a very formal beast.

Next time through the loop, *INTEGER* is 2, and the current value of *TOTAL* is one so, TOTAL=TOTAL+INTEGER, which boils down to TOTAL=1+2 and *TOTAL* assumes the new value three.

Next time through, *INTEGER* is three, so line 80 becomes in effect TOTAL=3+3 and *TOTAL* adopts the

DON'T LET BASIC BUG YOU continued

value six.

the next time through - the final in this case - line 80 is equivalent to TOTAL=6+4 and *TOTAL* becomes ten. We then drop through the loop and print the total out with line 100.

Program IX uses exactly the same tecnique, but this time I wrap the whole thing up in an outer loop that "does the input" for me, giving endpoints for the ranges I'm totalling over from one to ten (line 30,100). In other words I get all the totals for:

1

1+2

1+2+3

and so on.

I've also gone to some trouble to improve the appearance of the output. Just before I add *INTEGER* to the running total (line 70) I print it out, together with an accompanying + sign (line 60). Thus the effect of the inner loop is that not only do we calculate *TOTAL*, but all the integers being summed appear on the same line with + signs between them.

When we drop out of the inner loop we then print out the answer. However, we need to do a bit of tidying up first. You see, each integer is followed by a +, from line 60. After the last integer though, we don't want a +, we want =.

```
10 REM PROGRAM IX
20 PRINT CHR$(125)
30 FOR RANGE=1 TO 10
40 TOTAL=0
50 FOR INTEGER=1 TO RANGE
60 PRINT INTEGER;"+";
70 TOTAL=TOTAL+INTEGER
80 NEXT INTEGER
90 PRINT CHR$(30);"=";TOTAL
100 NEXT RANGE
```

Program IX

Well, once the cursor has printed the final + it moves on to the next space on that line of the screen (the effect of the semi-colon). All we do is to move the cursor back with the magical **CHR$(30)**, overprint our + with an = and print *TOTAL*. Line 90 shows how it's done.

Then, of course, line 100 loops us back if we haven't done all our totals.

A final point. Before each repedition of the inner loop, in other words before we do each running total, we set *TOTAL* to zero. It's vital we set it back to scratch this way each time, otherwise we'll be adding in the previous running total to our current one. Try leaving line 40 out and running the program if you don't see what I mean.

THE END

GRAPHICS - DISPLAY LIST

Part IV of MIKE ROWE's series on how to give your program displays the professional touch

Do a back flip and go for a vertical scroll

We have examined the nature of the display list and how to alter it to create more professional displays. Now let's move on to using the display list to create special effects, in this case vertical scrolling and page flipping.

Page flipping is a term used to describe an action directly comparable with turning over the page of a book - instantly switching from one page or screen of information to a different one.

On many computers the screen memory is restricted to a set area of memory and nowhere else. On Atari computers any portion of memory can be used as the screen memory even including the ROM areas. As these cannot be altered this is of little practicle use, but illustrates the versatility of the machines.

Thus for many machines to change from one displayed screen to another you would need to erase the screen memory and redraw or reprint the new screen.

On the Atari you can just simply skip to a new area of memory, which is almost instant even by machine code standards. The procedure to accomplish this is unbelievably simple.

You may remember in the first article in the series I described the construction of the display list. I mentioned that the fifth and sixth numbers in the list were the memory location from which the screen display would be taken in the order low byte, high byte.

It follows then that changing these two numbers would, with only two pokes, change the area of memory

GRAPHICS - DISPLAY LIST

being displayed, that is page flipping. Program I demonstrates this.

This program skips through three areas of ROM, displaying them on a Graphics 0 screen. The speed is impressive, but the display is practically useless. More useful would be a program skipping through previously created screens.

Program II is a very simple demonstration of this. It creates 10 simple Graphics 2 screens identifying each one differently.

This is done by using a Graphics 2 call from Basic which makes the operating system create a Graphics 2 screen at the top of memory and then the message printed on the screen. The machine is then made to think that the top of memory is 0.5k lower by changing the value in location 106.

This number indicated the top of the available memory in pages - one page in memory terms is 256 bytes. Therefore if you subtract two from this location you get the top of memory lowered by 0.5k. You can then make another Graphics call and the screen will be located 0.5k below the previous screen.

This has been repeated 10 times in the example to give 10 Graphics 2 screens. The values for the screen memory for each screen are stored in the variables arrays SCREENLOW and SCREENHI and it is a simple matter to repoke these values back into the first display list to give the effect of flipping through the screens 1-10 as in the example.

This can now be seen to be more practical. However it is not greatly faster than redrawing each screen. Where the technique really comes into its own is in the higher resolution graphics modes.

Here it can take several minutes to draw a screen, or to load a screen from data stored on disk or tape and to redraw the screen even in machine code.

Each time a change is made can be very slow and useless for animation. Here page flipping can provide a tecnique for giving animation to very detailed drawings.

The drawbacks become more pronounced however. First memory limitations. If you use a Graphics 8 screen then five screens have already consumed 40k, not to mention where your program and DOS will go.

In practice two or three screens of Graphics 8 are the limit. The Atari

GRAPHICS - DISPLAY LIST

130XE could, of course, ease this problem somewhat by switching in and out different blocks of memory for the screens.

A second problem is that the 8k modes have a second set of numbers pointing to screen memory half way down the display list, as explained in previous articles, to avoid screen memory crossing a 4k boundary.

You must remember to calculate the new values for these and also to alter these when flipping.

Thirdly, drawing the screens in Basic is both slow and also memory-hungry, especially if using data statements.

This can be avoided by either loading predrawn screens off disk or tape directly into memory - used extensively in commercial programs - or by having a separate program for drawing screens which then loads in the second program in which the flipping takes place.

For this reason Program III uses a very simple Graphics 8 picture just to demonstrate the possibilities.

Those who are thinking ahead will perhaps have realised that if you can flip to anywhere in memory why not flip just one screen line. Do it repeatedly and voila - scrolling!

This is shown in Program IV, which scrolls through ROM using 20 byte (one screen line in Graphics 2) page flipping. The scrolling is, however, jerky and quite unprofessional in appearance.

Believe it or not, some software manufacturers released Atari programs commercially with scrolling of this type.

Those who have seen programs with good quality vertical scrolling, such as Caverns of Mars or Firefleet, will know that the Atari can produce superb scrolling.

You will remember from my first article in issue 23 that a 32 added to the display list graphics mode number gives vertical scrolling. However this does not give instant scrolling with that single change - in fact alone it makes no difference.

Also an operating system memory location is involved - decimal 54277 ($D405). In Program V the vertical scroll is enabled in line 3 of a Graphics 1 screen by adding 32 to that line in the display list and then 54277 is altered and there it is -smooth scrolling. But only of one line and only to the height of one character.

GRAPHICS - DISPLAY LIST

If all the graphic mode numbers in the display list are altered by adding 32 to them then all the lines will scroll together. However it is only to a maximum of 16 scan lines -two characters height in Graphics 1.

Now a bit of lateral thinking will provide the full answer. If you combine the two techniques of coarse and fine scrolling you will have true, full screen fine scrolling.

In other words fine scroll all the lines one character (eight lines in Graphics 1) by incrementing 54277 from 0 to 7. Then do a coarse scroll by one character by pointing the display list screen memory one line on and simultaneously poke 54277 back to 0.

Repeat this continuously and you have your scrolling, all in Basic, no machine code in sight. Program VI shows this technique.

But wait a minute - the screen flickers or flashes occassionally. Well, if you are a perfectionist - and with a perfect machine shouldn't we be? - it does flash occassionally.

This is because Basic is not instantaneous with its alteration in the values in the display list and in location 54277.

If the screen is in the middle of being drawn when a change is made a flicker occurs or the wrong line is displayed for a split second. Don't despair, there is a solution, but it means machine code.

In Program VII the same technique as Program VI is used, but instead of Basic poking the changes a small machine code subroutine is used. This does several pokes at once with machine code speed shortening the time lapse between the pokes, thus theoretically decreasing the glitches produced on screen.

As you will see, this is the case, but they still occur. In fact the only way to prevent the flicker completely is to make sure that the changes do not occur pat way down as the screen is drawn.

This means doing the dirty work during the vertical blank interrupt (VBI). As briefly explained in my previous articles, this means a short machine code routine which runs each time after the screen has been drawn and before the next starts.

Vertical blanks are a subject deserving of an article of their own, so I will go into no further detail than this at present.

At last you have it. True vertical scrolling as good as any arcade

GRAPHICS - DISPLAY LIST

game.

The only snag left is screen memory. Of course you are covering a much bigger area than one screen, so simple Plots and Prints will not really be adequate as such.

You have three real choices as to how to design your screens. Firstly you can use a long string to hold the data. This has the advantage that it relocates itself automatically and thus memory management is taken care of.

The snag is that you may accidentally cross a 4k boundary and cause chaos when the scroll reaches this point.

Another method is to calculate an area of memory you know is free and directly poke (or load off disk or tape) the screen data into that area. This is the method used in Program VI.

Finally, you can use a similar method to the page flipping demo in Program II - that is, repeated graphics calls after lowering the top of memory pointer.

This is not straightforward and will also involve playing around with the display list memory pointers and locations 88 and 89 to ensure that the screen data is continuous with the previous screen's data, thus avoiding garbage showing up between the screens as you scroll over them.

The advantage is that you can use Plot and Draw from Basic. I recommend the first two methods.

There you have it - your vertical scrolling completed. What? Your favourite games use horizontal or diagonal scrolling ?

Before continuing with part 5 of this scrolling article let me tell you first that all the DEMO programs are on the issue disk. There are 7 small demo programs and they are named VERTDEMO.1 to VERTDEMO.7

MIKE ROWE

takes a look at horizontal scrolling in Part V of his series on how to give your program display the professional touch

GRAPHICS - DISPLAY LIST

Horizontal scrolling is essentially achieved in a similar way to the vertical scrolling described above. However, as you will see, things are never quite that simple.

You will remember that coarse vertical scrolling can be achieved by moving the start of screen memory down the screen data one line at a time. Horizontal scrolling can be similarly achieved by moving the pointers for screen memory along one character at a time. This is shown in Demo #1.

Competent Programmers can do it sideways

Firstly you need to decide what data you want to show. I have chosen to hold the data in a string (A$). You will notice that you have to use CHR$(0) that is Control, (the heart symbol) to represent a space. This is because whenprinting to the screen a space is CHR$(32) but the Atari converts this to a 0 in screen memory. In fact all the numbers printed are stored as a different number in screen memory. Play around with A$ to confirm this.

Back to the program. The high and low bytes of the address of A$ - that is, screen memory - are calculated and stored in LO, OLDLO, HI and OLDHI. A cursor display list is created in Page 6 (memory location 1536), and the operating system is told that it is there by poking the low and high bytes of the display list into decimal 560 and 561.

Now we are ready to scroll. Firstly increase the low byte of the address of screen memory by 1. If the number is greater than 255 then reset LO to 0 and increase HI by 1.

You now have the new address of screen memory moved along by one byte - one character. Theses values can now be placed in the two bytes following the LMS command (Load Memory Scan -see previous articles) as in line 210. Repeat this and there you have coarse scrolling of one line.

Smooth scrolling is again similar to

GRAPHICS - DISPLAY LIST

vertical smooth scrolling. Demo #2 shows how this is used. A similar display list is used but a decimal 16 is added to the mode byte. This is 71 - that is, Antic mode 7 + LMS instruction 64. Adding a 16 enables smooth scrolling in tha line.

The horizontal smooth scrolling register is decimal 54276 ($D404). This can be poked with numbers uo to 15 which will move the line along one pixel at a time up to a maximum of 16 - two Graphics Mode 2 characters.

Now obviously combining these two tecniques will result in true smooth horizontal scrolling. Demo #3 is essentially the same as Demo #1 but with the smooth scrolling added to it.

Now we begin to get the first drawback, screen flicker. This is because the changes often occur part way through the creation of the screen on the television.

Things can be improved somewhat by making some of the changes more rapidly in machine code, as in Demo #4. the machine code here simply pokes each memory address with the byte following it but much more rapidly than in Basic.

Okay, so we have reasonably good smooth scrolling of one line. Not going to make much of a game is it? The next step is to extend this to full or part screen scrolling.

Unfortunately things are not as simple as in vertical scrolling, where the rest of the screen will follow the first line.

Firstly the screen memory needs to be considered. In vertical scrolling the screen is the same width as a normal screen. However in horizontal scrolling the screen is going to be much wider. Therefore using operating system commands such as DRAW or PRINT are really out of the question.

Secondly each line needs to have its own LMS instruction. Therefore the display list will consist of groups of three numbers.

the first will be a 64 (LMS) + mode number + 16 (for smooth scrolling). The LMS instruction means that the next two numbers will be the low and high bytes of the screen memory for that line.

Now we have a display list consisting of numerous individual lines each similar to the single line in Demo #1 to 4. To scroll these you need to move each of the memory location pointer along one byte at a time.

Demo #5 does just this in Basic. As

GRAPHICS - DISPLAY LIST

you can see, the "scroll" goes in waves down the screen. Basic is just too slow.

Demo #6 is the same program with a machine code routine doing the job of increasing the 10 screen location pointers in the display list. Now the screen moves along in a single block.

There's that flicker again...

The next step is to add the smooth scroll. Demo #7 does this with an improved machine code routine for the coarse scroll. However it is less generalised, and will only work for a 10 line screen scrolling in one direction.

You now have your smooth horizontal scrolling. Disappointed? I would be, because there's that flicker and flashing again.

The same problem arises as before, because the changes happen part way through drawing the screen, only now things are worse because so many alterations are being made to get smooth scrolling.

As in vertical scrolling, the only way around this is to use machine code during the vertical blank interrupt (this is the small delay between the drawing of each screen).

Although VBI routines are too complicated to discuss at the end of this article, Demo #8 will give you some idea of how much improvement they can give in scrolling.

As a parting note, on the XL and XE models smooth, upward vertical scrolling is very easy in Graphics 0.

Try this. First load a relatively long Basic program. Secondly type POKE 622,255 then press Return. Thirdly type GRAPHICS 0 then Return. Now list the program.

Again all the DEMO programs are on the issue disk. The programs are named HORIDEMO.1 to HORIDEMO.8

GAMES REVIEW

Hello there! Well, the AMS '96 show has been and gone so hopefully some of you will have managed to have made it there. I had to miss this one due to work commitments so I'll have to change my normal routine and attend the next one. If you've still got any money left, here's a review of a couple more items of software which you may wish to try:

Review by Kevin Cooke

Title: AUTODUEL

Sold by:

Micro Discount
265 Chester Road,
Streetly,
West Midlands B74 3EA,
ENGLAND.
Tel:0121 353 5730

Price: 5 Pounds 95p (+ P&P)

Autoduel is a strategy/ arcade-type game, based on the CAR WARS board game by Steve Jackson. If it gives you an idea as to what the game is about, the title screen says "Autoduel -where the right of way goes to the biggest guns"!

For those users who are new to the Atari and decide to buy this, you'll probably be very surprised when it pops through your letterbox. "Why?", I hear you ask! Well, in the "good ol' days" of Atari computing, the bigger software companies spared no expense on their software packaging. Autoduel is no exception. Inside of the glossy box is a thick instruction manual (much like the one that comes with the Alternate Reality series of games), a quick-reference card, a leaflet detailing other software by Origin, a road map and.... wait for it... even a tool kit containing a miniature spanner, hammer and screwdriver! Yes, all real and made of metal! Of course, you're unlikely to actually use any of them but it's a nice freebie!

Of course, the main items in the box are the two disks. Thoughtfully, when

GAMES REVIEW

you load the first disk, your computer asks you whether you would like to use one or two disk drives - a nice thought for those of us that own more than one as it cuts down considerably on disk swapping.

The game is set on a futuristic Earth (in America to be precise) and your character starts off as a nobody owning only $2000. By traveling from city to city, you must aim to increase your money and status. This can be done in a number of ways - either by traveling the roads, acting as a vigilante by clearing it of outlaws, by fighting in the American Autoduel Association's (AADA) arenas, or by acting as a courier for the AADA.

You start off the game in New York with your immediate aim being to get some money. There are two viable methods for doing this at this early stage in the game - by traveling by bus to Atlantic City and gambling in the casino, or by competing in Amateur night at the AADA arena. The arena are kind enough to loan you a poor, but adequate "Killer Kart" with a front-mounted machine gun for this purpose and, after you've won a few matches, you'll soon have more money and prestige. At this stage, you can either become a full-time competitor in the arenas (and improve until you reach the top where you can earn really big money), or take another job. Whatever you choose, you'll have to buy a car from the choice of 7, pick it's chasis type, armour type, suspension, weapons, tyre type and power plant to get it moving (these futuristic cars don't need conventional engines!). What components you use on your car will depend on what you want to use it for, and of course, the all-important money factor!!!

Once you start to play Autoduel, you soon realise that it is more a way of life than a game - you almost start to feel that you ARE your character. Every detail has been seen to in the game - the instruction manual gives in-depth descriptions of playing methods, theory of car designs, information about the arenas, outlaws, etc. whilst the map allows you to plan your journeys effectively.

There isn't much sound at all in the program but, then again, in a game of this type where you are likely to spend long periods of time playing it, music would almost certainly start to

GAMES REVIEW

get on your nerves anyway.

Graphically, the game is very good. Everything is well designed and set out, making play a joy.

The only let-down is the fact that the graphics are artifacted - a method that creates very colourful graphics if you happen to live in America but which, in the UK produces black and white screens. However, as colour is not an integral part of the game, no problems are caused and you soon get used to it.

Apart from this, I can't recommend Autoduel enough. The mixture of the arcade driving/arena sections with the added strategy of designing your cars, etc. make it almost perfect.

The only other programs with depth anything like this are SEVEN CITIES OF GOLD (which is no longer available anyway) and the two ALTERNATE REALITY programs.

If you own these and are still looking for more, or if you feel like an envolving arcade game, give this a try. At this price, it's a snip!

Title:

EUROPEAN SUPER SOCCER

Sold by:

Micro Discount,

265 Chester Road,

Streetly,

West Midlands B74 3EA,

ENGLAND.

Tel:0121 353 5730

Price: 1 pound 50p (+ p&p)

European Super Soccer ("ESS" from now on!) is a footie game written by Brian Jobling (of Zeppelin fame) for Tynesoft a few years ago.

The game kicks off with a lovely title screen and some very lively, if not particularly apt, music.

At the time the game loads, one very large footballer is shown on the title screen, indicating that the game is in one-player mode. By pressing the OPTION key, another player appears on the other side of the screen and the player can now play a two-player game. Pressing the select key will change the player's kit-colour - a nice touch.

GAMES REVIEW

Whichever game mode you choose, pressing start will bring part of the pitch onto the screen. ESS is a horizontal scrolling game and so you only see about one fifth of the whole pitch at any one time.

When you start playing the game, you realise that it's actually OK. The scrolling is smooth, the graphics are well done with the players being well defined (if a little blocky).

Actually, there's not a lot that can be said about a football game. If I had to make any criticisms, they would have to be that the ball never moves above waist level (no headers here!).

Also, instead of having the two teams in strips of totally different colours, the author opted to have player one in (for example) a red shirt and black shorts and player two in a black shirt and red shorts. This means that it can be difficult to find your own players for much of the time. Under the circumstances it may have been better if the author had made team one all dressed in red and team two all dressed in black.

In two player mode you get to choose your team from a list of six before the computer randomly chooses it's team. By the way, how long have the USA and "Atari World" been European teams?!!!

Overall, ESS is a fairly good version of football (or soccer!). The computer doesn't play particularly harshly so, if you're always getting a thrashing at other football games, this one could be right up your street!

I don't think that ESS beats Anco's KICK OFF or Thorn Emi's SOCCER for playability but if you're looking to add another footie game to your collection, it's worth a try at this price.

[Now for the bad news - I've just spoken to Derek Fern and he tells me that he has sold out of all of his copies of Autoduel at the show. He has contacted American Technovision in the USA to try and obtain more stocks but they have apparently decided to leave the Atari scene and move totally onto the PC. If you want to try Autoduel, keep a look out in future editions of Derek Fern's catalogues - who knows, he may get more stocks of it. In the meantime, why not take a look and see what other software he has on offer?].

GAMES REVIEW

Now, onto other things. It is with great regret that I have to announce that, from this issue onwards I am no longer able to do this review column for TWAUG. College work and my part-time job has been gradually eating into my spare time and, these days, it seems that the only time I get on the computer is to write this review column and college essays!!! Apart from this, I keep getting VERY close (and sometimes missing!) the deadlines and it's not fair on the TWAUG team to put them under extra pressure. I feel this is the stage to bow out gracefully and let some other able person write the reviews.

I hope that you've enjoyed reading my columns as much as I have writing them. I must state that I am NOT leaving the Atari 8-bit scene (you may well see some more programs, articles and reviews from me in the TWAUG, Futura and Page 6 magazines if or when I get time to write them) so, if you have been writing to me, don't stop - I enjoy getting the letters!

All that remains for me to do is to wish TWAUG all the best for the future and hope that the forthcoming years are as good, or better, than the last few.

This review by Kevin Cooke should have been in the last issue, but as I mentioned then I was unable to retrieve it from the disk and as soon as Kevin read about my predicament he immediately sent me another disk.

This will be the last of the reviews unless we find a replacement reviewer, so we are still hoping some kind and able person will come forward and offer his or her contribution to TWAUG.

BIT WISE

MIKE BIBBY gives you the lowdown on ...

The inside story of binary operations

In previous articles we've seen that binary numbers can be added and subtracted just as our more familiar decimal numbers are. And, of course, we can multiply and divide them.

There are, however, other ways of combining two binary numbers that are extremely useful in dealing with computers. They're also easy to use, so let's have a look at them.

Firstly, we'll see how we can NOT a binary number - simple, one-bit numbers first. By the way, we're going to be dealing exclusively with binary numbers, so we can drop the % sign.

The rules for doing a NOT are simple:

If the bit is 1 then it becomes 0
If the bit is 0 then it becomes 1

If you like, the NOT converts a bit into its opposite.

So **NOT 1 = 0**

And **NOT 0 = 1**

Why do we use the word NOT?

Well, mathematicians often use the number 1 to mean true and 0 to mean false.

So NOT 1 means not true, which means false, which is 0. That is, NOT 1 is 0. And, as not false is most certainly true, NOT 0 is 1.

If we are to NOT a binary number consisting of several bits, we simply apply the rule for NOT to each bit individually.

So **NOT 10110010**

becomes **01001101**

Some people think of this process as turning the number on its head, so it's sometimes called inverting. Others call it taking the complement

BIT WISE

of the number.

NOT just works on a single binary number. However, there are other sums or operations that have a set of rules for combining two binary numbers.

For instance, we can AND two binary numbers. Let's look at the rules for ANDing a single bit with another bit.

When you think about it, there are four possible combinations of bits that we could AND - 0 with 0,0 with 1,1 with 0 and 1 with 1.

We write that we are ANDing, say, 0 and 1 as 0 AND 1.

The rules for ANDing are:

0 AND 0=0 (case a)
0 AND 1=0 (case b)
1 AND 0=0 (case c)
1 AND 1=1 (case d)

Notice that the only time the result is 1 - true - is when the two bits ANDed are both 1 - true. This helps us to see why we use the word AND to describe the operation.

If you think of the first bit as "this" and the second bit as "that", what we're doing when we're ANDing is asking whether "this and that" is true.

"This and that" can only be true when both "this" is true AND "that" is true - hence the use of AND to describe the process.

For example, consider the statement that it is dry and sunny.

This is true only if dry is true and sunny is true - case d.

If either of the two, or both are false - case a, b, c - the whole statement is false, since it isn't both dry and sunny.

We can AND pairs of binary numbers of more than one bit - just apply the rules of ANDing to each bit individually.

For example:

AND 10010110
AND 10110011
gives **10010010**

We can also OR two binary numbers. The rules for ORing a single bit with another bit are as follows. Again there are four possible combinations:

In this case you only get a false result, 0, when both bits are false. If

BIT WISE

either or both bits are true, 1, the result is true. It's easy to see why we use OR to describe this. If one OR the other OR both is true the whole thing is true.

0 OR 0=0 (case e)
0 OR 1=1 (case f)
1 OR 0=1 (case g)
1 OR 1=1 (case h)

Let's use the meteorological analogy again. Consider the statement that it is dry and sunny.

This is only false when it is NOT dry and NOT sunny - case e -otherwise it is TRUE - cases f, g, h.

To sum up, with OR the whole thing is true if *either* or both the things being ORed is true.

As we did with AND, we can OR pairs of numbers with more than one bit - we just apply the rules of ORing to each bit individually.

For example:

10010110
OR 10110011
gives **10110111**

EOR - A way to find

Above we looked at the AND and OR operations on binary numbers - logical operations, as they are known. These are simply rules for combining numbers bit by bit. We shall continue our exploration now with a look at the EOR operation.

EOR stands for Exclusive OR - sometimes people call it XOR. Either way it's the same thing. EOR is a variant on the way we normally use the term OR.

For example, if I say:

Mike OR *Pete wears glasses*

this is true if Mike wears glasses, OR Pete wears *glasses*, OR both Mike and Pete wear glasses.

Now it's this last case of OR we're interested in, where they both wear glasses. EOR works just like OR up to this point. However, EOR does not "allow" both of them to wear glasses. Either one does, or the other, but not both.

To put it another way, the one who wears glasses does so *exclusively.*

If both are wearing glasses then

BIT WISE

while:

Mike OR *Pete wears glasses* would be true.

Mike EOR *Pete wears glasses* would be a downright lie!

We could signify that a statement is true with the letter T, and use F for false. At school our teachers used ticks for truth and crosses for false. Since we're using computers, though, we'll use numbers: 1 will denote true and 0 will denote false. We've chosen 1 and 0 because they fit in so well with the binary system.

So, in the above example, if Mike has glasses we can give Mike the value 1. If Pete hasn't glasses we can give Pete the value 0. Table I shows the idea, applied to each combination of spectacle user. The ones and zeros are known as truth values, states or conditions.

As you can see, there are four possible cases as far as Mike and Pete wearing glasses are concerned: neither can wear them as in case 1, where both Mike and Pete has 0 value.

Then again, Pete may wear them (1) whereas Mike does not (0), case 2, and so on.

If you look carefully at the numbers involved in all four cases, you see that we've got four pairs of bits we can combine. Each pair of bits is made up of the "truth bit" for Mike and the "truth bit" for Pete.

out who's telling the Exclusive truth

	Wears glasses		
	Mike	Pete	
Case 1	0	0	neither wears glasses
Case 2	0	1	Pete wears glasses
Case 3	1	0	Mike wears glasses
Case 4	1	1	Both wear glasses

Table I

What I've done in Table II is to combine these pairs for all four cases in accordance with out OR rules. We've stored the result in a third column.

We call such a table a Truth Table. In this case, it's the truth table for OR. We can use it to work out

BIT WISE

the result for any OR combination of two bits. All we have to do is to find the row that starts with the two bits values we're combining and then look in the third column for the result.

Mike wears glasses	Pete wears glasses	Mike OR Pete wears glasses
0	0	0
0	1	1
1	0	1
1	1	1

Table II

Table III shows a similar table for:

Mike AND *Pete wear glasses*

Again the first two columns are identical, covering all four possible cases. The third column combines them according to the AND rules.

Look again at Table II. This corresponds in a sense to our binary rule for OR: you get a 1 if either or both bits you combine contain a 1.

However if when talking about Mike and Pete you mean OR in the exclusive sense, EOR, then the combination of Mike wearing glasses and Pete also wearing glasses would have to be false. This is because EOR means either one or the other wears glasses, but not both - it's exclusively one or the other.

If we do mean EOR in this exclusive sense we'd write our statement about them as:

Mike EOR Pete wears glasses

Its Truth table is given in Table IV:

If you look at each case, you'll see that the only time Mike EOR Pete is true is when either one or the other wears glasses, but not both (or neither).

More formally, if both bits are 0, or both bits are 1 the result is 0. If either is 1 and the other is 0 the result is 1. To put it another way, if the bits are identical the result is 0, otherwise the result is 1.

Let's have a look at how we EOR binary pairs of numbers. Its the same as for OR and AND - just apply the rules for EORing to each pair of bits in succession. For example:

```
      %10110110
EOR   %11100101
gives %01010011
```

BIT WISE

Mike wears glasses	Pete wears glasses	Mike AND Pete wears glasses
0	0	0
0	1	0
1	0	0
1	1	1

Table III

Take a look at what happens when you EOR a number with zero:

```
      %10110110
EOR   %00000000
gives %10110110
```

that is, when you EOR a number with zero it leaves that number unchanged. Also something interesting happens when you EOR a number with itself:

```
      %10110110
EOR   %10110110
gives %00000000
```

Whenever you EOR a number with itself, the result is zero. This is as it should be: remember, when you EOR two identical bits the result is zero.

Now EOR has a property which makes it quite useful - lets look what happens when we take a number, EOR it with a second number and then go on to EOR the result once more with that second number.

```
First number          %10101101
Second number EOR     %01101000
Result                %11000101
Second number EOR     %01101000
Final result          %10101101
```

as you can see, the first number has magically re-appeared! This always happens when you EOR twice with the same number as, in a sense, the two EORings cancel each other out.

Table V summarises the process for all four possible pairs of one-bit numbers. As you can see, for all the cases the final resulting bit (when

Mike wears glasses	Pete wears glasses	Mike EOR Pete wears glasses
0	0	0
0	1	1
1	0	1
1	1	0

Table IV

BIT WISE

the first bit has been EORed twice with the second) is identical to the first bit.

Another way to think of it is that we are doing:

first number EOR second number EOR second number

Taking the underlined part first, we've already seen that any number EORed with itself gives a zero result. So what we're really doing is:

first number EOR 0

which, as we've also seen, must leave just the first number, since EORing with zero leaves a number unchanged.

All this may seem rather abstruse, but actually it's quite useful. In fact we tend to use AND, OR and EOR quite often in graphics, particularly in animation.

To simulate movement we frequently print something on the screen, then after leaving it there for a while to register on the eye, we blank it out and print it in a new position and so on.

First bit	Second bit	Result 1st EOR	Second bit again	Result 2nd EOR
0	0	0	0	0
0	1	1	1	0
1	0	1	0	1
1	1	0	1	1

Table V

Sometimes we blank the character out by printing it again in the same place but in the background colour.

We can, however, use EOR. If we use EOR to place our character on the screen - never mind exactly how for the moment - when it comes to wanting rid of it, we can just repeat ourselves.

That is, we just EOR the character on again. As we've seen, the effect of two EORs is to cancel each other out. In this case, they cancel out to the original background - and the character disappears.

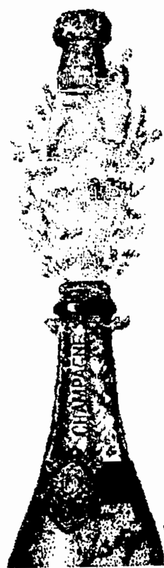
The point is, logical operators, as AND, OR and EOR are known, can be invaluable to both the Basic and machine code programmer. Next time we'll take a brief look at the idea of masks.

Another year is with us

We would like to wish you all the very best of health and happiness and good fortune

And please spare a thought for

C.W.A.U.G.



Have a very

Happy New Year

1997

T.W.A.U.G.'s SURVEY

We would like to find out how many of our subscribers use an Atari ST for a second computer!

I for instance use a MEGA 1 ST to do the work on the newsletter, because I find I can do the work a lot faster with a DTP program than I could using Daisy-Dot 3 on the 8-bit.

I have been toying with the idea of including in the newsletter material concerning the ST, maybe at some future date.

This is the reason for this survey, if we could get some feed back on this matter it would be very helpful.

The membership to the newsletter is around the 80 plus and I am sure some of you would like to air your views on this matter.

We would be very happy if everyone would write us a note.

MAX

SALE

Upgraded 800XL to 256K complete with Power supply for £40 or near offer.

Contact:

Alan Turnbull on

01670 - 822 492

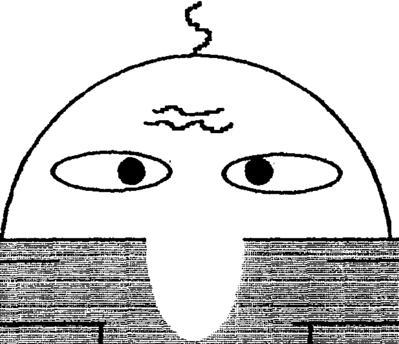
These computers make an ideal inexpensive gift.

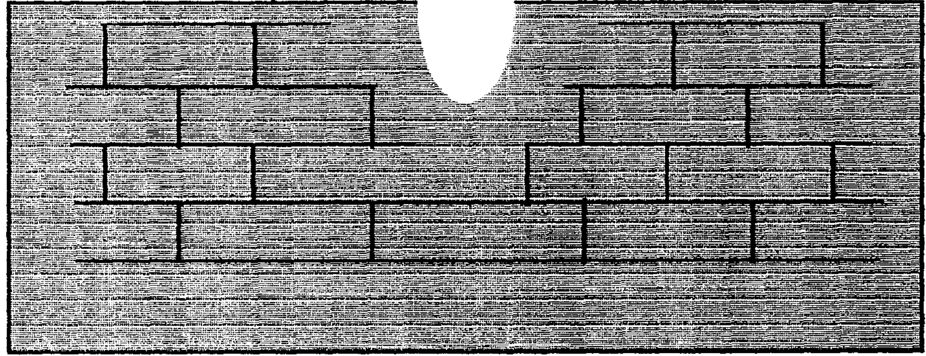
SENDING MAIL

Please do not forget, when sending letters and renewal forms to TWAUG to use the new address below.

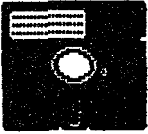
The reason for repeating this message is because some mail had still been addressed to the POST BOX. Fortunately John had been able to retrieve it. John had been told that any mail addressed to the box in future will be return to the sender, or if no return address on envelope, they will dispose of that letter.

TWAUG
c/o J. Matthewson
80 George Road
Wallsend, Tyne & Wear
NE28 6BU





WOT? NO MAIL?



DISK CONTENT

This disk has been dedicated to TWAUG for issue #25 by John Foskett, we thank him whole-heartedly.

Side A contains a variety of program and demos. The programs by John are:

My First Demo, a Mini colour scrolling Demo, Battle ship, Crazy Ball and a Card Game.

Now also on Side A, I included the demo program I have typed in to go with the article Display List. I've named the program so that you know which program is which when reading the article. From page 11 to page 15, the program are named VERTDEMO.1 to VERTDEMO.7. From page 16 to page 18, the program are named HORIDEMO.1 to HORIDEMO.8.

All these programs run alright, I've tested them after each I had saved them to disk.

Side B is in Autorun and Turbo Basic, that John Fosket set up, it run the Crazy Ball Game. We have also included on Side B the MyDOS with the manual. It had been requested by some members, this is a bonus for you to enjoy.

ADVERTISING USER GROUPS

LACE

The LONDON ATARI COMPUTER ENTHUSIASTS

As a member of LACE you will receive a monthly newsletter and have access to a monthly meeting. They also support the ST and keep a large selection of ST and 8-bit PD software.

The membership fee is

£8.00 annually

for more information contact:

Mr. Roger Lacey
LACE Secretary
41 Henryson Road
Crofton Park
London SE4 1HL
Tel.: 0181 - 690 2548

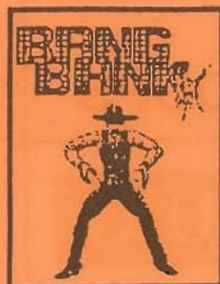
O.H.A.U.G.

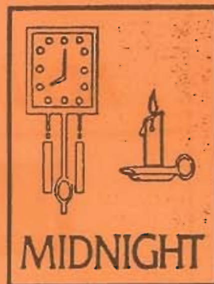
The OL'HACKERS ATARI USER GROUP INC.

O.H.A.U.G. is an all 8-bit user group in the STATE of NEW YORK.

They are producing a bi-monthly double sided disk based newsletter. The disk comes with its own printing utility, which lets you read the content of the disk, one screen page at the time, and/or you can make a hard copy of the disk, in one, two or three columns and 6 to 8 lines to the inch. A large PD Library is available.

Contact:
Mr.Ron Fetzer
O.H.A.U.G.
Secretary & Treasurer
22 Monaco Avenue
Elmont, N.Y. 11003
USA





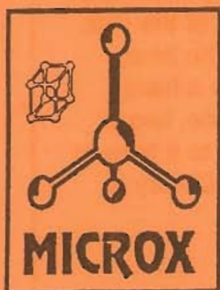
Are You Still Complaining about lack of New software for your Atari 8 bit ?

If you are you don't know about Mail Order From

MICRO-DISCOUNT

265, CHESTER ROAD.
STREETLY.
WEST MIDLANDS.
B74 3EA.
ENGLAND.

TEL 021 353 5730
FAX 021 352 1669



ADAX *HANS KLOSS*

DARKNESS HOUR