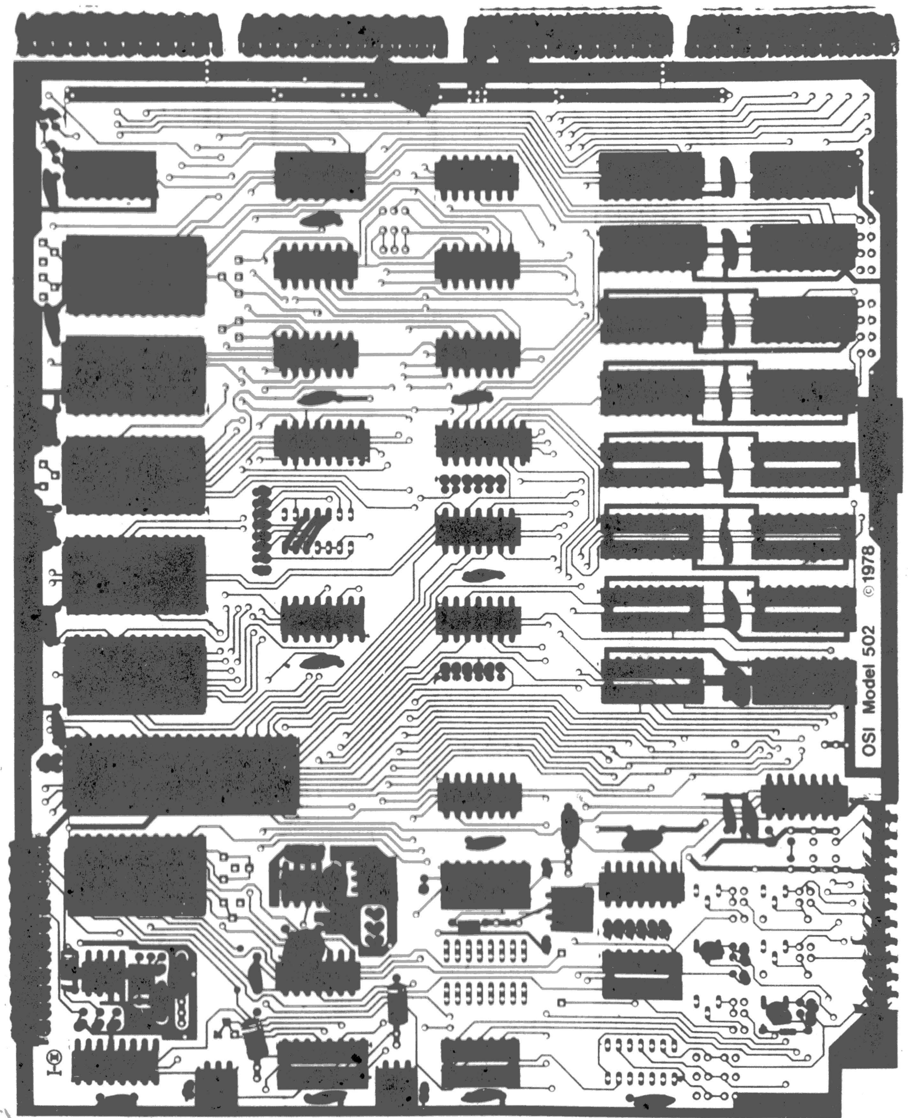
OSI UK User Group Newsletter

Vol.1 No.2 March 1980



502 CPU board, as used on C2 series

Documentation □ BASIC storage formats □ User Group notes
Superboard display conversion □ and much other information

Editorial

Welcome back! This second issue of the User Group Newsletter finds the Group expanding every bit as rapidly as we had hoped, and with some superb articles and smaller pieces coming in from members — the article on modifying the display of the Superboard being the most important one this issue. What are *you* doing with your kit? Let us know — to help everyone who uses OSI equipment.

User Group matters

We're finding that dealers as well as users want to know about User Group membership — partly as part of their technical back-up, partly because many are willing to offer special deals to members. One dealer has asked us to issue membership numbers, so that he can offer software at a reduced rate to members; and another is offering a superb deal on a matrix printer — see the User Group Notes at the end of this issue. But you have to be a member first... Membership is £5.00 per year, mainly to cover the cost of production and distribution of this quarterly Newsletter; cheques should made payable to *OSI UK User Group* and sent to the Group's London address, *12 Bennerley Road, London SW11 6DS.* Tell us which issue you want your membership/subscription to start from; and give us at least two weeks to turn subscriptions and letters round, as we're only doing this in whatever 'spare' time we can invent! Technical queries and the like should also be sent to George Chkiantz and Richard Elen at the London address.

Next issue

This depends on you! We'll be continuing the Aardvark BASIC series with a description of the main BASIC program processing loop; but we've little else definite yet. So send in articles, tapes or listings; let us know what you're doing. These should be sent to me (Tom Graves) at *19a West End, Street, Somerset BA16 0LQ*, and should arrive here by *mid-May* to be in time for the next issue, which will be issued in mid-June.

Documentation corner

As you're no doubt only too well aware, OSI's attention to detail in its documentation for its BASIC-in-ROM machines leaves much to be desired. As information comes in we'll publish it in this 'corner', as a regular column.

CLEARly in error

Richard Elen writes: Have you ever noticed that our good old under-described OSI/Microsoft BASIC has some features that are not discussed in any depth in the BASIC Manual supplied with the gear? Sometimes things aren't even mentioned.

One such feature is commonly available on other people's BASICs but never a word is spoken about it in OSI's documentation, other than a mention in the list of available commands on the back page. This is the CLEAR function.

If you've ever run out of memory with a load of strings or arrays, or succeeded in returning the dreaded 'B-Splurge Error' halfway through your fiendishly complicated data-handling program which uses all the names for strings known to Man and a few more, you might have wondered about this one. Other people's BASICs often utilise the CLEAR instruction to reserve string (or sometimes array) space in memory for BASIC use. For example, on some machines, the instruction CLEAR 4500 will allocate a nice wodge of memory to strings (one presumes 4500 bytes in this case, but of course you can write CLEAR *n*, where *n* is a suitable amount of memory). We don't appear, at first sight, to have such a useful command available on our machines.

But wait a moment. It is true that if you type:

```
10 CLEAR 5000
```

in your program, it will promptly return a SN ERROR, and similarly, attempts to enter shorthand forms, as on some BASIC variants, like CLR 3000 or whatever, you'll get the same response. But now try this:

```
10 CLEAR
```

— just the simple command, CLEAR, no following digits or anything. This will produce an ominous *nothing* from your machine, no SN ERROR, no B-Splurge message, no nothing. In other words, the machine has accepted it. In fact the list of tokens (pairs of hex digits which are stored in a look-up table which the machine uses to compress BASIC keywords into 1-byte mnemonics for its own use) includes one for CLEAR. The command, therefore, definitely exists; the machine accepts it and knows what to do with it. But days of poring over a disassembled listing of OSI BASIC still leaves me unenlightened. How is this CLEAR instruction designed to be used? Does it allocate some string and/or array space? How much? Is it a fixed amount of memory or is it automatically set by, say, the size of available RAM? Can you in fact add something to the CLEAR instruction to clear a space for a specific string like maybe CLEAR A$ or CLEAR A(500)? Does it reset the variables? Trouble is, I simply don't know. Do you?

Editor: I spent a fair amount of time chasing this after Richard brought it to my attention. CLEAR *does* reset all variables, strings *and* array dimensions: variables return to 0, strings are returned to LEN 0, and arrays, if called are returned to the undimensioned default limit of 10. What I haven't yet had time to check is whether after calling CLEAR, a DD ERROR (double dimension, not Double Diamond!) arises on re-dimensioning any array. It seems that CLEAR is intended to be used rather in the same manner as RESTORE; but it seems a little extreme in that it clears *everything*. CLEAR A producesa SN ERROR, as does CLEAR A$; and it doesn't seem to allocate any special space to strings, although there is a comment in the 'manual' that strings are limited to 255 bytes, which may do away with any need for that function for CLEAR.

In passing, I seem to have discovered the meaning of 'the dreaded B-Splurge Error'. It arises if you call a subscript outside the dimensioned limits for the array: in other words if A(X) is undimensioned, it defaults to 10, so calling A(12) will produce that 'B-Splurge Error' statement. The B-Splurge appears in the ROM listing (in the usual garbled form) as BS — but what the letters are supposed to stand for I haven't a clue! The ROM listing shows 17 error codes, as does the list in the manual; the one that appears to be missing (so to speak) is the OS (out of string space) error, which is made redundant by a combination of OM (out of memory), BS and the automatic limiting of the length of strings.

ON X...GOSUB

Like CLEAR, ON X...GOSUB is included in the list on the back of the manual. It certainly does exist: I used it as the key part of my *Tune Chaser* program. It is identical to ON X...GOTO, but calls subroutines rather than jumping to one of the given sequence of line numbers.

SGN(X)

Another bloomer on OSI's part, this one. The manual claims that SGN(X) returns 0 if X<=0 rather than (as with most BASICs) 0 if X=0 and -1 if X<0. OSI's documentation *is* wrong:

SGN(X) is 1 if X is greater than 0
SGN(X) is 0 if X equals 0
SGN(X) is -1 if X is less than 0

USR(X)

We dealt with USR(X) in some detail in the last issue, but there is one more point worth making. That is that *AE05*, the INVAR routine, transfers the processed value of (X) into *AE, AF* as a *fifteen-bit* signed integer (*not* a sixteen bit number) into those locations, and with the number laid out as a double-byte number (high byte first) rather than as a two-byte address (low byte first). If you want to use USR(X) to pick up an address via *AE05*, you have to trick it somewhat if you want to give it an address higher than $7FFF_{16}$ — which you will do if you want to place something on the screen using a STA ($AE), Y machine-code routine. You'll also need to swap the two bytes round, to make them address-format rather than number-format. A sequence I've used is JSR AE05, LDY 0, LDX $AF, LDA $AE, ORA 80_{16}, STA $AF, STX $AE, LDA (whatever), STA ($AE),Y, RTS. The screen addresses (for a C2) become 20480_{10} to 22528_{10} (or to 22400_{10} to the text base-line) if you want to use the routine to put something on the screen. If you don't do this, *AE05* tries to make out that the screen addresses are really -20480 to -22528 (because it treats the top-most bit as a minus-sign rather than as part of an address) and, not surprisingly, screws up the whole system, ending with an FC ERROR statement. Using this routine is only minutely faster than POKE, incidentally, because of the laborious sequence that *AE05* goes through to process the value of (X); can anyone work out a quicker way of transferring selected screen values?

USR buffs might like to know that there is an effective 'spare' ¾K of memory outside of the program space, so that memory does not have to be allocated for shortish machine-code routines through either the 'official' method of answering 'MEMORY SIZE' with a lower-than-total-memory figure, or the more practical one of POKEing 133_{10} and 134_{10} with the low and high parts of your 'new' top of memory. (If you place a machine-code routine up the top of memory, but don't reserve space for it, BASIC will overwrite it with the first string it has to store from your program.) The 'spare' space runs from 0240_{16} to around $02F0_{16}$ or later (the top may be used by some rarely-used BASIC pointers, as far as I can tell); if you use 0240_{16} the start-POKEs for the USR routine are POKE 11, 64: POKE 12, 2. Beware, though — some utility programs, such as Sirius Cybernetics' *C2 Screen Editor*, use this space while they are running; so don't try to run your USR routine when you are running that program co-resident with it, or the editor (or whatever) will be thoroughly scrambled!

Recovery from accidental Cold-start

Recovery from coldstart is possible if you answer "MEMORY SIZE?" with a number instead of <RET>. (Once you hit RETURN, BASIC fills the memory with test bytes until it doesn't get them back to see how much memory there is. That means your program is completely and irrevocably overwritten.) The easiest way is to go into the ROM monitor before you coldstart and find and copy the contents of locations 007B, 7C and 0301, 02. Then coldstart, entering your memory size (i.e. 4096 for a 4K machine, etc.) and after BASIC comes up, go back to the monitor and replace 7B, 7C (the end of program/beginning of variables pointer) and 0301, 02 (the pointer from the first BASIC statement to the second, which will be set to zeros by coldstarting — though the rest of the program is still there). If you have already coldstarted, look for the first zero byte after loc 0305, and put an address one higher than that zero in 0301, 02 (low order byte first; the contents of 0302 will be 03 always, unless you have hand-manufactured a very unusual BASIC program). The program will now list, but will wipe itself out if you try to run it. (Variables will overwrite the beginning of the program.) List the program, immediately use the monitor to find the contents of 00AA, AB and put those contents into 7B, 7C. Everything should then be back to normal. (In fact, immediately after listing any line, locations AA, AB will contain the address of the pointer of the next BASIC statement — or of the beginning of variable space if the last line of the program is listed.)

Machine-code and BASIC

Another of the 'missing' sections of the firmware in C1s and C2s is any machine-code SAVE routine. In the next issue we'll have one of the many machine-code routines that users have developed to do the job; for now we have a BASIC version from *Aardvark* (who also gave us the cold-start recovery sequence and the routine to load machine-code with a BASIC tape). They claim it SAVEs machine-code very nearly as fast as a machine-language routine; but there may be timing problems, as users of this routine have told us that it is a little erratic, getting out of phase after around 150–200 bytes! Try it, anyway; it's better than the nothing that OSI supply for free with their machines...

```
 10 SAVE: POKE 15,255
 20 A1=   (fill in start address, decimal)
 30 A2=   (fill in end address, decimal)
 40 ACIA=64512   (61440 for 1P's)
 50 ?".HHHH/";   (HHHH is start addr in hex)
 60 FOR A=A1 TO A2
 70 D=PEEK(A)
 80 H=INT(D/16)
 90 L=D-16*H
100 IF H>9 THEN H=H+7
110 IF L>9 THEN L=L+7
120 ?CHR$(H+48) CHR$(L+48);
130 WAIT ACIA,2
140 POKE ACIA+1,13
150 NEXT
160 ?".FE00G"
```

If you would like to be able to LOAD a BASIC tape and then have it automatically continue and load a machine language tape with the monitor, here is one way to prepare a tape that does that: Type: SAVE <RET> LIST (turn recorder on) <RET> (stop tape when done) ?"POKE 251,1: POKE 11,67: POKE 12,254: X=USR(X) (restart recorder) <RET> (stop tape when done). Now put the machine language you want on the tape. When you LOAD the tape, it will load the BASIC program, switch to monitor mode (without clearing the screen) and load the last part of the tape.

Disc system notes

One thing which worried us about the last issue was that all the information we had was for BASIC-in-ROM machines. Admittedly these are the most common ones, but we do have members with larger systems — we want to give you as much information as we can too! Just before this issue went to press we were able to collect the following, the first of what we hope will be, to quote our correspondent who supplied the information, 'a continuing series of hints, fixes and information from our spy in OSI'.

Date print-out and the real-time clock
OSI equipment running OS-65U has three locations assigned for the date. These are:

	Level I address	Level III address
Day	24569	55922
Month	24570	55923
Year	24571	55924

OS-65U Level III has access to a real-time clock at the locations specified and hence will not need anything to be added to BEXEC*. For Level I the following routine should be added to BEXEC* so as to obtain the date on boot-up and access it for printing later.

```
220  REM Routine to get date
221  REM
222  ?:INPUT"Please enter the date in the following format — Day, Month, Year";
DA$,MO$,YR$:?:?
223  DA=VAL(DA$): MO=VAL(MO$): YR=VAL(YR$)
224  IF DA<1 OR DA>31 OR MO<1 OR MO>12 GOTO 228
225  IF YR<1980 OR YR<>INT(YR) OR DA<>INT(DA) OR MO<>INT(MO)
GOTO 228
226  POKE 24569,DA: POKE 24570,MO: POKE 24571, YR: GOTO 229
228  ?"Please use integer (whole number) values for the day, month and year":
GOTO 222
229  REM
```

To retrieve the date in the form DT$ arranged as DD/MM/YY use the following routine under Level I:

```
A=24569: DT$=RIGHT$(STR$(PEEK(A)+100),2)+"/"
DT$=DT$+RIGHT$(STR$(PEEK(A+1)+100),2)+"/"
DT$=DT$+RIGHT$(STR$(PEEK(A+2)+100),2)
```

Memory access problems: Z-80 and the 520 board
If you're having problems with running the 510 board's Z-80 with certain 520 memory boards, the problem is due to the way that the memories require some set up time between accesses. The original circuitry could access memory (i.e. the R/W line could change from Read to Write) as the chips were being enabled. The modifications to the 510 board described below prevent the devices from being enabled until the R/W transition has occurred.

a) Isolate U34 pin 2 (a 7410)
b) Connect a jumper from U31 pin 8 to U34 pin 2.
c) Connect a jumper from the Z-80 pin 22 to U34 pin 10.
d) Connect a jumper from the Z-80 pin 21 to U34 pin 9.

Boot problems with hard disc systems under OS-65U V1.2
If you find that 65U V1.2 (N MHz) exhibits the following problem the solution below should fix it! Our correspondent doesn't know what it does but the results are good! The problem comes with CD-23 systems (23 Mbyte hard disc) — the floppy disk version of 65U V1.2 won't boot if the CD-23 is powered up. The hard disk version won't boot at all.

Solution: Run "change", "pass"

Select Hex mode, unit A, address offset -2AFD (-2B00 for Hard Disk)

```
Select address 302E
0000302E       3E ? 40
0000302F     4 34 ? /
00003030     C 63 ? 7B
00003031     4 34 ? /
00003032       EF ? 01
00003033     5 35 ? 36
00003034     C 43 ? 4A
00003035     7 37 ? X

OK
```

Input, screen-'print' and graphics for games and the like

As promised in the last issue, we'll devote this section to a number of matters that relate to input to and from the screen, for games and other uses. Many of these notes come from Aardvark's BASIC Notes and their software catalogue — thanks!

Screen clear
Another OSI oversight, especially as there is a perfectly good screen clear routine jumbled up unusably in the midst of ROM monitor code. The 'official' solution is a clumsy and inelegant FOR X=1 TO 30: PRINT: NEXT X or the agonisingly slow 'POKE the whole screen with blanks'. There are several good machine code routines, but these are a little tricky for beginners (I'll get round to including one machine-code screen-clear routine one of these issues!). But Aardvark included in their BASIC Notes a bizarre screen-clear subroutine which works by fiddling BASIC pointers and is absurdly fast, even though it does in fact scroll the screen.

```
10 A=PEEK(129): B=PEEK(130)
20 POKE 129,255: POKE 130,215
30 A$="  (65 blanks)
     "
40 FOR I=1 TO 32: A$=A$+"": NEXT
50 POKE 129,A: POKE 130,B
```

One alteration is necessary if this is to be used as a subroutine with three-digit line numbers. There simply isn't room within the input limit of 72 characters to get a long line number, the string label, equals, quotes *and* 65 blanks all in together. The simplest way out of this is to split the '65 blanks' into two parts: A$=" *(40 blanks)* " *(new line and line number)* AA$=" *(25 blanks)* ": A$=A$+AA$ — leaving A$ as a string of 65 blanks. Because this scrolls the screen, the screen area below the text base-line is untouched; so that still has to be cleared with FOR X=55167 TO 55295: POKE X, 32: NEXT X before the screen really is cleared. But the total time, even on a 1MHz C2 (as for the addresses above) is well under one second — a significant improvement!

Input: simple USR routines

Aardvark comment: everyone has times they want to input something without scrolling the screen. We usually use PEEKs of the keyboard — and still have to do so to run in real time. However, if you are doing a stop and wait for input, use this routine: POKE 11, 0: POKE 12, 253: X=USR(X): P$=CHR$(PEEK(531)). That will input one letter. If you want a number then P=PEEK(531). If you want a word or sentence, add up the PEEKs with A$=A$+P$. By using the 'PRINT AT' routine below, you can print the input to anywhere on the screen and seem to input at any location.

Editor: The USR routine calls the monitor's keyboard routine at $FD00_{16}$, which waits until a key is pressed and returns its ASCII value, parking it at 531_{10} in the process. The trouble with this way of handling input is that if the user hits the wrong kind of key — particularly an alphabetic key when the program expects a number — the program will crash with a TM (type mismatch) error. A way round this is to limit the possibilities by treating *everything* coming in in this way as an ASCII value rather than as a letter or number, and limiting the range with IF...THEN statements. The monitor's routine returns with the value in the A register; the USR OUTVAR routine expects the low half of a sixteen-bit value in Y, with the high half in A; so a simple swap-around before calling $AFC1_{16}$ returns the ASCII value of the key pressed to, for example, the variable P in P=USR(X). *Don't* use P$ without the CHR$ function — it will crash, since ASCII is numbers, not letters! To get a number, use P=USR(X)-48; a simple greater-than/less-than check limits the range of numbers, and allows the user to blunder through the entire keyboard (other than that be-wretched BREAK key!) without any adverse effect. The routine is relocatable without any change, since it refers only to ROM or BASIC-pointer addresses; in decimal, the sequence is 32, 235, 255, 168, 169, 0, 32, 193, 175, 96.

Input: automatic key-scan

Aardvark point out that, for many one-player games, there is no need to scan the keyboard, since the (combined) value of the current control key(s) pressed are stored at 57100_{10}, or rather, by a rather slap-dash bit of wiring, at every fourth location from 57100_{10} to 57220_{10}. There is thus no need to go through the somewhat messy procedure of disabling CTRL-C and the rest of that routine as described in the OSI Character Graphics handbook — although the PEEKs of the chosen location are still needed, of course. The only catch is that, since the SHIFT-LOCK is normally down during BASIC operation, its value of 1 will be added to the total picked up by PEEKing that location — the PEEK is thus likely to return a value one higher than you expect!

PRINT without scroll

The lack of a PRINT AT statement is one of the more irritating parts of OSI's BASIC; this is one of the ways round it, and others are below. The trick here is to convince the PRINT routine that, since the base-line of the screen is never completed, there is never any need to scroll the screen. CHR$(13) gives an equivalent of carriage-return on a print-terminal, returning the cursor to the start of the base-line; using a ; stops the PRINT routine from inserting a line-feed and scroll. Thus a statement of the type:

PRINT CHR$(13)"This prints without scrolling";

will do just what it says. Note that this overwrites the base-line only as far as the string to be PRINTed is long; you may need to insert a few blanks at the end of the string in order to wipe off a longer previous statement. Don't let the overall length exceed 63 characters, though, or else the routine will automatically insert a carriage-return/line-feed — which rather defeats the object of the exercise.

PRINT AT X, Y

Again, the lack of PET-like cursor-addressing is another of the annoying limitations of OSI's BASIC, both for games work and, in my case, for screen editing. Using the X-horizontal and Y-vertical notation, the point D can be expressed in several ways on C1s and C2s. If 0, 0 is top-left, for both C1s (in theory) and C2s

D=53248+X+64*Y

— I say 'in theory' for C1s because of the varying cut-off because of overscan on the video display. Defining 0, 0 anywhere for a C1 is thus a little tricky, and can only be determined for your own machine by trial and error. For C2s 0, 0 can also be placed at bottom left by changing the statement to:

D=55232+X-64*Y

(By the way, Y should be multiplied by 32 on C1s, not 64!) Remember to check that X<64 (32 on C1s) and Y<32, or else the statement will produce some unexpected addresses.

PRINT AT

Again from *Aardvark*, a short statement to print a string D$ on the screen starting at an address D — as defined by the routine above, for example.

FOR Y=1 TO LEN(D$): POKE D+Y, ASC(MID$(D$,Y,1)): NEXT: RETURN

Scores have to be done in a slightly different way, partly because BASIC include the sign (or the absence of one, with a positive number) as the first 'digit' in the string, and partly because the string routine above makes no allowance for increasing or decreasing numbers of digits. There are a number of ways of dealing with this, but most need a definite limit for the number of digits to work well — we've used five as the limit in the examples below. Convert the score to a string with the STR$ function. Then, if you want a simple counter for up to five digits, blank out the leading spaces a FOR:NEXT loop.

D$=STR$(*score*): FOR X=1 TO 5-LEN(D$): POKE D+X, 32: NEXT
FOR Y=X TO X+LEN(D$): POKE D+Y, ASC(MID$(D$,Y,1)): NEXT: RETURN

The position of the *lowest* digit will stay the same with this routine, as opposed to the *highest* with the simple string version. If you want to print leading zeroes, change the POKE in the upper line to POKE D+X, 48 rather than 32.

Suggested alterations to Aardvark games

J. B. W. Harkness writes: As a recent recruit, I've only just received my copy of the Newsletter. I though I'd write with my impressions of two games I bought with my C2-4P.

Fighter Pilot My opinion is much as yours, except that within half an hour or so my seven-year-old son had discovered that firing when the target was on the same line as the centre spot of the sight resulted in a hit. (It could be at either end of the line, it didn't matter.) It was cured by changing line 490:

```
490 IF ABS(IP-AP)<3 OR (IP+64-AP)<3 OR ABS(IP-64-AP)<3 THEN 530
```

Tank For Two This sets up a sort of obstacle course/maze in which two tanks are manoeuvred to shoot at each other. The missiles can be steered in flight and can also be launched from the side of the tank as well as the centre. Twenty hits decides the winner. It's quite an entertaining game except that it also has a fault in it. The initialization routine doesn't put in a left-hand margin, with the result that a tank can disappear off the left hand side, never to be seen again. The cure for this is to change line 220:

```
220 FOR X=1 TO 32: POKE C1+X*L, B: POKE(C1-31)+X*L, B: NEXT
```

I hope these can help others to enjoy these games.

Storage in BASIC: programs, variables and strings

Courtesy of Aardvark Technical Services

(*Editor:* The storage formats described in this article apply generally to most Microsoft BASICs, but the specific addresses and the like given here relate to the BASIC-in-ROM used on the C1/Superboard and C2 series machines, and also, in a slightly modified form, on the UK101.)

Your BASIC programs are stored, line by line, in a partially pre-digested form starting (normally) at memory location 0301_{16}. All BASIC keywords (FOR, GOTO, END, =, CHR$, etc.) are stored as one-byte 'tokens'. Tokens always have the highest bit set (i.e. they are always higher than 128_{10}). Other parts of your BASIC statements (like *AA* and *123* in LET AA=123) are stored as the ASCII characters you typed in. The line number is stored as a two-byte straight binary number (but that does not explain why the highest allowed line number is 63999 instead of 65535!). In addition to these, each stored line of BASIC source contains a two-byte pointer containing the start address of the *next* BASIC line. This lets BASIC search rapidly for a given line number. The format of BASIC statement storage is always like this:

null	*pointer to next line*	*line no.*	*BASIC code – tokens and ASCII*	*null of next line*
___	___ ___	___ ___	___ ___ ___ ___ ___ ___	___

(That information alone is enough to let you write a renumbering program for BASIC programs.)

The 'normally starting at 0301_{16}' pointer can provide interesting possibilities. 'BASIC workspace' — the area in memory where your program and variables are stored — begins at whatever address is contained in locations 0079_{16}, $007A_{16}$. Machine addresses are normally stored low byte/high byte. Thus, when the cold-start routine initializes these locations, it puts *01* in 0079 and *03* in 007A. Now, if you change this, with your trusty ROM monitor or with POKE statements, you can make BASIC store your programs anywhere you choose. In fact, you could have one program stored starting at 0301_{16}, another at 0901_{16}, and another... all using the same line numbers, if you want! BASIC will only find one at a time for running and listing — the one whose beginning is contained in 79, 7A.

Note: the byte immediately before the first line must be the initial null. Normally, the system puts a permanent 0 in location 0300_{16} during the cold-start, and the first byte of the first pointer goes in 0301_{16}. You *must* put the initial null in (at 0900_{16} in the example above) or nothing works.

After you change 79, 7A and put in that initial zero, type NEW, to get BASIC to reset some other pointers for you. Unfortunately, if you put one program one place, reset only 79, 7A and put another program somewhere else, trying to edit the first one will blow up the second program and not work in the first. You can, however, switch back and forth if all you do is RUN and LIST the programs. However, if you also replace 7B, 7C, programs are editable and can RUN happily.

Another note: either avoid programs with lots of variables that can wipe out other programs, or else also update 85_{16}, 86_{16} to indicate that the top of memory is just below the next program up. The hard one to fix is 7B, 7C. It points to variable work-space — so BASIC POKE statements using variables can't fix it: the variables are lost between the first and second POKEs!

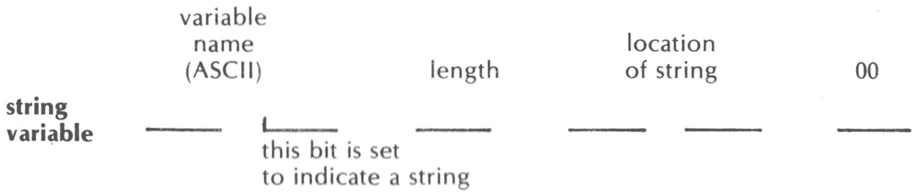
BASIC variable storage

BASIC also needs space to store variables. These are stored in memory above the program — numeric variables, preceded by their names, from the end of memory going up; and string variables from the top of memory going down, their names being kept in a table along with where in memory the strings are actually stored. Two data areas are kept (with name tables) — one for arrays (string and numeric), the other for single variables (string or not) and functions. Since only seven bits are needed for each character of the variable's name, the highest bits are used to show what type of variable is stored. A 1 in the top bit of the second character indicates a string; a 1 in the same bit of the first character indicates a function (in, e.g. DEF FNAB(X)). If neither top bit is set high the variable is numeric, while both top bits high indicates a string function (FNAB$) — although the system does not support the latter.

Single variables are stored immediately following the program, starting at the address pointed at by 7B, 7C on page-zero. (The abbreviation (7B, 7C) is used to indicate the contents of 7B, 7C. Thus, the single variables start at (7B, 7C).) Each variable is stored in a fixed-length six-byte block in this area:

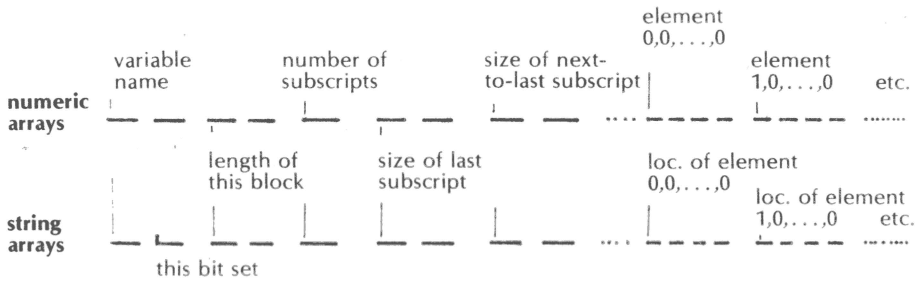
	function name (ASCII)		loc. of first char. after = in DEF statement		location of dummy variable	
function	___	___	___	___	___	___
	this bit set if function					

	variable name (ASCII)		floating point value			
numeric variable	___	___	___	___	___	___



To find a variable, BASIC searches the names, starting at (7B, 7C), skipping to the next name six bytes later each time until a match is found. If a string is being searched for, the actual string is not here, but stored starting at the address contained in the fourth and fifth bytes of the entry in the table. The search ends if a match is not found by the end of the area, (7D, 7E).

Arrays are stored in assorted length blocks from (7D, 7E) to (7F, 80) as follows:



To find an array element, BASIC starts at (7D, 7E) and looks at the name, then skips to the name in the next block (that's why we have that third byte!), and the next block, and the next. . . until a match is found, then skips four bytes per element until it finds the *element* it wants. If it's a string, we have the length and location stored here, not the actual string, as before. This table is finished by (7F, 80).

Strings are actually stored starting at the *top* of memory, this being indicated by (85, 86_{16}). Modifying the contents of 85_{16} and 86_{16} (or having answered a number less than the actual memory size to the MEMORY SIZE? request at cold-start) will keep the strings from wiping out any other programs or data you may want to tuck safely away at the top of RAM. BASIC uses this space at the top of the memory with no regard for saving space or re-using space until it runs out of free space. It keeps a pointer to the next free space (working from top to bottom) in (81, 82), putting any strings it needs there, whether array or not, and updating the pointer until it runs out of room — in other words, when (81, 82) equals (7F, 80). To keep from wiping out the array tables — the first thing it would run into — BASIC calls a 'garbage collection' routine that tries to shuffle the strings around back up to the top of memory and thus reclaim unused space. Unfortunately, there seems to be a bug in the garbage collection routine that makes it hang up if it has to try to relocate string arrays. Unless you try to do some fancy string array manipulations in big loops, you probably won't run into trouble — it seems to affect arrays of more than around twenty elements. In case you want to go bug-hunting, the FRE(X) routine at $AFAD_{16}$ calls the garbage collector before finding out how much room is left between (81, 82) and (7F, 80).

Representation of numeric variables

The floating point value of a numeric variable is stored in its four bytes in normalized binary exponential (scientific) notation:

exponent sign	sign and most significant bit		least significant bit
10000011	.00100000	00000000	00000000
exponent	binary point		

This would be read as: $.101_2 \times 2_{10}{}^3 = 5_{10}$

The last three bytes contain the number, to 24 bits' accuracy; the first byte is the power of 2 — if you like, the number of places to move the *binary* point. The binary point is like the decimal point, except to its right we have the $\frac{1}{2}$'s column, $\frac{1}{4}$'s column, $\frac{1}{8}$'s column, etc., instead of $\frac{1}{10}$'s, $\frac{1}{100}$'s, etc.

The most-significant-bit of the value (bit 7 — the topmost bit — of the second byte) is always interpreted as having the value 1. If it were 0, we could shift the number to the left — binary point to the right — until it *was* 1 increasing the exponent by as many places as we moved. Since this is understood to be so by the system, we can use that actual bit in memory as the sign bit: a 1 in that bit is negative. Negative numbers are not represented in two's-complement form; the exponent, however, is. Some examples:

5	10000011	00100000	00000000	00000000
1	10000001	00000000	00000000	00000000
2	10000010	00000000	00000000	00000000
3	10000010	01000000	00000000	00000000
4	10000011	00000000	00000000	00000000
7	10000011	01100000	00000000	00000000
15	10000100	01110000	00000000	00000000
-5	10000011	10100000	00000000	00000000
.375 ($\frac{3}{8}$)	01111111	01000000	00000000	00000000
0	00000000	00000000	00000000	00000000

If you want to explore this further, there follows a short BASIC program to read the binary representation of a number from memory. It looks at the second, third and fourth bytes after (7B, 7C). Killing line 30 lets you look at the variable name (and the first two bytes of the value).

```
10  INPUT M
20  P=PEEK(123)+256*PEEK(124)
30  P=P+2
40  FOR J=0 TO 3
50  N=PEEK(P+J)
60  GOSUB 200
70  PRINT "    ";
80  NEXT
90  PRINT
100 GOTO 10
200 FOR A=0 TO 7
```

```
210  B=N AND 2^(7-A)
220  IF B THEN PRINT "1";: GOTO 240
230  PRINT "0";
240  NEXT
250  RETURN
```

(Yes, lines 210 and 220 *are* correct.)

The program waits for you to input a number, then prints the binary representation of it, and then loops round to wait for another number.

Larger display for C1/Superboard series

Almost the only poor part of the design of the C1/Superboard series is its meagre video display — 32×32 if you're lucky, more likely 25×25 because of overscan on any TV used for the display. A modification we've recently heard of alters the video circuitry to produce 'guard bands' to get round overscan on the standard display — it's to be marketed over here, and we'll publish details when we have them. But the mod shown here tackles the problem in a different way, simply doubling the screen memory and using a software 'patch' to inform the video circuitry and BASIC print routine that the extra memory is there. This mod is a much-tidied-up variant of one that was published some time ago in the States — we haven't found out who by, though. Anyway, on with the article!

If only OSI had not skimped quite so much on the display driver section of the Superboard the machine would be amazing value instead of merely exceptional. This article describes a method of extending the format to 64 characters by 30 lines, which — allowing for overscan — will give a usable 50 character by 30 line display.

Although the system monitor is accessed by BASIC to determine the screen size continually, giving a peculiar output when LISTing programs, a software 'patch' can be implemented to allow use of the full screen area under monitor control.

The software patch need not be used, however, when the screen is accessed via POKE (or in machine code routines) as this function does not access the monitor's screen-size look-up table.

The modification requires few components (under £20 in total) but will require a fair degree of competence in soldering, and at least three hours' work. For those people who have some doubts about their capabilities all that can be said is that the modifications are reversible, and an unskilled person has carried them through from these instructions with complete success.

Components required are: one 8MHz crystal; two 2114L3 (possibly more, if 2MHz operation is required and any existing devices are not up to it!); one 74LS139; one 74LS161/3; two 16-pin IC sockets for the 74LS chips; and the usual assortment of wire, solder and the like.

Exchange the 4MHz crystal on the Superboard for the new 8MHz device, and check that everything still operates. This has doubled the master clock of the machine, so that everything will run at twice the original speed — including the processor *and* the cassette interface. Run a memory test on both the user RAM and video RAM, checking every bit in these areas — a slow process but it saves later difficulties and probably the cost of two 2114L3s. [We gave a simple BASIC memory test in the last issue, in the section on doubling the operating speed of a C2 — *Ed.*] On early machines slower RAM was used in the screen memory, and sometimes in the main user area. If any failures are found, take the following action:

a) If the machine is required to operate at 2MHz weed out the failing devices and sell them to a less demanding acquaintance! By enough devices to restore your user workspace plus four for the new video RAM.

b) If 2MHz operation is *not* required, try to find at least four fast devices for the new video RAM and replace the user workspace with four new 2114s (550ns minimum speed). Restore the processor clock to 1MHz operation by cutting the track to U8 pin 37 (Ø0 in) (component side) next to U8; then connect U8 pin 37 to U30 pin 12.

Having done either *a)* or *b)*, test the other conversion parts required by substitution in the machine, and start modifying. Note that the following abbreviations are used throughout, to save boring repetition:

(ts) — track side of board, i.e. the underside of the board.

(cs) — component side of board.

V_{cc} — any 5V point on the board.

Gnd — any 0V point on the board.

(U99 pin 21) means 'the track that used to connect to U99 pin 21 before that track was cut'.

PTH — plated through hole.

Conversion

The video memory is normally arranged as a 1K block occupying locations from $D000_{16}$ to $D3FF_{16}$ and is accessed by the video display circuitry via an address multiplexor. When the CPU writes new data into the video RAM, control of the video address lines is given to the system address lines by the multiplexor.

In order to allow the increased video RAM to be accessed by the system address bus the multiplexor control signals must be modified.

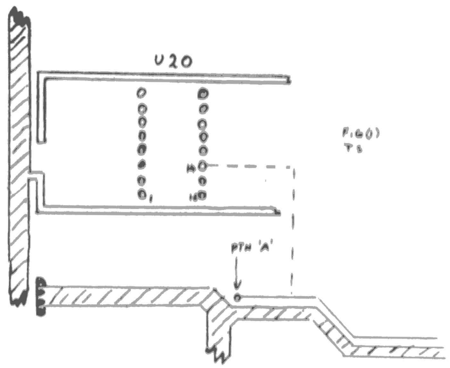


Fig. 1

Drill out PTH A and connect the (ts) track to U20 pin 14.

Isolate U20 pin 11 (1 cut).

Isolate U20 pin 1 and connect pin 1 to V_{cc} (1 cut).

Connect (U20 pin 11) to U20 pin 10.

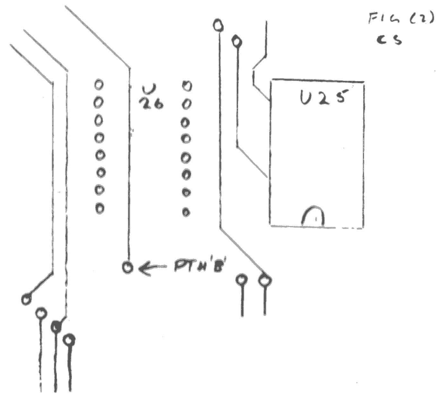


Fig. 2
Drill out PTH B.
Connect U56 pin 1 to U56 pin 2.

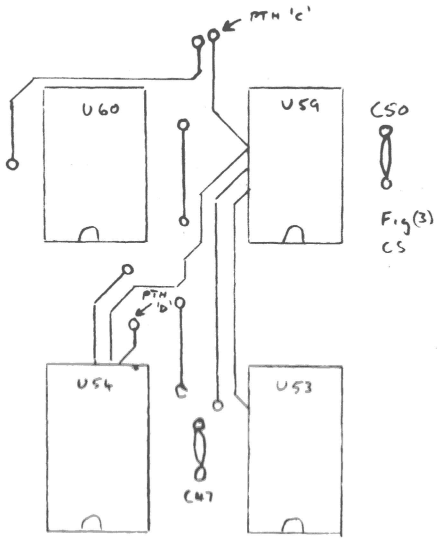
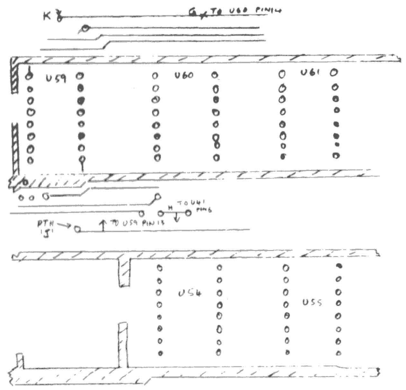
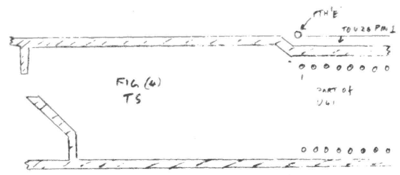
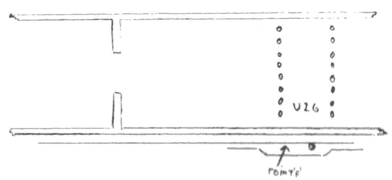


Fig. 3
Drill out PTH C and PTH D.

Fig. 4
Solder two 16 pin IC sockets into the prototyping areas at U26 and U27, observing the same orientation, and connect Vcc and Gnd to pins 16 and 8 of these sockets respectively.







On U26 common pins 3, 8, 14 and 15.
On U27 common pins 3, 4, 5, 6 and 8.
Drill out PTH E and connect the (ts) track to U26 pin 1.
Isolate U55 pin 10 and connect it to U26 pin 12.
Connect U55 pin 11 to U26 pin 5.
Connect U55 pin 13 to U26 pin 10.
Connect U55 pin 14 to U26 pin 4.
Connect U26 pin 2 to point F.
Connect U26 pin 13 to U27 pin 14.

On U27 common pins 16, 1 and 9.
Connect U27 pin 10 to U61 pin 15.
Connect U27 pin 2 to U30 pin 2.
Isolate U65 pin 1 and connect it to U26 pin 13. This track is accessible where it goes through a PTH just above U59 (point K).
Connect U27 pin 7 to U30 pin 15.
Connect U60 pin 14 to U54 pin 6 and to point G.
Connect U41 pin 6 to point H.
Drill out PTH J and connect the (ts) track to U59 pin 13.

Isolate U41 pins 6, 7, 8.
Connect (U41 pin 6) to U41 pin 7.
Connect (U41 pin 7) to U41 pin 8.

(This is a little unaesthetic but it saves a lot of soldering!)
Make up two composites each of two 2114 packages, soldering all pins together except the respective pins 8.
Solder a 10 cm length of wire-wrap wire to the uppermost pin 8 of each composite, and connect these to U55 pin 12.
Insert the composites into U39 and U40 sockets, leaving only the uppermost pin 8 of each composite unconnected to their respective socket-holes.
Insert a 74LS139 into U26 socket.
Insert a 74LS161/3 into U27 socket.

This completes the hardware modifications. All the usual admonishments to observe device orientation, using a heat sink, wearing aluminium foil underwear and chaining yourself to a water pipe whilst working have been left out, because if you haven't evolved a neat and efficient way of working you shouldn't be doing this! It is worth saying, however, that you should recheck your work, as even the best of us make mistakes — hence the jam-jar-full of expensive but useless devices in my workshop...

Connect up to a TV and — assuming the TV has warmed up — flick the on/off switch of the Superboard quickly. You should see a normal screen full of random characters — though rather more than before, of course. If you *don't* get that picture, you haven't been careful enough.

Check the video and user RAM again for speed failures and run a short screen test program to make certain that you can obtain the maximum display format. When you have gained enough confidence to leave everything on, the screen should be displaying about 50 characters per line and 30 or so lines.

If 'BREAK' is pressed the *top* half of the screen will clear and the normal D/C/W/M ? prompt will appear up and to the right of centre. Although somewhat confusing this is perfectly correct, as the original 1K video RAM now controls the top of the screen, while the additional section controls the lower half. When the normal 'return', 'return' sequence is entered as a response to the cold-start prompts the screen, under the monitor's previously normal 32-column-per-line step, will show *two* columns of alternate output:

```
                              D/C/W/M ?
MEM SIZE ?                    TERMINAL WIDTH ?
OK                            ?
```

In order to utilise the full screen width — and not go mad trying to understand just what your severely confused Superboard is saying to you — some way of modifying BASIC's handling of LIST and PRINT commands is required.

The routine that handles this in BASIC is stored at $BF2D_{16}$, which uses three locations in the monitor ROM as a look-up table to find out which type of machine it's in, and thus to determine where to put characters during output. The three locations used for this are:

FFE0	65	cursor rest position (cursor to start new line at $D365_{16}$)
FFE1	17	characters/line -1 (i.e. 24_{10} characters/line under normal routine)
FFE2	00	1K video RAM available (<>0: 2K video RAM available)

The neatest way round this is to burn a new monitor PROM copying the old ROM except for these locations, changing them to the C2's values of 40_{16}, 48_{16} and 01 respectively. This would not, however, solve the overscan problem — some characters would be lost in the left-hand margin. The other but dirtier way of sorting this is to use a software patch. The patch that follows is essentially a copy of the BASIC routine at $BF2D_{16}$, but as a free bonus gives a fast screen clear when executing ?CHR$(1) — which the BASIC routine should have been able to do had it not had to spend so much time trying to sort out whether it was driving a C1 or C2!

The patch is best placed in the 'safe' area of memory below the user RAM — from 0222_{16} upwards. The patch is almost 200 bytes long — much too long for hand entry every time via the monitor — so some form of machine load is essential. A BASIC routine using POKEs would do the job; likewise a machine-code tape, which would also be faster-loading than BASIC.

The patch also has the advantage that, being in RAM, the cursor's starting position, the line length and the like may all be user-specified, so as to allow for varying degrees of overscan on your system. The Superboard screen handling routine looks at locations $021A_{16}$ and $021B_{16}$ to pick up its output vector to $FF69_{16}$ for the actual screen handling; this has to be changed to the start of the patch, at 0222_{16}, in order for the patch to take over from the built-in routine. *This vector is reset to point at $FF69_{16}$ after 'Break', and so must be reset to point to 0222_{16} after any pressing of 'Break'.* An easy way of doing this is via POKEs in direct mode: POKE 538, 34: POKE 539, 02. But note that in direct mode these *must* be input in the same line — otherwise BASIC will find itself looking for a screen routine at $FF22_{16}$! If you load the routine by means of a machine-code tape, placing .BD11G at the end of the tape will send the system straight into BASIC's cold-start routine, obviating the need to reset and reload the patch start address in 021A, B.

The combination of the hardware modifications and this software patch will give a video display of about 50 columns by 30 lines — 12 to 14 columns are lost in overscan, and the top two lines are also lost off the top edge of the screen.

0222	8D 02 02	STA 0202	*Copy of BF2D routine*
0225	48	PHA	
0226	8A	TXA	
0227	48	PHA	
0228	98	TYA	
0229	48	PHA	*All registers saved on stack*
022A	AD 02 02	LDA 0202	*Retrieve A from parking space*
022D	F0 4C	BEQ 4C	*Return if null*
022F	AC 06 02	LDY 0206	*Start of delay routine:*
0232	F0 08	BEQ 08	*picks up delay loop counter from 518_{10};*
0234	A2 40	LDX 40	*larger value gives longer delay*
0236	CA	DEX	
0237	D0 FD	BNE FD	
0239	88	DEY	
023A	D0 F8	BNE F8	*End of delay*
023C	C9 0A	CMP 0A	*Linefeed?*
023E	F0 46	BEQ 46	*if yes, jump to 0286 for line-feed*

0240	C9 01	CMP 01	*Screen clear?*
0242	D0 1A	BNE 1A	*if no, jump over screen clear routine*
0244	A9 20	LDA 20	*Screen clear routine*
0246	A0 08	LDY 08	
0248	A2 00	LDX 00	
024A	9D 00 D0	STA D000, X	
024D	E8	INX	
024E	D0 FA	BNE FA	*Load Dn00+X, 20_{16} until one 'page' is done*
0250	EE 4C 02	INC 024C	*Dn00=D(n+1)00 – restart one 'page' down*
0253	88	DEY	
0254	D0 F4	BNE F4	*Do until all eight 'pages' of screen are done*
0256	A9 D0	LDA D0	
0258	8D 4C 02	STA 024C	*Restore routine to start value – D000*
025B	4C 7B 02	JMP 027B	*and exit*
025E	C9 0D	CMP 0D	*Carriage return?*
0260	D0 06	BNE 06	*if no, jump over carriage-return routine*
0262	20 D2 02	JSR 02D2	*Else do carriage-return*
0265	4C 7B 02	JMP 027B	*and exit*
0268	8D 01 02	STA 0201	*If A not above, save it at 0201*
026B	20 C8 02	JSR 02C8	*and print it*
026E	EE 00 02	INC 0200	*Increment cursor index*
0271	A9 F9	LDA F9	*Maximum for cursor index (see Note 1)*
0273	CD 00 02	CMP 0200	
0276	30 0B	BMI 0B	*Do CR/LF if greater than maximum*
0278	20 DA 02	JSR 02DA	*Else print cursor*
027B	68	PLA	*retrieve registers*
027C	A8	TAY	
027D	68	PLA	
027E	AA	TAX	
027F	68	PLA	
0280	4C 6C FF	JMP FF6C	*and exit back to BASIC*
0283	20 D5 02	JSR 02D5	*Save next char. position – for backspace?*
0286	20 C8 02	JSR 02C8	
0289	A9 BF	LDA BF	*Nominal cursor start – see Note 2*
028B	EA	NOP	
028C	EA	NOP	
028D	8D 02 02	STA 0202	
0290	A2 07	LDX 07	*Pick up scroll routine from $BFF3_{16}$*
0292	BD F3 BF	LDA BFF3, X	
0295	9D 07 02	STA 0207, X	*and store at 0207_{16} to $020E_{16}$*
0298	CA	DEX	
0299	10 F7	BPL F7	
029B	A2 D7	LDX D7	*X defines the bottom of the main scroll;*
029D	A9 40	LDA 40	*40_{16} here tells the routine to transfer the*
029F	8D 08 02	STA 0208	*char. 40_{16} 'down' to the current Dnnn*
02A2	A0 00	LDY 00	*and do scroll*
02A4	20 07 02	JSR 0207	
02A7	D0 FB	BNE FB	
02A9	EE 09 02	INC 0209	*move down after first four lines done*

02AC	EE 0C 02	INC 020C	
02AF	EC 09 02	CPX 0209	*Down to D7nn yet?*
02B2	D0 F0	BNE F0	*If not, go back for another 4-line 'page'*
02B4	20 07 02	JSR 0207	
02B7	CC 02 02	CPY 0202	*Do until nominal cursor start – see Note 2*
02BA	D0 F8	BNE F8	
02BC	A9 20	LDA 20	*Clear text entry line*
02BE	20 0A 02	JSR 020A	
02C1	CE 08 02	DEC 0208	
02C4	D0 F8	BNE F8	
02C6	F0 AE	BEQ AE	*and jump back to exit-to-BASIC*
02C8	AE 00 02	LDX 0200	*0200 stores current cursor position*
02CB	AD 01 02	LDA 0201	*0201 stores character to be printed*
02CE	9D 00 D7	STA D700, X	*Place A on screen*
02D1	60	RTS	
02D2	20 C8 02	JSR 02C8	*Place A on screen – see Note 3*
02D5	A9 C8	LDA C8	*Carriage return – C8 is cursor start – Note 4*
02D7	8D 00 02	STA 0200	
02DA	AE 00 02	LDX 0200	*Enter here to save next character on*
02DD	BD 00 D7	LDA D700, X	*Could be used for backspace?*
02E0	8D 01 02	STA 0201	
02E3	A9 5F	LDA 5F	*5F is cursor – jump back to print cursor*
02E5	D0 E7	BNE E7	*at next character location on.*

Note 1 F9 here is maximum permitted displacement of cursor before a carriage-return is forced. The greatest possible displacement is FF: this will vary according to the amount of overscan on your system.

Note 2: BF here is the end of the line *above* the cursor line — it defines the end of the 'save and transfer' part of the scroll routine. BASIC's scroll routine is a nice example of storing a constantly-changed routine in ROM, to be collected each time a scroll is needed — have a look at the ROM listing, then see what this part of the print routine does with it.

Note 3: There are four possible entry points here! 02D2 replaces the cursor with the previous blank space before doing the carriage-return; 02D5 sets the cursor start position (see Note 4); 02D7 could be used for PRINT AT anywhere in the bottom four lines of the screen, by loading A with a new displacement. 02DA saves the current contents of D700+X which, since X has usually just been incremented, is normally 20_{16}, a blank; but by changing X, via (0200), this could be used to forward-space or backspace the cursor.

Note 4: C8 here is the cursor start position, allowing for eight characters' overscan. The maximum possible, without overscan, is C0; change this to suit your system.

[*Editor:* We have checked and rechecked this article as carefully as possible to remove any errors; but obviously we cannot be held responsible for any damage to your machine arising from errors that have managed to get through in this article. The important point, obviously, is to work with care, on both the hardware and software sides of this mod. If done properly, it converts a Superboard/C1 series machine from an interesting but limited gadget into a superb tool — so it's worth doing well!]

Technical literature

We have at last located some technical literature on the smaller OSI systems, published in the States by the Howard Sams group early this year, but with OSI's name as 'publisher' on the cover. There are two separate books, the *C1 Technical Guide* and *C4 Technical Guide* — the latter for the new C4 machine which has not as yet trickled its way across the water, but which shares most of its boards with the C2 used by many of our members. The books contain complete board schematics for pretty well everything, including mini-floppy drivers, and sets of trouble-shooting guides that include full 'scope patterns. In many ways these should have been included with the kit in the first place... but no doubt OSI would argue that these are for electronics buffs, not for the kind of 'home computer user' that they're aiming for in the States. (If you want to see what kind of animal that is, see the advertising blurb for the new C4 — it drives a full 'home security system' among many others...) The jacket prices are $7.95 for the *C1 Guide*, $15.95 for the *C4 Guide*; we don't yet know of any dealer with them in stock, but we'll tell you as soon as we do.

Dealer Notes

There seems little point in repeating the whole of the dealer list from last issue; we'll include other dealers and their specialities in other issues as and when that information comes in. We've had quite a lot of help from several OSI dealers, as can be seen throughout the pages of this issue — thanks to you all! We also have a letter from Alan Caves of *Cavern Electronics*:

"Thank you for including us in your *Dealer Notes* (but note spelling of Wolverton!). As you mentioned we are only selling C1s at present although any of the range can be obtained to order. Some of the standard software should also be available shortly — when we can get it! I am also engaged in writing some useful BASIC routines, general purpose in nature, such as a graph plotter. These will be available in a month or so [from February '80 — *Ed.*]. I am also willing to market on a royalty basis good programs from your readers.

Finally, as you know, the main fault with the C1 is the poor display. Has anyone come up with a mod. for increasing the number of characters per line? Also details of how to add parallel port facilities would be useful. All of these could be sold on a royalty basis.

If I can be of any help to you or your members please let me know."

Cavern Electronics are at 94 Stratford Road, Wolverton, Milton Keynes MK12 5LU.

One piece of information that would help all of us — OSI users and dealers — is accurate statistics on the reliability of OSI equipment. It would be even more useful to have these as comparative statistics against the reliability of other system ranges — dealers selling equipment by a number of manufacturers please note! The reason we ask is that in a recent issue of *Personal Computer World* one of the reasons given by members of the *Byte Shop* chain for their financial difficulties was very poor service back-up by OSI. It seems likely that if dealers are not able to provide accurate information to counter the obvious rumours that will start from that comment — and the once all-too-accurate image, courtesy of ADHOC, that OSI kit was over-priced — OSI computers are going to remain very much in the second or third league as far as sales are concerned. Bad documentation doesn't help either... So dealers, it's over to you — we'll help all we can, but you have to supply us with the information before we can publish it!

User Group notes

User Group membership

Despite the fact that as yet (February '80) we've had no mention in the small-computing press, our membership is now quite a respectable size, and growing at the rate of one or two new members a day. Part of this is due to help from Lotus Sound and Mutek, both of whom included our application forms with their regular mail-shots to former clients — thanks! After our initial worries about the financial risk we were taking — the production of the Newsletter is going to cost at least £1,000 for this year — the Group *does* look as though it is going to be viable in a financial sense. We are well on the way to that 'minimum membership for survival' of two hundred, and it's also more than likely that there *will* be enough money left over to finance a number of 'extras': more about those in a moment. For those interested, our membership at present is almost evenly divided between users of C1/Superboard and C2 systems, with a handful of C3s and also some UK101s. Many thanks to those of you who did fill in the section on the form about applications for your systems: this will be useful to us later.

Video planning charts

Steve Bridges wrote in to say that he was thinking of producing pads of video layout sheets in the various formats for C1s and C2s, as shown in the back of their manuals' Graphics Handbook. He reckons the cost should be around £1.50 for a 50-sheet pad, but the price that he is able to get them printed for will depend on the print run, which will depend on the number of people interested — so contact him at 11 Shaws Road, Southport, Merseyside PR8 4LR.

In the same vein, one of the ideas we're working on is a range of sets of write-on/wipe-off planning charts for the development stage of programming. We'd be including things like charts for variables, for assembler labels and addresses, memory allocations (memory-map and 256-byte page), 6502 and Z-80 opcode lists, hex/double-hex/binary/decimal conversion chart-calculator and the like, as well as video charts in proper screen ratio and in 32×64, 32×32, 25×25 and (for UK101 or, for that matter, Nascom users) 16×48 formats. Although they will have to be glossy for the wipe-off to work, they can be photocopied by placing a sheet of matt tracing paper over the top. They'll be punched on *both* edges (so you can place them face-to-face when working), in both two-hole British format and three-hole American to fit in your OSI manual. Anyone who's used a planning chart of this kind in business will know just how useful these would be in programming. We reckoned on a retail price for a set of ten or so cards (including pen!) of around £5.00, so members should be able to have them for around £3.50 or so. We should have them ready in May or June, but don't send any money until they *are* ready! We would like to know if you're interested, though, so we can gauge the overall print run.

Documentation

A slightly longer-term plan is to get together enough information for an equivalent of *The PET Revealed* for OSI's C1 and C2 series. If OSI can't do a decent job of documenting their products, we'd better do it for them! We're aiming to produce a *complete* manual for the C1/C2 series that actually *does* explain how the machines work and what they can (and can't) do, and which *does* explain how to use them to their fullest extent. Their fullest extent, as we are discovering, is a very long way indeed. The great advantage of these small machines (the C2 especially) is the simplicity and cheapness of interfacing them to the outside world — for comparison, look at the price of a *single* D/A converter or decent parallel interface for the PET! But without adequate documentation these superb facilities can be merely frustrating, worse than useless. So let us know what *you* are doing, what *you* have found out. We want to get this in book form and into the bookshops and computer stores in time for the User Group's first birthday in December '80, so get moving! And there will, of course, be a special reduced price for members of the Group!

Hard copy service

Following enquiries by a number of members, we can now offer a hard-copy service for material on Challenger-format tapes, either as listings or as output from a run. The only practical way of costing is on a time basis, with a minimum charge of £2.50, to cover our handling and postal costs. You'll get a printout of around 16K's-worth of BASIC listing for that — rather less for assembler or machine-code — so for best value put two or three programs on any tapes you send us. For obvious reasons, *don't* send us your master tape or your only copy of a program! The tape will, of course, be returned with the printout. Turnround should be less than a week door-to-door. Contact Tom Graves for more details, at 19a West End, Street, Somerset BA16 0LQ; telephone Street (0458) 45359.

Printers

Along the same lines, here's our first hardware offer to members, worth about thirty times your annual subscription! One OSI dealer — who shall, for the moment, remain nameless for obvious commercial reasons — has offered to import for us a number of Base-2 printers at a price way below the going rate. The Base-2 is a fairly typical medium-speed matrix printer which is just coming onto the UK market at around £450–£500. It's typical in that it's fairly small, reasonably quiet, and uses plain paper. It's not unusual in having both friction and tractor feed as standard, the tractors being adjustable to take up to 8½" paper. But it is unusual in having four interfaces built-in as standard, with two complete character sets in ROM, options on two more, and room for a further user-programmed set in RAM as standard. Most of its format functions are software-controlled as well. The version we're after will have that all-important £ sign in the character set, and should have the entire Challenger graphics in the ROM as well. All this for £325! — plus the dreaded 15%... The dealer says he would prefer an order for at least five printers between us to make it worth his while: so if you've been thinking about buying a printer, contact Tom Graves as soon as possible.