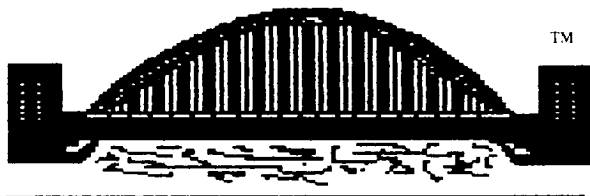


TYNE & WEAR



ATARI



8-BIT

USER GROUP

Issue 24

November/December 1996

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

A Reminder

From this month "November" our old PO Box address has been closed.

As mentioned in our last two issues, we closed the PO Box because we didn't get the service we had expected.

Over the last four years, that we used the PO Box, we had a number of unexplained occurrences where our mail had been put into other boxes and we never received it. We complained on numerous occasions, but things never changed.

All we wanted was that the mail addressed to our PO Box was put into the correct box.

This doesn't paint a nice picture about our post offices, but we have had enough, and yet we still have to use them.

So 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

TWAUG NEWSLETTER



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

NEEDS  YOU

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 25 is due mid-January 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	10
BIT WISE	
by Mike Bibby	17
MEMORY MONITOR	
Program is on disk	26
SALES SECTION	27
ARTISTIC IMPRESSION	
by Max	28
DISK CONTENT	29
BAD NEWS REPORT	29
USER GROUPS ADVERTS	
for LACE & OHAUG	30
FESTIVE SEASON GREATINGS	
from TWAUG to members	31
ADVERTISING	
MICRO DISCOUNT	32

TWAUG NEWSLETTER

DON'T LET BASIC BUG YOU

Part VII of MIKE BIBBY's guide through the micro jungle

I told you a lie in last issue when I said we'd be continuing with loops! Before that we've an interesting diversion into the world of strings.

Strings are simply groups of characters, letters, numbers or punctuation marks and so on, "strung" together. The microremembers them as a group. More often than not, they're words or sentences as in:

```
PRINT "This is a string"
```

Notice the quotes - they're the way we signal to the Atari that it's a string we're dealing with. Also, whatever's between the quotes is reproduced exactly so:

```
PRINT "This is a string"
```

and:

```
PRINT " This is a string"
```

will give different outputs, since the second has two spaces between the quotes and the first word.

We saw that we can label strings with variables as in:

```
STRING$ ="This is a string"
```

The rule for string names are identical to those for numeric variables, except that string variable

names
must end
with a \$.

Going

Perhaps
the most

important thing about strings is that you have to tell the micro in advance the maximum size they're going to be. That is, you have to DIMension them as in:

```
30 DIM STRING$(6)
```

In this case, STRING\$ can only ever be six characters long. Of course, it can be under six long, it just can't be over six, as Program I illustrates.

If you run it, you'll see the following output:

```
12345
```

```
123456
```

```
123456
```

```
123
```

loopy over

strings...

TWAUG NEWSLETTER

DON'T LET BASIC BUG YOU continued

I think you can guess what's happened to the 1234567 you assigned to STRING\$ in line 80 - it was one character too long, so the Atari simply ignored the last character.

```
10 REM PROGRAM I
20 PRINT CHR$(125)
30 DIM STRING$(6)
40 STRING$="123456"
50 PRINT STRING$
60 STRING$="123456"
70 PRINT STRING$
80 STRING$="1234567"
90 PRINT STRING$
100 STRING$="123"
110 PRINT STRING$
```

Program I

However, the micro doesn't mind the string being shorter than maximum size, as shown by its accepting 12345 quite happily.

And just as you can lengthen strings, so you can shorten them, as shown by the assignment of 123 in line 100.

After running the program, enter:

```
PRINT STRING$
```

and you'll find that it's 123, as the program left it. Now enter:

```
STRING$=""
```

Notice that the two quotes go directly side by side, with no space between. Then enter:

```
PRINT STRING$
```

This time instead of printing 123 as the value of the string, absolutely nothing appears on the screen save for the READY prompt. This is because the value of STRING\$ is absolutely nothing since:

```
STRING$=""
```

has absolutely nothing between the quotes, not even a space. This string is called the null string and we use it when we wish to wipe out, or nullify, a string.

To see some more aspects of strings on the Atari, assign:

```
STRING$="HELLO"
```

and then enter:

```
PRINT LEN(STRING$)
```

You should receive the answer:

```
5
```

You see, LEN() is a function that tells you how long a particular string is. We've used it with a string variable, but you can, if you wish, use it with a string in quotes as in:

```
PRINT LEN("ABCD")
```

This might seem a bit cockeyed to

TWAUG NEWSLETTER

DON'T LET BASIC BUG YOU *continued*

you, though. Why do we need LEN since if the string's in quotes you can see how long it is, and if it's a variable you've already dimensioned it? Don't forget, though, that a string doesn't have to be the size you've dimensioned it, it can be less.

Often, when you're using INPUT with a string variable, you won't be too sure how long the string will be. LEN lets you find out so you can, for instance, allow for different lengths in your screen layout.

Before we leave LEN, try to find the length of the null string with:

```
PRINT LEN("")
```

You'll receive the answer zero, proving that the null string consists of absolutely no characters whatsoever.

Incidentally, we've only been able to do all this assigning to STRING\$ because we've dimensioned it when we ran Program 1. If we tried to assign to a new variable as in:

```
TEST$="HELLO"
```

we get an error message, since TEST\$ hasn't been dimensioned.

Anyway, at present STRING\$ should hold the value HELLO. Try entering the following:

```
PRINT STRING$(1)
```

and you'll see HELLO appear. Now

try:

```
PRINT STRING$(2)
```

and ELLO should arrive.

```
PRINT STRING$(3)
```

will give you LLO, while:

```
PRINT STRING$(4)
```

will produce LO.

No prices for guessing what:

```
PRINT STRING$(5)
```

gives you!

What's happening is that by following STRING\$ with the brackets, only part of the string is printed. It's as if we're taking just a slice of the string.

The start of the slice will be the character whose position in the string is given by the number in brackets. It finishes at the end of the string.

So STRING\$(3) would give us from the third letter of STRING\$ to the end, giving LLO. With STRING\$(1) the whole string is printed out since the 1 means the slice starts at the first letter. On the other hand STRING\$(5) gives us a single character slice, since the fifth letter of the string is also the last.

See what happens when you try:

```
PRINT STRING$(6)
```

TWAUG NEWSLETTER

DON'T LET BASIC BUG YOU *continued*

Program II gives a graphic example of this slicing using a FOR...NEXT loop. Here the loop variable START decides the starting position of the slice. Each time through the loop the slice starts further up the string.

```
10 REM PROGRAM II
20 PRINT CHR$(125)
30 DIM STRING$(10)
40 STRING$="ABCDEFGHJIJ"
50 FOR START=1 TO 10
60 PRINT STRING$(START)
70 NEXT START
```

Program II

Let's look at this in detail. The first part of our program clears the screen then sets STRING\$ equal to ABCDEFGHJIJ. We then enter the loop.

The first time through START has the value 1 so line 60:

```
60 PRINT STRING$(START)
```

is in effect:

```
60 PRINT STRING$(1)
```

If you've been following this you'll see it means we can print out from the first character of STRING\$ to the last. That is it prints:

```
ABCDEFGHIJ
```

The next time through the loop

though START is 2, so line 60 is in effect:

```
60 PRINT STRING$(2)
```

so we can start at the second character of STRING\$ and continue to the end to give:

```
BCDEFGHIJ
```

On the next cycle, START is 3, line 60 being:

```
60 PRINT STRING$(3)
```

which gives you:

```
CDEFGHIJ
```

and so on.

Finally, START has the value 10, so line 60 prints out the slice from the 10th character to the end - the single letter J.

Actually we can slice off any part of a string we want by giving two numbers in brackets separated by a comma. The first number specifies the start of the slice and the second the finish.

Enter the following (assuming you've run Program II):

```
PRINT STRING$(2,6)
```

You'll get back:

```
BCDEF
```

Remember, STRING\$ is ABCDEFGHJIJ, so STRING\$(2,6) gives us the

TWAUG NEWSLETTER

DON'T LET BASIC BUG YOU *continued*

slice with its second letter B, and finishing with its sixth letter F. Notice you get five letters, not four that 6-2 might lead you to expect.

Program III allows you to experiment with slicing STRING\$. Initially STRING\$ is printed out, then you'll be prompted for the number of the character you want to start from, and the number you want to finish with.

The slice you've requested will be printed out, and the process

```
10 REM PROGRAM III
20 PRINT CHR$(125)
30 DIM STRING$(10)
40 STRING$="ABCDEFGHIJ"
50 PRINT STRING$
60 PRINT "START";
70 INPUT START
80 PRINT "FINISH";
90 INPUT FINISH
100 PRINT STRING$(START, FINISH)
110 PRINT
120 GOTO 50
```

Program III

repeated. (You can escape from the loop by pressing the Break key.)

Play around with various slices until you're sure you understand how they operate, then have a look at how

Program III actually works. Lines 60 and 80 prompt for an input value for the aptly named numeric variables START and FINISH. Line 100:

```
100 PRINT STRING$(START,FINISH)
```

then gives us exactly the slice we want.

For instance, if we wanted a slice from the second character to the sixth, we would input 2 for START and 6 for FINISH. Line 100 then becomes effectively:

```
100 PRINT STRING$(2,6)
```

which gives us the slice we require, starting at the second character of STRING\$ and finishing with the sixth.

Program IV uses this slicing technique to give us the inverse of Program II by printing out the first character of the string, then the first two, followed by the first three and so on.

The loop formed by lines 50 to 70 does the actual printing out. The slice always starts at the first character of the STRING\$ so the first number inside the brackets in line 60 is fixed at 1:

```
60 PRINT STRING$(1,FINISH)
```

FINISH varies from 1 to 10 throughout the loop, so the end of our slice gradually gets further and

TWAUG NEWSLETTER

DON'T LET BASIC BUG YOU *continued*

```
10 REM PROGRAM IV
20 PRINT CHR$(125)
30 DIM STRING$(10)
40 STRING$="ABCDEFGHIJ"
50 FOR FINISH=1 TO 10
60 PRINT STRING$(1,FINISH)
70 NEXT FINISH
```

Program IV

further along STRING\$, giving us our triangle of letters.

After you've run Program IV enter:

```
PRINT STRING$(1,LEN(STRING$))
```

As you'll see, the whole of STRING\$ is printed out. The reason is that LEN(STRING\$) gives us 10, the length of STRING\$. This means that what we've entered above boils down to:

```
PRINT STRING$(1,10)
```

Since STRING\$ starts at its first character and finishes with the tenth, the whole of the string is printed out.

Finally, take a look at Program V. We're using the fact that:

```
PRINT STRING$(5,5)
```

prints out the fifth character of STRING\$, since the slice starts and ends with the fifth character. Instead of specifying a number, however, we've made the loop variable, LETTER, which ranges from 1 to 10,

```
10 REM PROGRAM V
20 PRINT CHR$(125)
30 DIM STRING$(10)
40 STRING$="ABCDEFGHIJ"
50 FOR LETTER=1 TO 10
60 PRINT STRING$(LETTER,LETTER)
70 NEXT LETTER
```

Program V

so line 60 reads:

```
60 PRINT STRING$(LETTER,LETTER)
```

this will pick out and print each letter of STRING\$ in turn, as you'll see when you run it.

By the way, we could have written line 50:

```
50 FOR LETTER = 1 TO LEN(STRING$)
```

Since LEN(STRING\$) is 10, this is equivalent to the original line 50. It has the advantage that, if you missed out one of the letters of STRING\$ when you typed in line 40, the LEN(STRING\$) automatically compensates for the error, calculating the true length.

Well, perhaps I didn't mislead you too badly after all - we've used quite a few loops this month. And in the next issue there'll be even more!

GRAPHICS - DISPLAY LIST

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

A CUSTOM display list, mixing several modes on the same screen, can quickly and easily give your display a professional touch.

There are two ways to create one. Firstly you can modify a standard display list created by the operating system after a graphics call. Secondly you can create an entirely new list from scratch, or even have several display lists in memory at the same time.

Before you start to construct your list there are several problems to be considered.

If you are modifying an existing display list it is safest to use the graphics mode that takes up most memory in your final display list as a starting point for your modified list.

Also try to avoid your screen memory crossing a 4k boundary -4k, 8k, 12k, and so on to 48k - as it will cause problems. If you must cross a

border, say if an 8k mode is used, then when the screen reaches the boundary you need to insert another load memory scan - see last issues article - in the display list to point to the start of the next 4k block of screen RAM.

Different graphics modes take up a different amount of screen RAM per line. If the operating system expects a line to take 40 bytes and in the modified list a line takes only 20, then the data below this line will be shifted half way across the screen.

There are two ways of avoiding problems with this. First you can use "dirty programming" and design your new lines in groups of lines which add to make the correct number of bytes - see examples later.

The other way is to avoid using the operating system for Drawto, Plot or Print commands and poke directly to screen memory.

If you are to use Basic commands such as Plot, Drawto or Print on the screen you may need to fool the OS into thinking it is drawing on the correct screen.

This is done by poking location 87 (\$57) with Basic graphics mode of the line involved.

Second is the problem of Basic

TWAUG NEWSLETTER

GRAPHICS - DISPLAY LIST

checking each command to check that it is in the range allowed by the graphics mode it thinks is in use.

up per line in each mode is therefore needed as it is for the second point above.

There are ways

This can commonly lead to Basic thinking it is going to print off the screen and giving an error when you know full well that it is on the screen.

This is solved by tampering with locations 88 and 89. These contain the location of the top left corner of screen memory and the OS uses these to calculate the legality of a

round those dirty

screen command.

The top corner can be calculated by $PEEK(88)+PEEK(89)*256$. If these locations are poked with the memory location of the start of the line to which you want to plot or print, then the start of this line becomes position 0,0 and therefore within legal range.

Knowing the number of bytes taken

Basic mode	Antic mode	Bytes per line
0	2	40
1	6	20
2	7	20
3	8	10
4	9	10
5	10	20
6	11	20
7	13	40
8	15	40
9	15	40
10	15	40
11	15	40

On to some examples. The simplest way to write a modified list is shown in Program I. This will add two lines for a larger, coloured title to the top of a Graphics 0 screen.

All example Demo programs are on the issue disk, to save you the job of typing them out.

programming techniques

TWAUG NEWSLETTER

GRAPHICS - DISPLAY LIST

It works, but again it is dirty programming. The maximum number of scan lines allowed in a display list is usually 192. This display list is more than 192 scan lines long.

In reality Antic can cope with slightly more lines than the theoretical maximum. I have found that an extra 24 usually is stable, but more than this and the screen will roll.

See last issue's article for a table of the number of scan lines for each mode line.

A better programming technique would be to calculate the number of scan lines being used and make sure that the total is 192 or less. This will usually involve moving the end of the display list and rewriting it as in Program II.

As can be seen, the end of the display list is indicated by a number 65-\$41. The two numbers following this are the location of the start of the display list in the order Low Byte, High Byte. Therefore the first number can be found by PEEK(560) and the second by PEEK(561), as these should be the same.

The third way is to create your own list from scratch. This is how virtually all machine code programs get their display and one of the reasons that

they can be so spectacular.

If you avoid using the OS to draw to the screen then many of the limitations of custom display lists also disappear. However the other side of the coin is that the OS no longer does the hard work for you and the programming becomes more difficult.

Program III demonstrates both these points but to keep it short does not do justice to the capabilities of your Atari.

As I mentioned previously, Graphics modes 12-15 are only directly available on the XL and XE Ataris. However, all the machines are capable of displaying these modes.

Many commercial games in fact use Graphics 12, Antic mode 4. The two most useful of these modes, 12 and 15, can be obtained using programs IV and V.

Program VI is just a little bit of lunacy for light relief.

This is a brief overview of custom display lists and gives some idea of how they can improve the appearance of a simple screen.

However, to bring it to life you can use Display List interrupts to achieve numerous special effects.

GRAPHICS - DISPLAY LIST

Don't panic, but interrupts must be in machine code...

An interrupt, in computer terms, is when the computer temporarily stops executing the main program - Basic, machine code or any other language - and executes another program in memory before returning to the original program.

There are several types of interrupt which are useful for different functions and they can be divided into two types.

NMI - non-maskable interrupts - cannot be disabled by the 6502 processor and include vertical blank interrupts - VBI - display list interrupts - DLI - and Reset.

The VBI occurs during the screen blank after drawing one screen and before starting the next. These occur every 50th of a second.

A DLI can occur after each line is drawn on the screen and takes place in the small delay between drawing each line on the screen.

The other type of interrupts are called IRQ - interrupt request. These are maskable, which means they can be disabled by the 6502 processor.

There are several timer interrupts, peripheral and serial bus input/output interrupts, the Break key and 6502 Break instructions.

NMIs are handled by the Antic chip while IRQs are handled by the Pokey and PIA chips.

But we are only interested in DLIs for now. A DLI can occur every time a line is drawn on the screen.

Therefore, because it enables you to have a small program running each time a DLI occurs, it means you can change various parameters as the screen is drawn.

The result of this is that you can change, for example, the colour of any of the registers part way down the screen once or many times.

This allows many more colours to be displayed at once. Other possible uses are to change the character set in use part way down, change sound

GRAPHICS - DISPLAY LIST

or music, move players or split players several times, fine scrolling in different directions - such as in Frogger - and different screen widths. Any of these without interfering with your main program at all.

All this sounds too good to be true and there has to be a drawback. Well, if you are purely a Basic programmer there is. The interrupts must be in machine code.

However, don't panic. There is no reason why you cannot use DLI routines from other sources in your own program because using them is very easy.

Let's begin by looking at the routine itself,

should you decide to try writing your own. The first point is that timing is critical. Only a relatively short amount of time is available in a DLI, so the routine must be short.

It must start by storing any registers it uses - A,X an Y - on the stack, as the main program will

require these back after the interrupt and it must restore them at the end of the routine.

Also note that many locations for colour or other functions have two locations, the hardware register and the shadow register. In these cases you would normally POKE to the shadow register and the OS transfers the number to the hardware register during the vertical blank.

Any registers changed by the routine should be the

hardware registers, not the shadow registers used in Basic.

For example, to change the background colour in Basic you would POKE 712 with a number. This is the shadow of the hardware colour register for the background - 53274.

However, if you POKEd directly to this the operating system would reset it to the value in 712 during the VBI.

This can be used to your advantage in that any colour change in a DLI will be reset to normal

each time the screen is redrawn, thus keeping the change just below the DLI.

It's not at

all rude to

interrupt...

GRAPHICS - DISPLAY LIST

Any colour change on a line will not occur in a constant position on that line. This can be overcome by storing the value in WSYNC - 54282 - before the colour register. This delays the interrupt until the end of the line making a neat boundary.

The best way to write your own DLI is to examine the ones in the examples in this article first.

Having got your DLI routine - written or borrowed - it is used as follows.

First modify the display list. This involves changing each display list line where DLI is required by setting bit 7. In other words add 128 - \$80 - to the line. this can be a single line

For example, Page 6 is 1536. Here machine code is safe from Basic and most other operations. POKE in the code from 1536 and POKE 512 and 513 with 0 and 6 respectively - $6 * 256 + 0 = 1536$. Finally POKE 54286 with 192 to enable the DLI to occur.

Note there is only one address for DLI routines and if more than one is to be used then the interrupts must change the address as each is executed.

If this seems complicated, it isn't really, as the following examples show.

First you can change colours part way down the screen. Program 1

Decimal	Hex	Disassembly	
72	\$48	PHA	;Save accumulator on to stack
169,22	\$A9,16	LDA \$16	;Load accumulator with change in colour
141,10,212	\$8D,0A,D4	STA \$D40A	;Store in WSYNC
141,26,208	\$8D,1A,D0	STA \$D01A	;Store in background hardware register
104	\$68	PLA	;Restore accumulator value
64	\$40	TIR	;Return from interrupt

Figure 1: Disassembly of DLI or several lines.

Next POKE 512 and 513 - \$200, \$201 - with the low and high byte values of the location of the DLI machine code routine.

changes the background colour half way down a Graphics 2 screen. Note that more than one colour register could have been changed.

The machine code for the DLI in this program disassembles as shown in

GRAPHICS - DISPLAY LIST

Figure 1. This shows a single change of colour. The colour can be changed several times down the screen.

In Program II we increase the colour of the foreground by 4 on each line to produce a multicoloured design.

The next step from this is to make the colours rotate. Program III produces a gradual change in colour rotating up the screen as seen in numerous Atari demos.

Spectacular isn't it? And so easy.

Graphics 0 is rather plain and boring normally. By using a DLI on each line a spectacular multicolour - 48-colour - Graphics 0 screen can be made as in Program IV. You can alter the colours by changing the Data statements in line 210 and 230.

All these examples involve altering colours on screen, but other uses are just as easy. As mentioned before, you can change character set part way down screen.

You can run sound effects or music in DLIs. You can split players part way down the screen to make it appear as though there are more than eight player-missiles. Finally you can use DLIs to get horizontal scrolling in different directions on different lines.

Remember, as far as the Atari is concerned, it's not at all rude to interrupt.

Missing programs

Where are all those programs mentioned in the Display List and Interrupt articles? Do not fear, I haven't missed them off, all these programs are on this issue disk, to save you the task of typing them out yourself.

The first part of the Display List has 6 Demo programs and I named them as DLIDEMO.001 to DLIDEMO.006.

The second part from page 13 mentions 4 programs and I named these programs, EXAMPLE.1 to EXAMPLE.4.

BIT WISE

MIKE BIBBY continues his series of articles aimed at lifting the veil of mystery cloaking the fundamentals of the Atari's workings

We have seen that we can code our numbers in ways other than our usual denary, or decimal, system. We also looked at a way of coding known as the binary system, which uses the digits 0 to 1 to represent any number - unlike the denary system which uses the digits 0 to 9.

To distinguish the two systems, we decided to prefix binary numbers with the symbol "%".

The number "one hundred and sixty two" is encoded in each system like this:

In denary,

$$162=100+60+2$$

In binary,

$$\begin{array}{r} 128 \ 64 \ 32 \ 16 \ 8 \ 4 \ 2 \ 1 \\ \% \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \\ = 128+32+2 \end{array}$$

Each column in the binary system, known as a "bit", contains either a one or a zero.

Although the binary representation of a number is rather cumbersome to write, this simple two-state system is easily represented by electrical circuits - which are either ON or OFF.

We saw that the computer handles bits in groups of eight at a time. Such a group is called a byte. Thus a byte contains eight bits labelled, somewhat perversely, bits 0 to 7. (See Figure 1.)

Bit 0, as you can see, is the "1" column. As this is the smallest value bit we say that bit 0 is the least significant bit (LSB). Bit 7, the "128" column, is called the most significant bit (MSB).

The reason for using the numbers 0 to 7 to label the bits instead of the more logical 1 to 8 has to do with powers, a subject you almost certainly covered at school.

$$2 \text{ to the power } 2 \text{ is } 2*2 = 4$$

$$2 \text{ to the power } 3 \text{ is } 2*2*2 = 8$$

$$2 \text{ to the power } 4 \text{ is } 2*2*2*2 = 16$$

and so on. "2 to the power 8" would be eight twos all multiplied together.

Notice that as the power of two

TWAUG NEWSLETTER

BIT WISE

increase - that is, as we multiply more twos together - the answers are doubling, just as our column or bit values do.

Also, 2 to the power of 2 is 4, the value of bit 2, while 2 to the power of 3 is 8, the value of bit 3. It shouldn't come as any surprise to you to find that 2 to the power of 7 is 128, the value of bit 7.

You can verify this on the Atari by using the symbol "^^" which stands for "to the power of". It shares a key with the * sign.

Try:

PRINT 2^4

PRINT 2^7

Be sure to try 2^1, which will show you why bit 1 has the value of 2.

Also try 2^0. The answer may surprise you. The fact is that any number to the power 0 is 1!

Hence bit zero has the column value of one. Figure 11 illustrates this.

Look at this sum:

$$\begin{array}{r} \% 1 \\ + \% 1 \\ \hline \% 10 \end{array}$$

If you think about it, that is correct, since the sum adds one and one and the answer %10 is binary for two.

One way of relating this to our usual way of doing sums is to say that we carry when we get to two, instead of ten, as we do in our normal denary sums.

Another way to look at it is that we have to carry when we get to two because we aren't allowed to use the digit "2".

If you remember, last issue we had a rule forbidding two "coins" of the same value.

Try this sum:

$$\begin{array}{r} 4 \ 2 \ 1 \\ \% \ 1 \ 1 \ \text{in} \qquad 3 \\ + \% \ 1 \ 0 \ \text{denary} \quad + 2 \\ \hline \% 1 \ 0 \ 1 \qquad \qquad \underline{5} \end{array}$$

Here we carry from the second column to the third.

Addition is not very hard at all - just make sure that you always "put 0 down and carry 1" when you get two.

If you get a three then "carry one for two and put one down".

For example:

$$\begin{array}{r} 8 \ 4 \ 2 \ 1 \\ \% \ 1 \ 1 \ 1 \ \text{in} \qquad 7 \\ + \% \qquad \qquad 1 \ 1 \ \text{decimal} \quad + 3 \\ \hline \% \ 1 \ 0 \ 1 \ 0 \qquad \qquad \underline{10} \end{array}$$

BIT WISE

Subtraction is a little more complicated and depends on whether you borrow or decompose! The latter phrase doesn't describe the current economic climat. It's just that there are two schools of thought on the way subtraction should be taught - the borrowers and the decomposers.

Fortunately, we can ignore binary subtraction since we can manage without it - as does the microprocessor inside your machine.

If you want to do some binary subtraction it is straightforward enough provided that you remember that it is two you're borrowing or taking, not ten. Figure III illustrates the progress - without any attempt to explain it.

Before we leave the realm of simple sums, look what happens if we shift everything in a binary number over to the left, putting a zero into bit 0, which would be left vacant otherwise.

For example:

8 4 2 1

% 1 0 1 which is 5

becomes

8 4 2 1

% 1 0 1 0 which is 10

This shifting to the left doubles the

number automatically. This isn't too hard to visualise, because the value of each bit is transferred to the next higher bit, which is of course double in value - so the end result is that the whole number is doubled in value.

Similarly, we can do the binary equivalent of 12 divided by 2 by shifting to the right.

For example:

8 4 2 1

% 1 1 0 0 which is 12

becomes

8 4 2 1

% 1 1 0 which is 6

and, of course, 12 divided by 2 gives you 6.

As each bit is moved to the right, it occupies a column exactly one half lower in value, thus the sum total of all the bits is one half lower. Note the original bit 0 has disappeared altogether. The loss of this bit can cause some inaccuracies. After all, if it were 1, when it's halved it should contribute 0.5 to the answer. As it is, it's ignored.

For example, if we try to do 13 divided by 2 in binary by shifting each bit right, the equivalent of 13,

TWAUG NEWSLETTER

BIT WISE

Bit number	7	6	5	4	3	2	1	0
	1	0	0	0	1	1	0	1
Bit value	128	64	32	16	8	4	2	1

Figure I: The bit pattern for 141

Bit number	7	6	5	4	3	2	1	0
Bit value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
	1	1	0	0	1	1	0	0

Figure II: The bit pattern for 204

```

      8 4 2 1
%    1 1 0 1
becomes
      8 4 2 1
%    1 1 0
    
```

which is 6 in decimal. Now 13 divided by 2 gives 6.5, not 6, so what happened to the 0.5?

Well, when we shifted over the original bit 0 (which had the value 1), we shifted it "out of the byte". If you like, it dropped off the end and

doesn't appear in the answer.

Of course, it's this disappearing 1 that would give us the missing 0.5 when it's halved.

This sort of division, where you're only concerned with the wholes in the answer and ignore any remainders, or decimal parts, is called integer division.

Well, that's enough binary for this issue, now let's look at hexadecimal.

TWAUG NEWSLETTER

BIT WISE

practice - each group of four bits is called a "nybble", would you believe?

It's not too hard to see that the biggest number you can represent in a nybble is 15, and the smallest is 0.

%1111 and %0000

respectively. After all, you've only got four bits to play with!

So we can split up our byte into two nybbles of four bits each. Now when we split up a binary number in this manner we call the left-hand nybble the most significant nybble (MSN) and the right-hand nybble the least significant nybble (LSN).

We have already created one new number system - the binary system.

%10111011 = 187

%10101101 = 173

%10001111 = 151

%111110110 = 246

Table I

Let's design another one that combines the advantages of the denary system with those of the binary. That is, it will be easy to read and write, yet will still allow us to perceive the binary manner in which the machine handles things.

The system we want is called hexadecimal. This consists of using our standard digits 0 to 9 for the numbers zero to nine respectively, and the letter A to F for the numbers 10 to 15. In this way it allows us to code the numbers available in a nybble (that is, 0 to 15) with just one digit. This digit will be in the range 0 to 9 or A to F.

It may take a while to adjust to the idea of using letters of the alphabet for numbers, but it soon becomes second nature. You just have to get used to counting:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Remember, there are B people in a cricket team, D in a rugby league team and F in a rugby union team. There are C months in a year and E days in a fortnight.

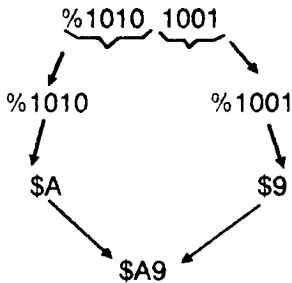
Now just as we prefix all our binary numbers with %, we prefix our hexadecimal numbers with \$, to avoid confusion. So \$F means 15, while \$9 means 9.

Studying Table II will really pay dividends - I suggest you practise writing down bit patterns of nybbles and their hexadecimal equivalents until it becomes second nature.

Given that we can encode a nybble in one hexadecimal digit and that a

BIT WISE

byte consists of two nybbles, it should readily be apparent that we can encode a byte as two hexadecimal digits side by side, for example:

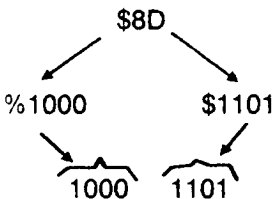


That is:

$$\%10101001 = \$A9 = 169$$

You just split the byte up into two nybbles - a left hand and a right hand nybble, encode each as a hexadecimal number, then put the two side by side.

You can go from hexadecimal to binary just as easily:



That is:

$$\$8D = \%10001101 = 141$$

Although you have probably never thought of it in these terms, you are

well aware that the value a digit

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1010	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table II

represents depends on the column it is in. The number 230 is not as large as 320, though both numbers contain the same digits.

In hexadecimal coding too the column a digit is in is important. For example, \$10 is far greater than \$01. In binary each column is worth twice the preceding one. In denary, our usual number system, each column is worth 10 times the preceding one. In hexadecimal, each column is worth 16 times the preceding one.

Believe it or not, the columns in a

BIT WISE

four digit hexadecimal number, from greatest to least, are worth 4096, 256, 16 and 1 respectively.

This means that:

$$\text{\$1101} = 4096 + 256 + 1 = 4353$$

For the moment let's concentrate on the two digit, that is, two column, hexadecimal number, as these are all we need to store our bytes in. In this case the left-hand column is the "sixteens" column, the right-hand the units column.

So:

16 1

$$\text{\$ 2 1} = 2 * 16 + 1 = 33$$

16 1

$$\text{\$ 2 D} = 2 * 16 + 13 = 45$$

16 1

$$\text{\$ 8 0} = 8 * 16 + 0 = 128$$

16 1

$$\text{\$ C 0} = 12 * 16 + 0 = 192$$

To translate a two digit hexadecimal number into denary simply multiply the number in the left-hand column by 16 and add it to the number in the right-hand column - remembering to translate A to F if necessary.

The second column has the value 16 since the first column can only handle numbers up to 15 (\$) - the

largest you can fit into a nybble (%1111). After 15, you have to use a second column for 16, that is \$10.

Just as in denary, we "carry" at 10 since the largest value our columns can handle is 9, so in hexadecimal we carry at 16, since the largest our columns can handle is 15 (\$F).

It is the fact that we carry at 16 that gives this number system its name "hexadecimal" - here "hex" stands for 6, "decimal" for ten.

$$\text{"Hexadecimal"} = 6 + 10 = 16.$$

Given a second column \$10, as we have seen is 16, 17 will be \$11, while \$12 is 18 and so on until we reach 31, which is \$1F.

We have then run out of legal digits for the units column, so if we want to go on to 32 we had better give ourselves another 16, and set the unit column back to zero, that is \$20.

Another way of looking at the second column is that it comes from the most significant nybble. To turn the least significant nybble into the most significant nybble, we have to shift it over to the left four times.

If you cast your mind back to near the beginning of this article, this is equivalent to multiplying it by two four times in succession, that is

BIT WISE

$$2 \times 2 \times 2 \times 2 = 16.$$

This is why a hexadecimal digit representing the most significant nybble is 16 times larger than the same digit representing the least significant nybble.

The largest number you can store in a two-digit hexadecimal number is \$FF = $15 \times 16 + 15 = 255$. This is, of course, the same as the largest number we could store in a binary byte - we often refer to a two digit hexadecimal number simply as a byte.

To obtain the hexadecimal equivalent of a positive integer (whole number) less than 256, we divide it by 16. The quotient is the left-hand digit, the remainder the right-hand, translating into A to F where necessary.

For example:

$$174 \div 16 = 10 \text{ R } 14$$

That is:

$$\text{\$A R \text{\$E}}$$

$$\text{Hence } 174 = \text{\$AE}$$

Anyway, here's a program that will convert from denary to hexadecimal for you. The workings shouldn't be too hard to follow.

Once you've understood it, how about writing one that will convert from

hexadecimal to denary?

That's all for now. In next issue we'll be looking at ways of combining binary numbers.

```

10 REM DENARY TO HEX
20 DIM HEX$(16),ANSWER$(2)
30 HEX$="0123456789ABCDEF"
40 PRINT "DENARY ( 0 - 255 )";
50 INPUT NUMBER
60 IF NUMBER<>INT(NUMBER) OR
NUMBER<0 OR NUMBER>255
THEN GOTO 40
70 HI=INT(NUMBER/16)
80 LO=NUMBER-HI*16
90 ANSWER$=HEX$(HI+1)
100 ANSWER$(2)=HEX$(LO+1)
110 PRINT ANSWER$
120 PRINT
130 GOTO 40

```

Program I

MEMORY MONITOR

Here is a Hexadecimal/Ascii memory dump utility written entirely in machine code. Although memory dump utilities are fairly common, this little routine is far from standard. The program is written in machine code and resides in Page 6, that is the area known as 1536 to 1792 in decimal or \$600 to \$700 in hexadecimal. This area of memory is not used by BASIC or the operating system, so is free for utilities such as this. The machine code is unaffected by LOAD, SAVE or NEW. This makes it possible to run Basic programs at the same time and examine how they are stored in memory. The operating system can be examined and the system variables can also be monitored.

The routine displays 192 memory locations on a Graphics 0 screen in both Hexadecimal and Ascii. When it has completed this task it goes back and displays the same 192 locations again. This is repeated until one of the cursor keys are pressed.

You might imagine that printing 192 memory locations in hex and Ascii would take a long time. However, remember that this is machine code. The routine updates the screen 30 or

40 times a second. The advantage of constantly displaying the same area of memory over and over again is that locations that change are instantly updated and can be seen quite easily. The system clock, for instance, can be seen rapidly ticking away in page zero.

To monitor any section of memory enter:

`X=USR(1536,n)`

where n is the address from which to start displaying. To return to Basic press the space bar.

Pressing the cursor up key (No need to hold CONTROL) will increment the start address by 8 and the display scrolls up. Pressing the cursor down key decrements the start address by 8, scrolling the screen down. the screen continues scrolling until any other key is pressed. The Return key is the most convenient key to stop.

The scrolling is very fast. The routine takes about three seconds to whizz through 1k of memory, so its very easy to move backwards or forwards through RAM or ROM. There's quite a lot to be learned by scanning the memory using the monitor. Try looking at Page zero first. The clock can be seen ticking away at

MEMORY MONITOR

\$12-\$14. Every time \$14 reaches 0 \$4E is incremented. Press a key and \$4E is reset to zero.

Page 1 is the 6502 stack. This can be seen to be flashing quite rapidly as data is pushed on and pulled off.

\$22B is interesting. When a key is pressed the delay before auto repeat comes into operation is placed here. This counts down to zero, and if the key is still being pressed the auto repeat delay is placed here. Again this counts down to zero.

The Basic line buffer is around \$580. You can see here what you've just typed in.

Basic programs are stored at around \$1F30 on my Atari 800 XL with disc drive. It may be different on other Ataris.

The program Memory Monitor is on the issue disk. This article is also included, in case someone wants to make a hard copy of it instead of using the mag all the time.

SALE

Black Box with the enhanced Floppy Board, 90 megabyte Hard Drive, 4 PBI Drives plus Modem --- all for £400 or near offer.

Contact: John Matthewson on
0191 - 262 6897

This is a bargain, the cost of the Black Box with enhanced Floppy Board alone, when purchased was about £300 plus.

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

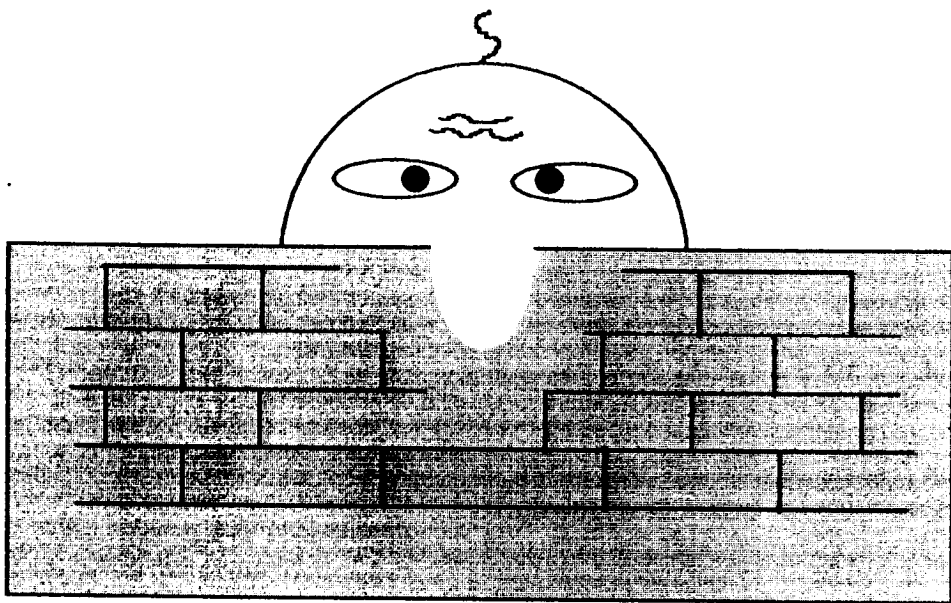
Contact:
Alan Turnbull on
01670 - 822 492

These computers make the ideal inexpensive Christmas gift.

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

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

TWAUG NEWSLETTER



W O T ?

N O M A I L ?



DISK CONTENT

This month issue disk contains a number of demos, those demos are from the article GRAPHICS - DISPLAY LIST and INTERRUPTS, starting at page 10 to page 17. I've named the demos DLIDEMO.001 to DLIDEMO.006 and for the second part, covering the interrupts article, the demo programs are called EXAMPLE.1 to EXAMPLE.4. I cannot say much about them, the explanation for them is in the article.

The program Memory Monitor, again the article on page 26 tells you all about it. I have used that program not long after I started computing and I was fascinated with the display. This is a program for the beginner to look into the memory of his computer.

In issue 23 I mentioned on page 35 that the Routines for Coding Capers written by Tomohawk would be included in issue 24 and these are on Side A of this issue disk. You must dig out issue 23 for the explanation.

On Side B is a game called ROBO MASH, I never played this game and therefore I cannot say anything about it.

I have some bad news to report, no, no! I am not leaving TWAUG, but Kevin Cooke is. I received the Game Review, it was very late, but I still waited to include it in this issue, unfortunately the disk was corrupted and I am unable to rescue anything from it. "PANIC!" It sure was, I was so upset I could have sat down and cried my eyes out. Never before, not since I've started doing this newsletter, was I feeling so low as I did when I knew I couldn't include the review of Kevin. I was short of material as well and so in desperation I had to cut down on the size of the issue. I apologize for that, John wanted the master copy, as the deadline had already past for the publication.

Now Kevin Cooke also informed me, that due to lack of spare time he now has, that this was the last of his reviews and I can't even include it. So we are now looking for a replacement reviewer. Anyone willing to take on that job? The payment is free issues as long as you are willing to write for us, or PD library disks, or like me voluntary and unpaid. Get in touch! MAX

TWAUG NEWSLETTER

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

Merry Christmas

O.H.A.U.G.

The OL'HACKERS ATARI USER GROUP INC.

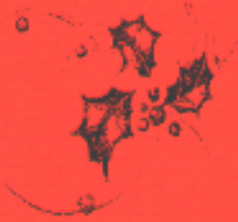
O.H.A.U.G. is an all 8-bit user group in the STATE of NEW YORK.

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 Fetzter
O.H.A.U.G.
Secretary & Treasurer
22 Monaco Avenue
Elmont, N.Y. 11003
USA



Season's
Greetings



Greetings

Staff

of

T.W.A.U.G.

would like to wish

our

Subscribers

a Merry Christmas

and

a Happy New Year



TWAUG NEWSLETTER

MICRO-DISCOUNT

Offers
The complete Mail Order
service for Atari 8 Bit



XL/XE users
4th September 1995

