# the AARDVARK JOURNAL

APRIL 1982 VOL. 3 NO. 1

*** INDEX ***

## THE BEGINNERS CORNER
## PEEKS & POKES

PEEKS and POKES look a lot more complex and a lot more difficult to use than they really are. Part of the problem is that the normal explanation in the manual is relatively (turse) and uninformative. Most BASIC manuals say something in the order of:
PEEK (I) Returns the decimal value of the specified memory or I/O location. (decimal) Example: X=PEEK (741) checks to see if LIST is enabled (76 indicates that is is enabled).
POKE I,J Loads memory location I (decimal) with J (decimal). I must be between 0 and 65536 and J must be between 0 and 255. Example: 10 POKE64256,255 loads FB00 with FF (hex).

That certainly doesn't sound very impressive, does it? However these are two simple concepts that have an awful lot of handy uses. All you have to remember in the way of definition is that when you PEEK a location in memory, you get back whatever number is actually stored in that memory location and that when you POKE a memory location you force the machine to place a certain number in a given location. What do you mean what's it good for? There are about 3 reasons why PEEKing and POKEing is extremely handy for computerists.

The first thing that makes PEEKs and POKEs handy is that the people who wrote the BASIC language had to put a lot of things in the RAM. For instance the number of nulls that the computer puts on the end of a line of print varies fROM 0 to 255. Since that number has to vary, they couldn't very well put the value in a ROM someplace. Therefore, for this and a lot of other values, the BASIC language is told to look in a memory location where it will find the value to use. In this case, the number of nulls that you put on the end of a line is stored in location 13 in basic in ROM machines and in location 21 in 65D.

Now the reason you care about that is that the people who wrote the BASIC never figured that you might need more than 8 nulls at the end of line. They always assumed that we would use expensive printers or very good printer interfaces. Therefore, if you tried to put in a number like 25 for the number of nulls in BASIC, (NULL 25) you would get back an FC error. Unfortunately, unforseen by the writers in BASIC, some of us use selectrics and some of us use pieces of antique equipment. We can often need as many as 80 nulls at the end of a printed line in order to get that slow carriage back. Also, when we use high speed cassette interfaces (such as 1200 and 600 baud), we often have to add a lot more nulls on the end of BASIC line to give the machine time to process the BASIC as it goes back into the machine. It comes down to the fact that we can circumvent the original intention of the writers of BASIC by directly poking a higher number into location 13. By bypassing the null command and executing "POKE13,80" we can make this system put 80 nulls at the end of every printed line. In this and a lot of other ways we can expand beyond what the programmers envisioned when they set this system up.

Cont'd Pg. 15

RICHARD TRETHEWAY, MINN., MN

Dear Rodger:

With regard to my letter that appeared in the last Aardvark Journal. Let me publicly apologize to Mr. Steven Gale. I did not mean to hold him up to ridicule. When I wrote the letter, I wrote it to you and I did not intend it for direct publication, but you had no way of knowing that since I didn't say so. It was my hope that you would publish a correction or addendum and not my letter, or I certainly wouldn't have sent in the hardcopy stating "THE ARTICLE IS BALDERDASH". I again apologize to Mr. Gale and to your readers. I hope you will print this in the next issue.

Sincerely,

STEVEN GALE, SHAKER HTS, OHIO

Dear Editor,

I almost went into shock after reading Mr. Trethewey's letter regarding my article on 65D V3.3. There has been a misunderstanding that needs to be repaired. The intention of the article was not to take potshots at OSI. I am very optomistic about the DOS and I thought that showed in the article. A good review has a balance of good and bad features. My article was about three-fourths on the positive side and one fourth on the negative.

I stand behind ALL of the criticisms made in my review. When indirect files are not used, <CTRL> 'X' does crash the system. Because 'X' is next to 'C', I have crashed many times by accident. A better letter would have been 'I' (for indirect). Mr. Tretheway must not have read his manuals. Location 9976 is listed as disable ":" terminator. If you add this poke to his sample program, it crashes when an open command is given. Loading and unloading the R/W head is not good for the drive. Poke 9976 and shorten the life of your disk drive!

Insulting my capability as an author and reader was not the right way to have more information about the DOS available to the public. If you read my article carefully, you will see phrases like "minor problems", "decoder is great", "utilities...are great", "manuals are great", "much improved", etc. I was not writing a 10 page article, so I couldn't have a whole paragraph on each utility and reference manual. I'm glad that Mr. Tretheway made those features clear. Aren't we both on the same side?

Sincerely,

## DISK BUFFERS, WHAT ARE THEY?
by Don VanSyckel

The prupose of a disk buffer simply stated is to buffer data which is going to or coming from a disk. Alright, so what does this mean? In OSI disk systems, the diskette subsystems are dumb subsystems or peripherals. This is evidenced by the fact that the main processor actually operates the drives via an Asynchronous Communication Interface Adapter (ACIA/6850) and Peripheral Interface Adapter (PIA/6821). With this in mind, some background is required in order to understand why things are structured in the manner they are. Any diskette has data written to it or read from it in a continuous stream. As such, the diskette drive can not directly locate a particular byte or record on a floppy diskette. On many systems, and OSI in particular, the diskettes are soft sectored. This basically means that the diskette drive can only find one given point on any track with its hardware. This point is located by using the index hole whose status comes to the CPU as a **discrete** input via the 6821.

Let's examine reading from the diskette first, when the CPU is queued by the index hole, it initializes the 6850 to look for the data stream. When the 6850 has a character, the CPU reads it and the 6850 looks for and receives the next character and so on. Now, the CPU is receiving data from the diskette. The data stored on the track is not all user data but also includes formatting data. I do not intend to explain formatting here for it is user transparent. When the user data comes to the CPU, the CPU can do one of two things with it; either look for and save the specific data which is required or save all the data and let another routine examine it and extract the given data that is required. The later is much less complicated and has the advantage that all the data on a given diskette track may be examined at will, with only one diskette access. As opposed to the first method where separate diskette accesses would be required for each piece of information required even if it was on the same track as the last accessed data. So where should the CPU save the data? In the disk buffer, of course.

Writing to the diskette is just the reverse. The CPU waits to be queued by the index hole and then it pauses to allow a gap. The CPU then writes all formatting and user data to the diskette. The information for a particular track sector is all written at one time so all of the user data must be available when this writing operation begins. Where should the user's program store this information? In the disk buffer.

The disk buffer is a block of memory, usually RAM, where the CPU writes data to or reads data from during a diskette read or write, respectively. The disk buffer can, in the general case, be located anywhere in memory and be any size. In a specific case, the disk buffer may not reside where there is something else in memory already, such as your program, Disk Operating System (DOS), stack, variables, etc. For a diskette read, the disk buffer must be RAM; however, for a diskette write, the disk buffer could be ROM and/or RAM. OSI has also restrained the simple disk buffer to be a multiple of 256 bytes, one page; up to a total of 13 pages on one track if one sector is used or twelve pages total if two or more sectors are used on a track. When more than one sector is used the diskette drive handler routine leaves a gap after each of the sectors so that the beginning of the subsequent sectors can be located for reading and rewriting.

The DOS is the program which actually handles the reading from and writing to the diskette. The CALL command is used to read and the SAVE command is used to write. In the general form these commands are:

```
CA MMMM=TT,S
SA TT,S=MMMM/P
```

where:

MMMM=4 digit hexidecimal number which is the address of the first location of the disk buffer, leading zeroes must be included (<$1000)
TT = 2 digit decimal number of the diskette track, leading zeroes must be included (01-09)
S = 1 digit hexidecimal number of the sector on the specified track
P = 1 digit hexidecimal number indicating the number of pages to write to the diskette.

Let's examine each of these two commands in detail. CALL command parameters are passed to the appropriate DOS routines. The DOS commands the diskette drive to select the specified track, the track is read to locate the beginning of the sector, and the data is read from the diskette and stored sequentially in the disk buffer. In this case, the buffer must be RAM, but it could be special function RAM such as the video display or color control on the OSI 540 board. It should be noted that the user can not specify how much data is to be read. The DOS will read the entire sector specified in the CALL command. For example, if the command CA D000=40,1 were executed to fill the video memory with a picture and sector #1 of track #40 were 13 pages long, the DOS would write data out to $D000 thru $DCFF which would destroy anything in its path from $D800 thru $DCFF. If sector #1 of track #40 were 8 pages long, then it would exactly fill all of video memory.

The SAVE command parameters are passed to the appropriate DOS routines. The DOS, as in the CALL command, selects the specified track, the track is read to locate the beginning of the sector specified, and the data is stored sequentially onto the disk. The disk buffer which contains the data is usually RAM but could be an EPROM, the system monitor PROM, or video memory. For example, if you use some device other than the video monitor (device #2) as the system console you could create a picture on the monitor and store it on diskette for later recall. The command SA 45,1=D000/8 would save the 8 pages of video memory from $D000 thru $D7FF. If you changed the picture slightly and saved it also and repeated this procedure several times, a small BASIC program could be used to CALL them to the screen creating an animated effect.

Not that general disk buffers have been discussed, let's examine the specific disk buffers which the DOS and BASIC use. These are as follows:

| USED BY | INSTRUCTION |
|---|---|
| DOS | LOAD |
| DOS | PUT |
| BASIC | DISK OPEN * |
| BASIC | DISK CLOSE * |

* for both DIRECT and SEQUENTIAL files.

In the DOS commands LOAD and put, the DOS routines which perform these functions essentially supply the inputs to the CALL and SAVE commands respectively.

In addition, CALL and SAVE are called multiple times if necessary; once for each track which is in the file. OSI BASIC or assembly language source files begin at $3179. The first 5 bytes of each file are listed below:

```
MEMORY ADDRESS    FUNCTION

12,665 $3179    LO\   /SOURCE START
12,666 $317A    HI/   \ADDRESS

12,667 $317B    LO\   /SOURCE END
12,668 $317C    HI/   \ADDRESS

12,669 $317D    # OF TRACKS REQ

12,670 $317E    START OF USER AREA
```

Let's examine what happens when a file is loaded as called by the command:

```
LO 45
```

The LOAD routine evokes the CALL routine passing it the information:

```
CA 3179=45,1
```

After the track has been loaded into memory, the LOAD routine determines is there is another track. If there is then the CALL routine would be evoked again with the information:

```
CA 3D79=46,1
```

Note here that the LOAD command had to determine that the first track called has 12 pages on it.

This procedure is continued until the entire file is in memory. The LOAD command then passes control back to the calling routine. If the LOAD command:

```
LOAD "FILE"
```

had been used, where "FILE" is a file name, then an extra routine must be called to read the diskette directory to ascertain what track the file begins on.

The PUT command functions in a similar manner. The command: PU 45 would cause the PUT routine to call the SAVE routine passing it the information: SA 45,1=3179/C. After the track has been written to the diskette the PUT routine determines if there is another track. If there is, then the SAVE routine would be called again with the information: SA 46,1=3D76/C. The PUT command has the number of pages to place on a track stored in DOS.

The other special disk buffers which were listed were those used by BASIC for devices #6 and 7. If the standard OSI method is employed to use these buffers then the "CHANGE" program must be employed to create them. If one buffer is to be used it is device #6, and if two are to be used then device #7 is also used. There is a discussion in the OSI manual about creating these disk buffers and how to move your BASIC program around if you have already entered the program source code and then remember that you need one or more disk buffers. This point seems rather confusing to a number of people. The problem lies in the fact that OSI did not give themselves enough "hooks" in their BASIC program file or in the file header itself and therefore the user must do manual operations to set up disk buffer(s).

As stated before, the BASIC program file begins loading at $3179 and the user has the memory from this location through the end of memory to work with. If the user's program needs disk I/O then the user must provide memory for the disk buffer(s) out of his allocation from $3179 and up. OSI chose to have the disk buffer(s) reside immediately after the 5 byte file header and before the BASIC source code. For example, if one disk buffer of the normal OSI size of 12 pages is needed, running the CHANGE program and telling it to leave 12 pages and no extra bytes will cause location $3179 and $317A to be loaded with $7E and $3D, respectively. These are the first two locations of the program file buffer which contains the starting address of the BASIC source code. Note that these locations usually contain $7E and $31, respectively, or the location of the sixth byte of the program file buffer, when disk buffers are not used in the program. The BASIC interpreter looks at $3179 and $317A to determine where the BASIC source code starts. Since the disk buffer(s) are tucked away here, the program, when called, can use memory from the end of the BASIC source code through the end of memory for variables as it usually does. The drawback is that when the program file is stored, everything from the first byte of the file header through the last byte of the BASIC source code is saved. If there is a disk buffer(s) resisent in there it also is saved and takes up a (two) track(s) on the disk.

An alternative to using these standard OSI buffers is to assign your own buffers. This is very easy to do with one or two lines at the beginning of the BASIC applications program. The appropriate information must be entered into the locations listed below.

| MEMORY | ADDRESS | FUNCTION |
|---|---|---|
| 294 | $0126\ | /FIRST BYTE OF |
| 295 | $0127/ | \BUFFER DEVICE #6 |
| 296 | $0128\ | /LAST BYTE OF |
| 297 | $0129/ | \BUFFER DEVICE #6 |
| 428 | $01AC\ | /NEXT INPUT BYTE |
| 429 | $01AD/ | \DEVICE #6 |
| 451 | $01C3\ | /NEXT OUTPUT BYTE |
| 452 | $01C4/ | \DEVICE #6 |
| 302 | $012E\ | /FIRST BYTE OF |
| 303 | $012F/ | \BUFFER DEVICE #7 |
| 304 | $0130\ | /LAST BYTE +1 OF |
| 305 | $0131/ | \BUFFER DEVICE #7 |
| 509 | $01FD\ | /NEXT INPUT BYTE |
| 510 | $01FE/ | \DEVICE #7 |
| 534 | $0216\ | /NEXT OUTPUT BYTE |
| 535 | $0217/ | \DEVICE #7 |

For example, if you have a 48K system, your RAM is located from $0000 thru $BFFF (0 thru 49151). Let's create two buffers in upper memory fordevices #6 & 7; each being 13 pages long (3328 bytes). It does not matter to the computer which buffer is where, so I choose to put device #6 at the very top and device #7 just below it. The folowing code could be greatly streamlined; however, it is presented here in this fashion to provide an explanation of the process.

```
10 MP=48*4:POKE296,MP:POKE297,0:POKE294,
MP-13:POKE295,0
20  POKE428,MP-13:POKE429,0:POKE451,
MP-13:POKE452,0
30 POKE304,MP-13:POKE305,0:POKE302,
MP-26:POKE303,0
40 POKE509,MP-26:POKE510,0:POKE534,
MP-26:POKE535,0
50  POKE132,255:POKE133,MP-27:POKE8960,
MP-27:CLEAR
```

In line 10, "MP" is the maximum memory page +1. The first two pokes set up the end address +1 for device #6 buffer and the last two pokes setup the start address for the device #6 buffer. The pokes in line 20 initialize the input and output pointers for device #6 buffer. The pokes in line 30 & 40 do for device #7 what the corresponding pokes in line 10 & 20 did for device #6. Line 50 loads the location of the last byte of RAM available to BASIC (LOC 132 & 133) and the maximum memory page (LOC 8960). The CLEAR causes BASIC to update its pointers for end of memory, from locations 132 & 133. The maximum page is initially loaded by the system of power up by testing the memory and need not really be changed for the program to work. However, if your program bombs and you perform statements in the BASIC immediate mode, the values in locations 132 & 133 get reset sometimes. If you GOTO back into the program and the pointers have been updated, the BASIC interpreter will write string variables over the disk buffers.

Now that the actual data transfer and storage locations have been covered, let's examine the manner in which BASIC uses a sequential file. Executing the following command causes DOS to read the first track of the file into the disk buffer.

```
     DISK OPEN,6,"FILE"
```

If the program reads from the file with an INPUT#6, the DOS returns the characters from the beginning of the file thru the first carriage return ($0D) character. The next and subsequent input statements return the characters from after the previous carriage return character until the next one. This process continues as the user's program does inputs until the DOS reads the last character in the disk buffer; then the DOS reads the next track in to the disk buffer, starts at the beginning of the buffer again, and finishes reading the characters thru the carriage return character. This continues thru the end of the file at which time DOS will give you an error message if you attempt to read past the end of the file.

The process of writing to a sequential disk file is similar to reading. The data written out is entered into the next location in the disk buffer and a carriage return character is inserted. When the last location in the disk buffer has been written into, the DOS writes the disk buffer to the diskette and places the rest of the current message in the disk buffer starting at the beginning again.

It should be pointed out here that BASIC terminates its input strings with a carriage return and ignores miscellaneous non-printable characters such as line feeds. The BASIC PRINT statement always terminates with a carriage return and line feed. The line feed can be eliminated by the following procedure:

```
10 CR$=CHR$(13)
.
.
.
100 PRINT#6,"USER DATA";CR$;:PRINT#9
```

The CR$ variable in the print statement puts a carriage return in the disk buffer, the ";" after CR$ causes the print statement to be **extended** or continued, and the print to device #9 finishes the print with a carriage return and line feed to the null device (bit bucket). This one byte per data item may or may not be worth while. If your data is a series of 3 digit numbers for example this trick will reduce each data entry from 5 bytes (3+CR+LF) to 4 byte (3+CR) for a 20% space savings or a 25% increase in data storage.

After all operations to a give file have been done the file is closed with the command: DISK 6,CLOSE. When you close a sequential file you have written to, DOS puts an end of file marker after the last data written by the user and then forces a write of the buffer to the disk. If you have been reading, the disk buffer is only written out. The DOS routines could be upgraded considerably to make them more intelligent as to just when disk I/O was needed.

The use of direct (random) access files is very similar to the use of sequential files. However, in this case, the program indicates which record is to be read; DOS calculates what track it is on and reads that track into the disk buffer. I don't care for the OSI direct file handling capabilities at all and obtain much better performance both in the number of disk accesses and in data density from a subroutine package written in BASIC which handles the disk with CALL and SAVE commands. Once again, the trick of circumventing the line feed character per item in each record can be saved. For example, in one of my applications there are 14 items per record and I needed records 114 characters long. At this size 29 records fit on one track. If the normal **print had** been used which included the line feed character, records 128 (114+14) characters long would have been **required** and only 26 records would have fit on a track. The extra 3 records per track add 228 (3X76) records to the capacity of the disk. (An article about user control of direct files was printed in Vol 2 #3 p.10 of the Aardvark journal.)

If the standard OSI disk routines are used for random files, then the input **or** output to a particular record, say 8, is placed between the following statements.

```
DISK GET 8
.
.
.
DISK PUT
```

The GET command loads the proper disk track into the disk buffer and sets up the input and output pointers to the address of the first byte in the buffer of the referenced record. Random files need to be closed also.

In conclusion, the disk buffer, in general, is a section of RAM which must be allocated out of the section of RAM which is in the user's area. The buffer is only a holding area which can be either read from or written to at will. It is structured in this manner to minimize disk accesses and increase execution speed.

## HIDING MACHINE LANGUAGE ROUTINES
by Don VanSyckel

I have read several methods of loading machine language routines for use with BASIC programs. However, the method presented here for disk users requires the routine(s) to be loaded by the user or program only once and requires little effort to do it then. First write and debug your routine using the Assembler and Extended Monitor. When the routine executes properly, assemble it to upper memory with an offset such that the origin is really $317E (12,670). Save the code in a track sector which is only the required number of pages to contain the code.

When you begin to write the BASIC program run the "CHANGE" program first. When asked about "EXTRA BYTES" before BASIC, enter the size of the disk track sector that the assembly language code is stored in (256 X # of pages). After exitting from the "CHANGE" program, use the CALL command in the DOS KERNEL to load the disk track sector containing the machine language code to $317E. When you PUT your BASIC program, the DOS will save everything from $3179 (12,665) to the end of the BASIC source code. The first five bytes are the disk file header and from $317E to the end of memory is the user's area.

The BASIC program can call the routine(s) in either of two ways, either with USR or GO. Using the USR function one parameter may be passed to and one back from the assembly language routine(s). The USR requires some set up pokes so BASIC knows where the assembly language routine starts. If only one routine is called the pokes only need to be done once. However, if more than one routine is called then the pokes must be done before each USR. The GO, from the DOS KERNEL, can be used directly as DISK!"GO 317E" although no parameters can be passed directly. However, if needed BASIC can poke numbers into memory and the machine language routine can read them there. The ideal place for this "mailbox" is immediately after the machine language routine because there is probably space left over from rounding up to the next whole page size when leaving room or if not enough room is there a few extra bytes can be asked for initially in the CHANGE program. It should be noted that the return in the assembly language routine(s) must be a RTS and that more than one assembly language routine can be stored together in front of the BASIC source code. The entry points for each of the various routines must be known though. One other thing which the user must be careful to do is to clean up the 6502 stack at the end of the routine so that the RTS returns to the proper place.

## HOW TO SAVE ROM BASIC PROGRAMS ON DISK
by Wolfgang Baer

There are three ways of saving ROM Basic programs on OSI minidisk:

(1) The "normal" way is to make use of the cassette recorder. The ROM Basic programs are saved on cassette. Then you type (BREAK) D to start Disk Basic. After killing BEXEC* by NEW you can take the new program from cassette. Put the input on the cassette interface by POKE8993,1. This input will end with a reset to the keyboard if a syntax error is encountered on tape (as the "OK" message). Two things are important: Be sure you have a clean start of the listings on cassette, without the word "LIST" or any garbage because this will cause a reset. And because Disk Basic is much slower than ROM Basic, it will need more time to chew each line of Basic. So make the pauses between the lines longer before you save them from ROM Basic by POKE13,30. If you want to put a Disk Basic program on tape, do it by POKE8994,3: LIST line no. and afterwards POKE8994,2. Or you can LIST#1, line no. but will have no listing on the screen. Afterwards, you can load the program into ROM Basic from tape.

(2) After starting the disk system by (BREAK) D, type EXIT. Afterwards, you can make a coldstart by (BREAK) C. Give an answer to MEMORY SIZE? The lower 8K bytes are not used by diskette Kernel, so your memory size is 8192. Now you can work in ROM Basic. When your program is ready to be saved, type (BREAK) M 2A51 G. Now you're in diskette Kernel and can treat the ROM basic program as if it were machine code. You will have to save everything from zero page to the end of the program, as indicated in $007C. If you find an 1F in $007C (the highest possible number), then you will have to use four tracks, 2K Byte each. Zero page goes to disk by SA track,1=0000/1 (that is the only way), the rest of the first two Kbytes by SA track ,2=0130/7. If $007C is higher than $07, you go on: SA second track,1=0800/8 and so on. The program comes back from disk by CA 0000=first track, 1 CA 0130=first track,2...CA 1800=fourth track,1. Because all pointers and vectors have been saved together with the program, you can now go directly to Warmstart. If you do any chages to the program while editing, be sure to copy everything including zero page back on disk.

(3) The ROM Basic program starts normally at #0301. This is indicated at $0079 and $007A. If you change these locations together with the variable=end of program pointers at $007B and $007C, you can set that way your ROM Basic program to work on the disk Basic program that starts at $327E, with the

workspace pointers at $3279..327C. If you have a program on disk, you can reach it by ROM Basic with the following commands:

(BREAK) D to get Disk basic to work.
(BREAK) C to start ROM Basic. Give an answer to MEMORY SIZE? If you have 32 Kbytes and 2 Kbytes of USR routines at the top, your memory size is 30720. Otherwise, the memory test would destroy the disk kernel and the program you want to work on. Now you take the machine code monitor and copy the locations $3279..327C to $0079..007C. You can do this by hand or by this machine code utility:

```
A2 03       LDX #$03
BD 79 32    LDA $3279,X START
95 79       STA $79,X
CA          DEX
10 F8       BPL START
4C 74 A2    JMP WARMSTART
```

In my USR package starting at $7800, this program is at $7E1E. So I type:

(BREAK) M 7E1E G

If you have taken a Basic program from disk, it must appear now and can be used. Vice versa, a new ROM Basic program can be transferred to disk system by copying the locations $0079..007C to $3279..327C:

```
A2 03       LDX #$03
B5 79       LDA $79,X   START
9D 79 32    STA $3279,X
CA          DEX
10 F8       BPL START
4C 3C 22    JMP EXIT
```

So because this transfer program is at $78F1, I type:

(BREAK) M 78F1 G.

It ends with a jump to the disk Basic EXIT routine. It makes the kernel appear by "A*" and display the number of tracks needed. However, the first time, it displays nonsense **and** must be called again by GO 223C. Disk Basic will not be ready to work now because it has been damaged by the Coldstart. First put the program to disk, then power up again and recall it from disk. Be aware that the same tokens are used by ROM Basic for LOAD and SAVE as in Disk Basic for EXIT and DISK. If you want to change a ROM Basic program to run on Disk Basic, do not type statements as DISK OPEN #6,A$ because they will not be properly encoded by ROM Basic. Type instead SAVE OPEN #6,A$, and later Disk Basic will translate this into the right keyword. I want to point out again that it is not clever to declare disk buffers at the bottom of Basic, as the OSI CHANGE utility does. When saving programs on disk, the buffers will be put on disk together with the program and by this, waste one or two tracks. Always space off the disk buffers at the top of RAM!

## BASIC PLUS (OSI C1P)

by Patrick Townson, Ill.

BASIC PLUS is a machine code routine designed as an addendum to the Basic loaded into pages zero, one and two upon cold start. It needs to be loaded only one time, upon power up, and will remain until power is turned off. Subsequent cold and warm starts need only have the vectors re-pointed.

In the past, a variety of utility routines have been devised to help the C1P programmer. Most of them require calling by USR(X), and the locations for the routines were not standardized to allow all of them to run at one time. The program below does this. Once loaded, any of the special routines for screen clear, list, port audit, load and save can be accomplished with one button control. USR(X) is freed up to do other jobs. It need not be typed in. Pokes 11,12 need not be pointed at any special location (except as you wish for your own jobs elsewhere).

The program enclosed erases itself and leaves a poked up machine code in page two and in a small portion of page zero. The results are:

CONTROL L to load from tape (exit via space bar, as usual)
CONTROL Z to Save to tape. (exit via Control L and space bar, as usual)
CONTROL R to Run program. (exit as usual - via Control C or normal termination)
ESCAPE KEY to exit from Basic and audit activity at the ACIA port.
RUB OUT KEY for instant clearing of screen (or scroll window)
LINE FEED to list program, and while listing, CONTROL S to temporarily stop the listing flow; CONTROL Q to resume the listing flow. (you may alternate back and forth between S/Q as desired) However, for selective line listing (e.g. "LIST 100") you must still use the word "LIST" followed by specifics, since Line Feed in effects calls for List and gives CR at the same time, without allowing specifics to be requested. Thus, the S/Q feature.

After a warm or cold start: You must re-position the vectors,as follows: AS ONE LINE IN **IMMEDIATE MODE:**
POKE536,78:POKE537,2;POKE538,48
POKE539,2. Don't forget that on some older model C1P machines, the first command entered after warm start give an OM Error. So to avoid typing the above line twice, first enter some dummy command like Print P.

Although most machine codes offering utility functions seem to be located beginning at Hex 0222 (decimal 546), my own particular machine has the Aardvark Edit Rom located (in part) at those locations. So I have devised the following program to begin at 560 decimal-where Aardvark's ROM leaves off. However, the user can place the routine wherever desired-even in the high portion of RAM. But the most logical place is page two (and part of page zero) since these areas are rarely used otherwise, and don't **waste** any usable memory (since the Basic program enclosed washes itself after running).

If the users want to relocate the program (for example, to begin at Hex 0222) care should be taken to change all the JSR's, JMP's, etc. to relate everything to each other. In particular, lines 150-160 will need attention. Also, lines 6 and 170 will have to be carefully re-arranged. Caution should be observed in moving the routine to other locations, since I have carefully woven them together relative to the 560-660 area of page two. Many of these routines were available elsewhere, quite independent of each other. The JSR's in particular just might cause alot of trouble if you break up (or move) the program.

For ease in loading, you might want to have the program self start after the tape has been run. While still in the SAVE mode after line 200 has passed, type in the immediate mode, Carriage Return, Carriage return, POKE515,0: RUN Carriage return. The above added to the tape will cause the machine to execute and erase the program as soon as you have loaded it. If all goes well, you should then be able to press "RUB OUT" and get an immediate screen clear.

The so called Port Auditing Routine (Escape Key) allows the programmer to review the contents of a tape (or any activity at the Back Door) while another program is operating, without loading the activity t the port inadvertently on to the current program. To use it, start running the tape to be examined, then hit "Escape". The characters on the tape will fill up your screen, until the space bar is hit.

If you don't like the use of L/Load; Z/Save and R/Run, you can change them as follows: (after program is run). From A to Z is from 1 to 26. To change load: POKE594,XX-Save:POKE601,XX-Run:POKE622,XX.

Normal values of these locations, of course, is 594(12); 601(26); 622(18). Line feed value is 10, which means it can also be worked by Control J.

BASIC PLUS - A machine code routine which provides additional control code features to users of OSI C1P Basic in Rom machines.
PURPOSE: When operative, provides user with Control L to load; Control Z to Save; Control R to Run; (RUBOUT) for instant screen clear; (ESCAPE) to audit port without loading or disturbing current run.
CAUTION: Basic does not load over this, so it does not have to be re-run after a cold start. But after a cold start or warm start, you must reset vectors: POKE536,78:POKE537,2:POKE538,48:POKE539,2.

```
6   For X equals 560 to 660
10  Read A:POKEX,A:NEXT
14  REM 560-589 is Control Output S/Q
15   DATA72,169,246,141,0,223,169,192,
44,0
20    DATA223,208,12,169,252,141,0,223,
169,192
30   DATA44,0,223,208,244,104,76,105,
255,0:REM 589 REM S/Q listings
70   DATA32,186,255,201,12,208,3,32,139,
255,201:REM 12 is L/LOAD
80  DATA26,208,3,32,150,255,201,127,208,
3,76:REM 26 is Z/SAVE
90
DATA127,2,201,10,208,3,76,181,164:REM
620 REM 10 is Line Feed/LIST
100  DATA201,18,208,6,32,119,164,32,194,
165,96:REM 18 is R/RUN
110  REM  lines  120-130 for  Rubout Key
Screen Clear routine
120 DATA72,169,32,162,0,157,0,208,157,0,
209,157,0,210
130 DATA157,0,211,232,208,241,104,96
140  REM  Lines  150-160  Set  Input and
Output vectors.
150
POKE11,78:POKE12,2:POKE536,78:POKE537,2
160 POKE538,48:POKE539,2
165  REM Pokes 216-235 is the Port
Auditing Routine
170  For  X equals 216 to  235: Read Y:
POKE X,Y:NEXT
180 DATA169,255,141,3,2,44,3,2,16,9,
32,186,255,32,45,191,24
190 DATA144,242,96
200 NEW:REM No usable memory  lost.This
erases  itself and leaves a  poked  up
machine  code in paes 2(560-660)  and a
little in page 0(216-235
```

TO SELF START, WHILE SAVING TO TAPE, ADD "CR/CR POKE 515,0:RUN CR"

## SCREEN PRINT by John Price

The following is a listing and sample printout of a "Screen Print" routine that copies the contents of the screen to a printer, for an OSI C1P. These are my comments on the program, by line number.

10. Dimensions an array (Screen) the size of the screen area to be printed. Sets upper left & right corners of the screen area & set X & Y to their initial values.

20. Starts reading the area of the screen to be printed & increments X by 1.

30. Peeks the screen location to determine what is displayed there.

40. Stores the ASCII value of the character at each location in the array.

50. If we have reached the right side of the screen we go to LN70 if not, continue to line 60 which takes us to the next screen location to the right. (LN 60 next)

70. Increments Y by 1 to provide proper storage in the array. Increments C & D by 32 to move us down 1 line on the screen and resets X to 0.

80. Determines if the program has reached the bottom of the screen area we wish to print. If we have, it moves to line 100 which starts the output portion of the program. If not, it falls thru to line 90 which returns us to the screen reading loop.

100. Turns on printer port. (RS232)

110 & 120. Provide X & Y values for reading array Screen, from left to right & top to bottom.

130. Sends the proper code (number) to the printer and screen to print what was on the screen. If we have read a complete line (24 characters) the "Print" provides a "CR" for the printer.

160. Shuts off RS232 and terminates program.

The sample run is how the screen (TV with RF modulator) looked when the program (minus LNS 1,2,&3) was listed and the program was then run.

The tape has the program on both sides.

If this is of no use, burn the paper in your fireplace & find a good use for the tape.

PROGRAM LISTING

```
1     REM SCREEN  PRINT  JOHN PRICE
KNOXVILLE IA
2    REM PROGRAM  SCANS VIDEO MEMORY
LOCATIONS THEN PRINTS
3   REM THOSE CONTENTS TO A LINE PRINTER
(WRITTEN 12/15/81)
10  DIMS(24,21):C=53413:D=53437:X=0:Y=1
20  FORA=CTOD:X=X+1
30  Z=PEEK(A)
40  S(X,Y)=Z
50  IFX=24GOTO70
60  NEXT
70  Y=Y+1:C=C+32:D=D+32:X=0
80  IFD>54045GOTO100
90  GOTO20
100 SAVE
110 FORY=1TO21
120 FORX=1TO24
130 PRINTCHR$(S(X,Y));:IFX=24THENPRINT
140 NEXT
150 NEXT
160 POKE517,0:END
OK
```

SAMPLE RUN

```
 10 DIMS(24,21):C=53413:
D=53437:X=0:Y=1
 20 FORA=CTOD:X=X+1
 30 Z=PEEK(A)
 40 S(X,Y)=Z
 50 IFX=24GOTO70
 60 NEXT
 70 Y=Y+1:C=C+32:D=D+32:
X=0
 80 IFD>54045GOTO100
 90 GOTO10
 100 SAVE
 110 FORY=1TO21
 120 FORX=1TO24
 130 PRINTCHR$(S(X,Y));:
IFX=24THENPRINT
 140 NEXT
 150 NEXT
 160 POKE517,0:END
OK
```

## USING THE VIDEO MOD II ON DISK SYSTEMS
by Ronald C. Whitaker

One of the hazards of making modifications to a standard system is that unexpected incompatibilities may later appear when the system is expanded. In my case, I had already added an OSI 610 Disk Controllerboard to my C1P and it was up and running OK. Then I read about the Video Mod II which increases the visable display from 24 X 24 to 32 X 32. (It's actually 28 X 32 on my monitor). This sounded like a good idea at the time so I ordered plans for the mod. At that time, AARDVARK was not yet recommending the mod only for non-disk systems. I soon received the plans for the mod and a new 5.40672 mhz. crystal and made the required changes and additions to my system. The mod worked fine on the first try and I immediately like the new display. All was fine until I tried to boot the disk and it wouldn't. I rechecked all wiring and found nothing out of place. Next I plugged into the schematics of the 600 and 610 boards and the video mod and came up with the following modification.

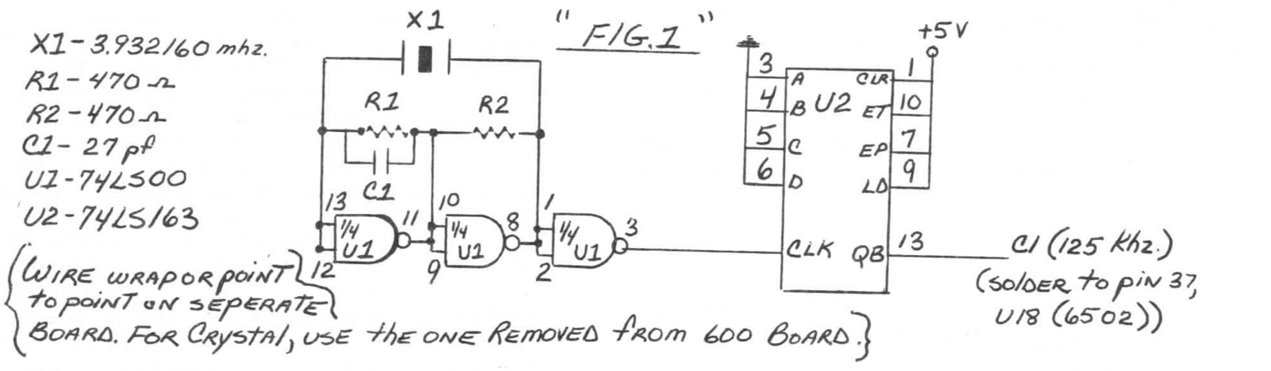
The standard time base (about 4.0 mhz) goes to a 16 stage divider chain formed by U30, U59, U60 and U61. These 16 outputs are labled C0 to C15. Each output in this chain is 1/2 the speed of the previous one. Thus, C0 produces a clock of 2.0 mhz, C1 is 1.0 mhz and so on through C15 which produces 60 hz. These various divided timing signals are used to drive the reset of the system. Of particular interest is C1 (1.0 mhz) which drives the microprocessor and C4 (125 khz) which produces the transmit clock (TX CLK) for the cassette interface. Several others are used to produce the video timing signals.

The Video Mod II works by replacing the original crystal with a faster one. This speeds up all of the timing signals which outputs the video characters to the display faster so more will fit on the screen. So far, so good. The timing of the cassette transmit clock is also increased so the cassette baud rate is increased from 300 to 412. This is only a problem if you want to load someone elses cassettes recorded at 300 baud. Since most of us do, the Video Mod includes a circuit which divides the faster C4 signal back down to 125 khz.

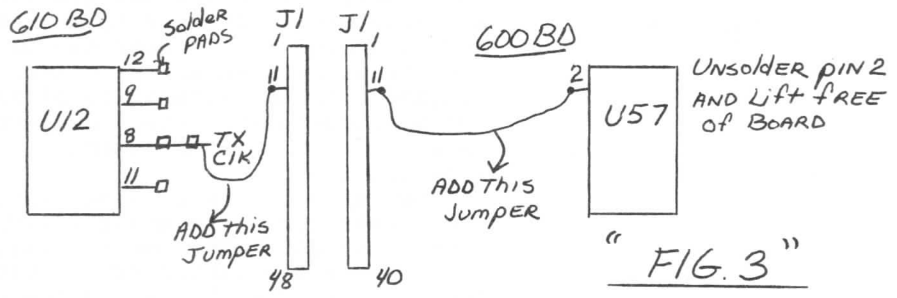
As already mentioned, the mod makes the microprocessor run faster too. This would be fine except that the microprocessor produces another clock called the Phase Two clock (02) which drives many components of the system including the transmit and receive speed of the disk controller. Since disk input and output is very timing sensitive, any changes will render the disk inoperative. The solution then is to not change the disk timing. To do this I chose to duplicate the original 4.0 mhz time base and the first stages of the divider chain on a separate piece of perf board. I used wire wrap for construction. The schematic for this is shown in Fig. 1. I then cut the trace from pin 37 of U18 (00 input to the microprocessor) and pin 13 of U30 (C1 output of divider chain). The output from the duplicate 4.0 mhz timebase is then soldered to the cut trace on the pin 37, U18 side. This is shown in Fig.2. This replaces the original timing to the microprocessor, the phase 2 clock, and the Disk transmit and receive. Now the disk works and so does the Video Mod.

I previously mentioned a circuit provided with the Video Mod which changes the too fast C4 signal back to 125 khz so the cassette interface operates at 300 baud. When you have the 610 disk controller board the additional circuit is not needed. There is already a 125 khz signal on the 610 board which may be routed through the unused pin 11 on the J1 connector between the 600 and 610 boards and then to pin 2 of U57 on the 600 board. This is shown in Fig. 3.

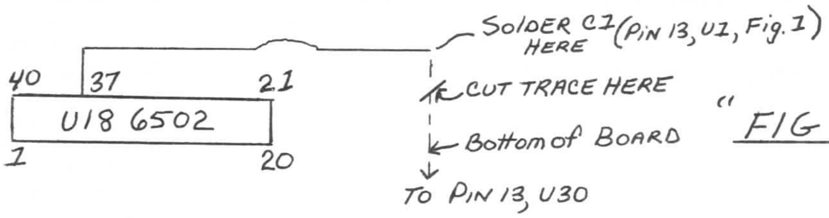
I'm sure there are other ways of using the Video Mod and a disk system together. This is the one that worked for me. Hopefully, it will allow those of you with disk systems to also use the Video Mod and AARDVARK can resume selling the Video mod for all C1P and Superboard systems.


X1-3.932160 mhz.
R1-470Ω
R2-470Ω
C1-27pf
U1-74LS00
U2-74LS163
X1
R1
R2
C1
1/4 U1
+5V
U2
A
B
C
D
CLR
ET
EP
LD
CLK
QB
C1 (125 Khz.)
(solder to pin 37, U18 (6502))
(Wire wrap or point to point on seperate board. For Crystal, use the one removed from 600 Board.)

"FIG.1"



"FIG. 3"



"FIG 2"

CHARLES HEPNER, STERLING HEIGHTS, MI

The following is a listing of my 6502-OP CODE table Generator program and the 6502 OP-CODE table it generates. This program came to me as a by-product of my Basic Dis-assembler program. In that program, I wanted to be able to determine the mneumonic for each OP-CODE without building a string array to do it.

The program listing is pretty straight forward. Lines 910 thru 995 are the data statements required for decoding and lines 1000-1020 is a little sub-routine to convert decimal to its character string hex equivelent. Lines 2000 thru 2090 are the main line program and take care of formatting and printing the table and the associated column headers.

```
RUN2000


     6502 ØP-CØDES IN DEC & HEX WITH MNEMØNICS & INSTRUCTIØN LENGTHS


DEC HX ØP L DEC HX ØP L DEC HX ØP L DEC HX ØP L DEC HX ØP L DEC HX ØP L
--- -- -- - --- -- -- - --- -- -- - --- -- -- - --- -- -- - --- -- -- -

0   00=BRK1 1   01=ØRA2 2   02=?    3   03=?    4   04=?    5   05=ØRA2
6   06=ASL2 7   07=?    8   08=PHP1 9   09=ØRA2 10  0A=ASL1 11  0B=?
12  0C=?    13  0D=ØRA3 14  0E=ASL3 15  0F=?    16  10=BPL2 17  11=ØRA2
18  12=?    19  13=?    20  14=?    21  15=ØRA2 22  16=ASL2 23  17=?
24  18=CLC1 25  19=ØRA3 26  1A=?    27  1B=?    28  1C=?    29  1D=ØRA3
30  1E=ASL3 31  1F=?    32  20=JSR3 33  21=AND2 34  22=?    35  23=?
36  24=BIT2 37  25=AND2 38  26=RØL2 39  27=?    40  28=PLP1 41  29=AND2
42  2A=RØL1 43  2B=?    44  2C=BIT3 45  2D=AND3 46  2E=RØL3 47  2F=?
48  30=BMI2 49  31=AND2 50  32=?    51  33=?    52  34=?    53  35=AND2
54  36=RØL2 55  37=?    56  38=SEC1 57  39=AND3 58  3A=?    59  3B=?
60  3C=?    61  3D=AND3 62  3E=RØL3 63  3F=?    64  40=RTI1 65  41=EØR2
66  42=?    67  43=?    68  44=?    69  45=EØR2 70  46=LSR2 71  47=?
72  48=PHA1 73  49=EØR2 74  4A=LSR1 75  4B=?    76  4C=JMP3 77  4D=EØR3
78  4E=LSR3 79  4F=?    80  50=BVC2 81  51=EØR2 82  52=?    83  53=?
84  54=?    85  55=EØR2 86  56=LSR2 87  57=?    88  58=CLI1 89  59=EØR3
90  5A=?    91  5B=?    92  5C=?    93  5D=EØR3 94  5E=LSR3 95  5F=?
96  60=RTS1 97  61=ADC2 98  62=?    99  63=?    100 64=?    101 65=ADC2
102 66=RØR2 103 67=?    104 68=PLA1 105 69=ADC2 106 6A=RØR1 107 6B=?
108 6C=JMP3 109 6D=ADC3 110 6E=RØR3 111 6F=?    112 70=BVS2 113 71=ADC2
114 72=?    115 73=?    116 74=?    117 75=ADC2 118 76=RØR2 119 77=?
120 78=SEI1 121 79=ADC3 122 7A=?    123 7B=?    124 7C=?    125 7D=ADC3
126 7E=RØR3 127 7F=?    128 80=?    129 81=STA2 130 82=?    131 83=?
132 84=STY2 133 85=STA2 134 86=STX2 135 87=?    136 88=DEY1 137 89=?
138 8A=TXA1 139 8B=?    140 8C=STY3 141 8D=STA3 142 8E=STX3 143 8F=?
144 90=BCC2 145 91=STA2 146 92=?    147 93=?    148 94=STY2 149 95=STA2
150 96=STX2 151 97=?    152 98=TYA1 153 99=STA3 154 9A=TXS1 155 9B=?
156 9C=?    157 9D=STA3 158 9E=?    159 9F=?    160 A0=LDY2 161 A1=LDA2
162 A2=LDX2 163 A3=?    164 A4=LDY2 165 A5=LDA2 166 A6=LDX2 167 A7=?
168 A8=TAY1 169 A9=LDA2 170 AA=TAX1 171 AB=?    172 AC=LDY3 173 AD=LDA3
174 AE=LDX3 175 AF=?    176 B0=BCS2 177 B1=LDA2 178 B2=?    179 B3=?
180 B4=LDY2 181 B5=LDA2 182 B6=LDX2 183 B7=?    184 B8=CLV1 185 B9=LDA3
186 BA=TSX1 187 BB=?    188 BC=LDY3 189 BD=LDA3 190 BE=LDX3 191 BF=?
192 C0=CPY2 193 C1=CMP2 194 C2=?    195 C3=?    196 C4=CPY2 197 C5=CMP2
198 C6=DEC2 199 C7=?    200 C8=INY1 201 C9=CMP2 202 CA=DEX1 203 CB=?
204 CC=CPY3 205 CD=CMP3 206 CE=DEC3 207 CF=?    208 D0=BNE2 209 D1=CMP2
210 D2=?    211 D3=?    212 D4=?    213 D5=CMP2 214 D6=DEC2 215 D7=?
216 D8=CLD1 217 D9=CMP3 218 DA=     219 DB=?    220 DC=?    221 DD=CMP3
222 DE=DEC3 223 DF=?    224 E0=CPX2 225 E1=SBC2 226 E2=?    227 E3=?
228 E4=CPX2 229 E5=SBC2 230 E6=INC2 231 E7=?    232 E8=INX1 233 E9=SBC2
234 EA=NØP1 235 EB=     236 EC=CPX3 237 ED=SBC3 238 EE=INC3 239 EF=?
240 F0=BEQ2 241 F1=SBC2 242 F2=?    243 F3=?    244 F4=?    245 F5=SBC2
246 F6=INC2 247 F7=?    248 F8=SED1 249 F9=SBC3 250 FA=?    251 FB=?
252 FC=?    253 FD=SBC3 254 FE=INC3 255 FF=?


  ? = NØT STANDARD ØP-CØDE



READY
```

```
LIST910-2999

 910 DATABRK1,ØRA2,?,?,?,ØRA2,ASL2,?,PHP1,ØRA2,ASL1,?,?,ØRA3,ASL3,?
 915 DATABPL2
 920 DATAØRA2,?,?,?,ØRA2,ASL2,?,CLC1,ØRA3,?,?,?,ØRA3,ASL3,?,JSR3,AND2,?
 925 DATA?,BIT2,AND2,RØL2,?,PLP1,AND2,RØL1,?
 930 DATABIT3,AND3,RØL3,?,BMI2,AND2,?,?,?,AND2,RØL2,?,SEC1,AND3,?,?,?
 935 DATAAND3,RØL3,?,RTI1,EØR2,?,?,?,EØR2,LSR2,?,PHA1,EØR2,LSR1,?
 940 DATAJMP3,EØR3,LSR3,?,BVC2,EØR2,?,?,?,EØR2,LSR2,?,CLI1,EØR3,?,?,?
 945 DATAEØR3,LSR3,?,RTS1,ADC2,?,?,?,ADC2,RØR2,?,PLA1,ADC2,RØR1,?
 950 DATAJMP3,ADC3,RØR3,?,BVS2,ADC2,?,?,?,ADC2,RØR2,?,SEI1,ADC3,?,?,?
 955 DATAADC3,RØR3,?,?,STA2,?,?,STY2,STA2,STX2,?,DEY1,?,TXA1,?,STY3
 960 DATASTA3,STX3,?,BCC2,STA2,?,?,STY2,STA2,STX2,?,TYA1,STA3,TXS1,?,?
 965 DATASTA3,?,?,LDY2,LDA2,LDX2,?,LDY2,LDA2,LDX2,?,TAY1,LDA2,TAX1,?
 970 DATALDY3,LDA3,LDX3,?,BCS2,LDA2,?,?,LDY2,LDA2,LDX2,?,CLV1,LDA3,TSX1
 975 DATA?,LDY3,LDA3,LDX3,?,CPY2,CMP2,?,?,CPY2,CMP2,DEC2,?,INY1,CMP2
 980 DATADEX1,?,CPY3,CMP3,DEC3,?,BNE2,CMP2,?,?,?,CMP2,DEC2,?,CLD1,CMP3,
 985 DATA?,?,CMP3,DEC3,?,CPX2,SBC2,?,?,CPX2,SBC2,INC2,?,INX1,SBC2,NØP1,
 990 DATACPX3,SBC3,INC3,?,BEQ2,SBC2,?,?,?,SBC2,INC2,?,SED1,SBC3,?,?,?
 995 DATASBC3,INC3,?
 1000 HI=INT(D/16):LØ=D-(HI*16)
 1010 C$=MID$(H$,HI+1,1)+MID$(H$,LØ+1,1)
 1020 RETURN
 2000 I=0:Y=0:H$="0123456789ABCDEF"
 2001 FØRX=1TØ3:PRINT:NEXT
 2002 PRINT"  6502 ØP-CØDES IN DEC & HEX WITH MNEMØNICS & INSTRUCTIØN ";
 2003 PRINT"LENGTHS":PRINT:PRINT:PRINT
 2004 FØRX=1TØ6:PRINT" DEC HX ØP L";:NEXT
 2006 FØRX=1TØ6:PRINT" --- -- -- -";:NEXT:PRINT:PRINT
 2010 FØRX=1TØ255
 2012 READØP$
 2020 D=X:GØSUB1000
 2040 PRINTD;TAB(5+12*Y);C$;"=";ØP$;TAB(12+12*Y);
 2050 Y=Y+1:IFY>5THENY=0
 2060 NEXTX
 2080 PRINT:PRINT:PRINT:PRINT"    ? = NØT STANDARD ØP-CØDE"
 2090 FØRX=1TØ3:PRINT:NEXT:END
READY
```

## AND YET ANOTHER JOYSTICK INTERFACE

by Ted Mahler

By using only two I.C.'s and a handful of components you can easily convert readily available linear joysticks for use on a Superboard II. An LM2901 quad comparator is used to compare the voltage level of the joystick pots with the voltage level of reference pots. This comparison yields a voltage which is used to simulate a keyboard key closure.

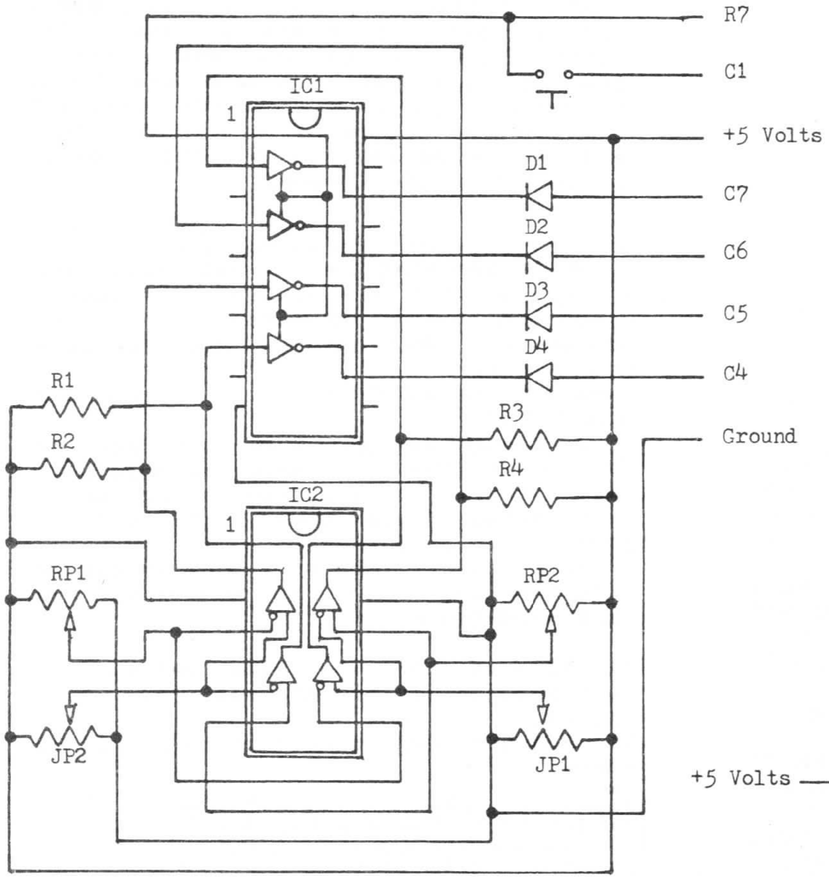
As you can see in figure 1, each comparator in the LM2901 has a joystick pot (JP1 or JP2) connected to one input and a reference pot connected to the other input. One of the reference pots voltage level is set below the voltage level of the joystick in the neutral position while the otherreference is set slightly above. This setup results in each comparator having its inverting input higher than its non-inverting input yeilding a low output. By moving the joystick to the left, the wiper voltage of JP1 will increase. Pins 9 and 10 of the LM2901, being connected to JP1, will also show this voltage increase. This will reverse the conditions on the comparator whose inputs are on pins 8 and 9 and will change its output to a high. This will simulate a key closure of the "1" key. Note that the conditions on the comparator whole input pins are 10 and 11 has not changed even though pin 9 has increased in voltage. When the joystick is moved right, the conditions on that comparator are reversed and the "2" key is decoded. By moving the joystick up, the wiper voltage on JP2 increases. This reverses the voltages on the comparator on pins 4 and 5 of the LM2901 and provides a high output decoding the "3" key. This comparator's mate, on pins 6 and 7, will change its output to a high when the joystick is moved down, giving a "4" key closure. By moving the joystick between any of these four positions, two adjacent outputs will go high. This provides four additional directions from the joystick. The pushbutton is the "7" key.

The comparator outputs, via a pull-up resistor, are buffered and inverted through a 74LS240 octal buffer/line driver. Pin 1 of the buffer is the enable for its 3-state outputs. This enable is strobed by the R7 line from the Superboard keyboard connector (J4 in the OSI schematics). Outputs for the buffer are connected to J4 as shown in figure 1.

Layout and construction of the joystick is, of course, up to the builder. Ifyou want the joystick positions to decode the same keys as I have decoded, then orient the joystick assembly as shown in figure 2. Also connect the +5 volts and ground such that the JP1 wiper voltage increases when moving the stick to the left and the JP2 voltage increases when the stick is moved up. All decoding circuitry was enclosed in a box containing the joystick, with an 8 pin ribbon connector linking it to a socket on the back of my computer case. You may wish to place this circuit in the computer and use six wires for the connection to the joystick.

Set up of the circuit is straight forward with only two adjustments being necessary, reference pots 1 and 2 (RP1 and RP2). With the joystick in the neutral position, adjust RP1 such that the voltage on pin 4 is approximately .250 volts above the voltage on pin 5. Likewise adjust RP2 such that the voltage on pin 11 is .250 volts below the pin 10 voltage. After you have the circuit operational you may want to adjust these references to obtain a better "feel" on the joystick. The closer the reference voltages are to the joystick voltages, the more sensitive the stick is to movement.

I have used this joystick directly with the AARDVARK Galaxia game. Other program can be modified to use this joystick by changing the control keys or by modifying the hookup.



| | |
|---|---|
| R1,R2,R3,R4 | 3900 ohms |
| D1,D2,D3,D4 | 1N914 |
| RP1,RP2 | 10,000 ohm pot |
| JP1,JP2 | Joystick pots |
| IC1 | 74LS240 |
| IC2 | LM2901 |

(Equal to or greater than JP1 or JP2)
(5000 ohms for this joystick model)

FIGURE 1. Joystick interface schematic.

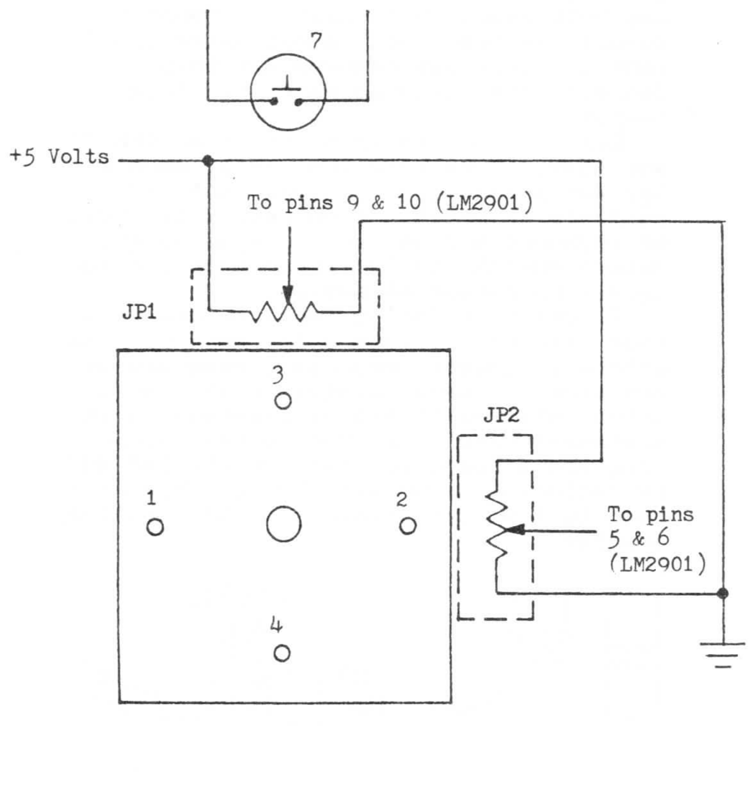


FIGURE 2. Physical layout of the joystick.

## RS-232 & 110 BAUD MODS FOR OSI C4P
by Donald Brennan

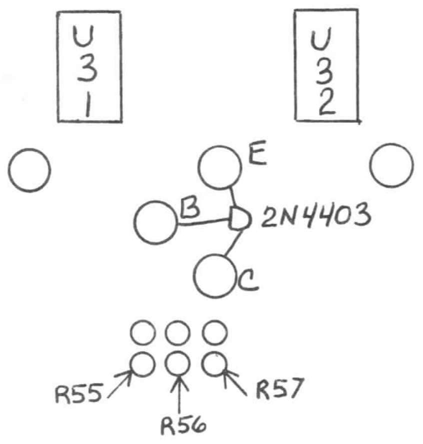
RS-232 MOD

On the 502 board install a 14 pin socket & insert a 7404 at U31 position.

Populate the TX Data Output circuit with a 2N4403 at Q2, a 10K 1/4W at R55, a 10K 1/4W at R57 and a 470 1/4W at R56.

Cut land at U41 pin 2 going to P/J3 pin 7. Add wire on P/J1 pin 24 to pin 27.

Use a twisted pair with signal wire going to P/J3 pin 7 and ground wire going to P/J3 pin 8. The other end goes to KSR-33 input terminal strip. Pin 7 and pin 6.



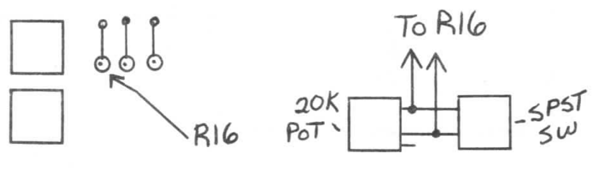
110 BAUD MODIFICATION

The 110 baud mod is obtained by adding a 20K pot and SPST switch in series with R16 and R17 of 555 timer circuitry on the 502 board. This pot is shorted out for normal 300 baud operation.

Lift end of R16 out of hole and bend up. Attach a twisted pair to lifted end and hole where R16 went. Mount the sw-pot on the back panel where the AC control jack was removed for this mod. Connect the twisted pair to this sw-pot.

Set switch to open. Turn on ASR 33 and C4P. Type save and hold down any key for printout. Adjust pot for readable printout. When set, type rest of keyboard and adjust it if necessary. Return switch to 300 baud position for normal operation of system.

To printout during C4P disk operation type list #1. This will send data to printer. Inyour programs, have one of the line numbers state: DV=1. Have print statements state: print#dv,"print statement here". You can modify other programs already written for the C4P DOS by replacing DV=2 with DV=1. This will reroute the data from the CRT to the printer.



JOSEPH TAVARES, NEW BEDFORD, MA

Thanks to your Journal and C2E Rom I finally found out how to program my C2. One thing I noticed is that there doesn't appear to be a program that uses circular motion on the OSI computer. This short program will demonstrate how this is done:

```
5 FORX=1TO35:?:NEXTX:POKE56900,0
10 L+53967:POKEL,219
15 R=7:PI=3.14159
20 FORA=0TO2*PISTEP.1
25 X=INT(R*COS(A)):Y=INT(R*SIN(A))*64
30 POKEL+X+Y,226
35 FORT=0TO6:NEXTT
40 POKEL+X+Y,32
45 NEXTA
50 GOTO20
```

When this program is run, you will see a circle revolving around a cross. If lines 35,40 and 50 are removed the program will draw a large circle. R=the radius and can be changed for different size circles. L=center of circle. This program can be used for planetary motion or an analog clock face.

BOBBY AKINS, APO SAN FRANCISCO

While perusing the Journal (Vol.2 #1) I realized that no mention has been given to Number Converting for Addresses. Your keyboard routine is a good example:

Keyboard is at $FE00 yet your pokes are 0 and 253!

I thought perhaps a means of getting from one to the other might be helpful. This little quick convertor can be manipulated faster than a conversion program can be loaded.

This manipulation will work in either direction and is quick enough for low cost programming.

1) Reverse the two Poke values corresponds to how address' are loaded.

2) Convert each number to an 8 bit BCD number.

| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| | | | | | | | |

EXAMPLE: 253=11111101/
0=00000000

3) Arrange together example:
11111101/00000000

4) Separate into 4 each 4 bit numbers
Example: 1111/1101/0000/0000

5) Starting at left look up each 4 bits per the look up table.

Example:

| F | E | 0 | 0 |
|---|---|---|---|
| 1111 | 1101 | 0000 | 0000 |

LOOK UP TABLE

| | |
|---|---|
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0010 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |

CONT'D

| | |
|---|---|
| 8 | 1000 |
| 9 | 1001 |
| (10) A | 1010 |
| (11) B | 1011 |
| (12) C | 1100 |
| (13) D | 1101 |
| (14) E | 1110 |
| (15) F | 1111 |

The important thing to remember is that there are an awful lot of these POKEs and PEEKs which you can use without knowing anything about machine code and even if you are a rank amateur and rank beginner. For instance, location 15 on the BASIC in rom system contains the number of characters to be printed before a line feed. Thats very handy to know. For some C1 systems you loose a character off the end of the line if you allow the system to to print 24 characters wide. If you, however, choose a shorter line length when you power up, you will find that you can't make tapes that can be reloaded back into the system. The tape interface requires that you have a line length of greater than 71 characters. As long as you know the location where the length of the line is stored you can POKE a number in there to shorten the line while you are programming and POKE another number in there to lengthen the line when you make a tape. After you have the system running, you can then execute "POKE 15,22" to shorten the line length and then execute "POKE 15,72" before you make a tape.

The First Book of OSI by Williams and Dorner, Carlson's OSI BASIC in ROM, MICRO, COMPUTE, and the Aardvark Journal all contain references to handy memory locations. Most of these you can use without knowing anything at all about machine code and by knowing very little about BASIC.

If you have a disk BASIC system, you have even more freedom. In the disk BASIC system very little is stored in ROM. The entire BASIC line is normally put into RAM. That means that you can modify virtually any part of your BASIC language by simply knowing where it is stored on the disk and knowing enough to either interpret the code or at least enough to read the manual where it is interpretted for you. For instance, it is very annoying to most BASIC users that you can't input a comma or a semicolon in the middle of an input statement. It sometimes makes it difficult to put togethor things like word processors and report writers when you can't input punctuation into the machine. Fortunately, the OSI documentation shows you where the section that finds the delimiters is located and tells you how to POKE them out of existance.

The first reason the POKEs are handy then is that some things have to be stored in RAM because they vary. By being able to manipulate them directly you can expand capabilities of the BASIC and set it up for conditions in which the original authors did not envision.

The second reason why PEEKs and POKEs are handy is that the people who wrote the BASIC for your system tried to write one basic to run on a lot of different machines. They're no dummies. It's expensive to write a BASIC and to cut new ROMs for every machine. Therefore, to make the BASIC more universal, they tend to write one 6502 BASIC and instead of doing things like putting input and output routines into the BASIC, they tell the BASIC to refer to certain vectors to find those things which vary from machine to machine. For instance, the location of the keyboard and decoding of the keyboard may vary from machine to machine as it does between the C1 and the C2/4/8 and the Apple machines. Therefore, rather than decoding the BASIC in ROMs, they put that decoding elsewhere in another ROM and put the address of that ROM in a location in memory which the BASIC looks at to find out where to find the keyboard or where to find the video screen or where to find the cassette port and all the other things that vary from machine to machine. This has the effect of making the BASIC more universal and it also allows us to get in and put in our own input and output routines. For instance, on the C1P the BASIC ROMs do not know where the keyboard is or how to interpret it. They do know that in locations 536 and 537 they will find an address which will point them toward the keyboard routine for whatever computer they happen to be sitting in. (BASIC in ROM does not know if it is in a C1 or C4) The same thing is true for the output routines. They are not in BASIC ROMs but the BASIC ROMs know that they are at an address which is pointed to by the information in 538 and 539.

That allows us to do some neat things by putting our own addresses in there and by pointing the basic to our own routines. For instance, the cursor control program which was published by Aardvark, and similar ones which have been published in PEEK 65 and MICRO, work by pointing basic to a new input routine which we poke into existance ourselves. In a similar manner, we used to market a high speed load and save tape routine for the C1P which worked by intercepting the load and save vectors and redirecting the machine to our own routine. While the beginning user may not be sophisticated enough to write such routines for himself, a number of them are available in old issues of MICRO, the AARDVARK Journal, PEEK 65 and similar sources which even the beginner can implement.

The third thing that makes PEEKs and POKEs handy is that some things look like memory locations which really aren't memory locations. Printer ports, cassette ports, and video screens look a lot like memory locations and work the same way. For instance, while your system will use a UART or PIA for cassette I/O, it will look to the system as if there was one memory location that when you put things into they go to the cassette and another memory location where things magically appear from the cassette. Knowing that, we can manipulate some of the peripherals directly. Some of this is simple enough for almost anybody to do and it's a lot of fun. For instance, in the old Aardvark catalogs we printed a PEEK A PORT utility which worked by pretending that the cassette input was a cassette location. It printed to the screen whatever it found in that memory location and therefore told you what was going by on the cassette tape.

CONT'D

## PEEK A PORT UTILITY

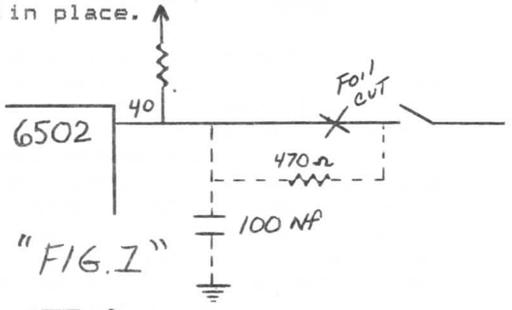
```
4 REM PEEK A PORT UTILITY
8 REM C2/4/8 VALUES
10 A=64512:B=A+1
20 WAITA,1:PRINTCHR$(PEEK(B));:GOTO10
35 REM C1 VALUES
40 A=61440:B=A+1
50 WAITA,1:PRINTCHR$(PEEK(B));:GOTO50
```

This is a two line program. C2/4/8 users enter lines 10 and 20. C1 users enter lines 40 and 50.

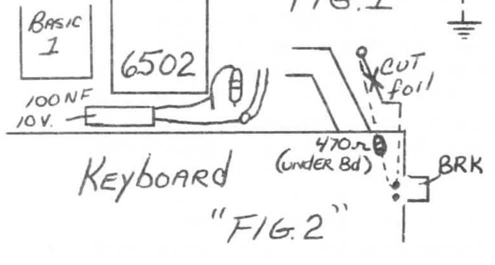
There are, of course, a couple of other reasons why Peeks and Pokes are used, but they are somewhat beyond the scope of this article. They are, of course, the basic method by which graphics is done with the OSI systems. We've covered that in so many previous articles that I don't think it needs to be repeated here. They also are a handy method for storing and manipulating machine code programs from a basic system. That, however, is somewhat beyond the scope of the beginners article and will be covered in the upcoming issues on machine code usage.

JOHN SEYBOLD, MADISON, WISCONSIN
Here is an easy way to add a short delay to the break key on a Superboard or C1P. This in only for the models that do not already have a delay circuit. The mod is shown schematically in fig 1 and pictorally in fig 2. All we're doing is adding a delay to the reset line of approximately R*C seconds. With the values shown, I got about a 1/2 sec delay. To increase the delay, simply use a larger capacitor. Do not try using a larger resistance unless you install a larger pull up resistor on pin 40 of the 6502. This circuit will also force the computer to come up already reset when the power is turned on.

When I made the mod to my Superboard, I installed the resistor on the underside of the board between the break key and the hole next to the trace cut. I advise you to sleeve the leads if you use this approach. The capacitor was mounted on top of the board as shown. You may wish to experiment with different capacitor values before soldering one in place.



"FIG.1"



"FIG.2"

EARL MORRIS, MIDLAND, MI
FIXES FOR ALIEN RAIN

To make the "Bullet" more visible:

```
45 MP=MP-V:IFPEEK(MP)=UTHENPOKEMP,F:
POKEMP+V,U:GOTO45
46 POKEMP+V,U
```

To avoid leaving garbage when a bomb drops using level one:

```
190 FORB=0TODD:POKEDB(A)+B,F:
POKEDB(A)-B,F:NEXT
```

To printout the score after a miss to show the loss of 5 points:

```
130 IFPM(SRTHENSC=SC-5:GOTO120
```

To make the program easier to follow and a little faster, TL(CC) can be replaced with a simple variable TL in lines 225, 235, 240, 260 and 420. Likewise DB(7) can be replaced with D7 in lines 135, 140, 150, 155, 160, 165, 170, 230 and 420.

PERSONAL NOTE FROM C. THOMAS HILTON

Mr.Hilton noticed an article in one of our Journals for a customer who is blind. He has offered his assistance to any "handicapped" individual who may need it. C. Thomas Hilton
P. O. Box #7
Deer Lodge, Montana
59722

## RUBIK'S CUBE
by John Wright, Canada

Rodgers rules of games programs says you
should not bother with games, it is easier to play in their original form. This program is legitmate on two counts -first, that I have trouble following through on planned moves on the cube itself -second, my son won't let me.

I had thought that it might be easier to see patterns with the problem in two dimensions, but so far this has not come true.

The cube is displayed as an upside down crucifix. The faces are:

```
  6
  3
2 1 5 (don't ask why)
  4
```

The program asks which face you wish to deal with and how many counter clock clockwise rotations you wish to apply. Three counter clockwise equals one clockwise. If the number of rotations is zero, then the face called for is moved to the '1' position.

The program itself is quite simple but the coefficients were laborious to work out. The picture is kept in MA(2, , ) which is a 9 X 12 grid. The corners are filled with blanks and letters for the colours occupy the rest. A new position is mapped into MA(1--). When the new poistion is complete it is copied back into MA(2--) and POKEd to the screen. The screen could have been used instead of MA(2--) but it seemed easier to use a second matrix than PEEKs. As intermediate steps are completed the screen is updated.

The first block of data is used for swapping the faces around. Line 36 moves face 5 to position 1 and is read as:

-Move face 1 to p2. Do not rotate.
-Move face 2 to p6. Rotate twiçe.
-Leave face 3. Rotate 1 turn Cw.
-etc.

Line 50 is the low numbered corner of each face.
Line 65 is the coefficients needed to rotate a face in SUB600.
Lines 80 through 90 are the coefficients to rotate an edge attached to a rotating face. For 65 and 80-90 you need faith or a pile of scrap paper.
Line 100 reads in a new cube and then offers the opportunity to overwrite it with whatever is in lines 560 and 570. The data on these lines is my son's cube as of 12 nov. 81-9p.m. The program is in lines 110-160. The subroutines are preceeded by REMs describing their functions.

The ME matrix remembers 100 moves and will print them. If 7 is input in response to FACE?. If you have more than 8K, then MS can be increased.
Manipulating the matrices is quite slow. Machine language routines would help. Taking out SUB350 would speed it up a little, but the intermediate updates would not appear.

```
 1 REM RUBIK'S CUBE
 2 REM SOFTWRIGHT (613) 731-8521
 3 REM V1.2, 12 NOV 81
 5 OO=53608:LL=32:REM THESE ARE FOR 600
BOARD
 10 DIM MA(2,9,12),MC(6,6,2),MR(6,2),CO(
6,3),CF(6,6,4),CC(6)
 14 MS=101
 16 DIM ME(MS,2):N=1
 20 FORI=2TO6:FORJ=1TO6:FORK=1TO2:READMC
(I,J,K):NEXTK,J,I
 30 DATA 5,0,1,0,3,3,4,1,6,2,2,2
 32 DATA 4,0,2,1,1,0,6,0,5,3,3,0
 34 DATA 3,0,2,3,6,0,1,0,5,1,4,0
 36 DATA 2,0,6,2,3,1,4,3,1,0,5,2
 38 DATA 6,0,2,2,4,0,3,0,5,2,1,0
 40 FORI=1TO6:FORJ=1TO2:READMR(I,J):NEXT
J,I
 50 DATA 4,7,1,7,4,4,4,10,7,7,4,1
 55 FORI=1TO9:FORJ=1TO12:MA(1,I,J)=32:NE
XTJ,I
 60 FORI=1TO6:FORJ=1TO3:READCO(I,J):NEXT
J,I
 65 DATA 7,6,10,4,6,10,7,3,7,7,9,13,10,6
,10,7,0,4
 70 FORI=1TO6:FORJ=1TO6:FORK=1TO4:READCF
(I,J,K):NEXTK,J,I
 80 DATA -1,0,4,0,14,-3,-4,2,-1,3,4,-2,2
,0,4,0,11,0,-4,0,2,3,4,-2
 82 DATA 4,0,0,0,-6,1,6,0,4,0,0,0,-3,1,6
,0,4,0,0,0,12,1,-6,0
 84 DATA -2,-1,6,0,7,0,0,0,-1,3,4,-2,-1,
0,4,0,16,-1,-6,0,7,0,0,0
 86 DATA -6,1,6,0,9,0,0,0,-1,3,4,-2,17,0
,-8,0,12,1,-6,0,9,0,0,0
 88 DATA 6,0,0,0,16,-1,-6,0,6,0,0,0,1,-1
,6,0,6,0,0,0,-2,-1,6,0
 90 DATA -7,0,8,0,2,3,4,-2,-1,3,4,-2,20,
0,-8,0,17,0,-8,0,14,-3,-4,2
 100 GOSUB510:GOSUB500:FORI=1TO18:PRINT:
NEXTI:GOSUB300
 110 INPUT"FACE";FA:IFFA>6THEN700
 120 INPUT"ROTATION";RO:FORI=1TO18:PRINT
:NEXTI:GOSUB350
 130 ME(N,1)=FA:ME(N,2)=RO:N=N+1:IFN=MS-
1THENPRINT"N="MS-1
 140 IFN>MSTHENN=1
 150 IFRO=0THENGOSUB400:GOSUB300:GOTO110
 160 FORL=1TORO:GOSUB200:NEXTL:GOSUB300:
GOTO110
 190 END
 199 REM ROTATE FACE IN QUESTION
 200 FB=FA:GOSUB600
 220 FORI=1TO3:FORJ=1TO2:FORIA=1TO6
 230 CC(IA)=CF(FA,IA,1)+CF(FA,IA,2)*I+CF
(FA,IA,3)*J+CF(FA,IA,4)*I*J
 240 NEXTIA
 250 MA(1,CC(1),CC(2))=MA(2,CC(3),CC(4))
 260 MA(1,CC(3),CC(4))=MA(2,CC(5),CC(6))
 270 NEXTJ,I:GOSUB300:RETURN
 299 REM SWAP MATRICES, PRINT
 300 FORMJ=1TO12:Z=OO+LL*MJ:FORMI=1TO9:M
A(2,MI,MJ)=MA(1,MI,MJ)
 310 POKEZ+MI,MA(2,MI,MJ):NEXTMI,MJ:RETU
RN
 350 FORNJ=1TO12:Z=OO+LL*NJ:FORNI=1TO9
 360 POKEZ+NI,MA(1,NI,NJ):NEXTNI,NJ:RETU
RN
 399 REM MOVE FACES AROUND
 400 FORK=1TO6:KB=MC(FA,K,2):IFKB=0THEN4
20
 410 FORKC=1TO4-KB:FB=K:GOSUB600:GOSUB35
0:NEXTKC
 420 NEXTK:GOSUB300:FORK=1TO6:KA=MC(FA,K
,1):IFKA=KTHEN450
 425 FORI=0TO2:FORJ=0TO2
 430 MA(1,MR(KA,1)+I,MR(KA,2)+J)=MA(2,MR
(K,1)+I,MR(K,2)+J)
 440 NEXTJ,I
 450 NEXTK:RETURN
 500 INPUT"START NEW OR INPUT";Q$:IFLEFT
$(Q$,1)="S"THEN RETURN
 510 BL$="   ":FORJ=1TO12:READA$:IFJ<7OR
J>9THENA$=BL$+A$+BL$
 520 FORI=1TO9:MA(1,I,J)=ASC(MID$(A$,I,1
))
 530 NEXTI,J:RETURN
 540 DATA RRR,RRR,RRR,BBB,BBB,BBB,GGGOOO
YYY,GGGOOOYYY
 550 DATA GGGOOOYYY,WWW,WWW,WWW
 560 DATA OGR,OOO,OOO,BBB,BBB,BBB,YYYRRR
GGG,YYYRRWGGG
 570 DATA YYRGRGOOY,WWW,WWW,WRW
 599 REM 3x3 ROTATION
 600 FORI=1TO3:FORJ=1TO3
 610 MA(1,CO(FB,1)-I,CO(FB,2)+J)=MA(2,CO
(FB,1)-J,CO(FB,3)-I)
 620 NEXTJ,I:FORI=1TO3:FORJ=1TO3
 630 MA(2,CO(FB,1)-I,CO(FB,2)+J)=MA(1,CO
(FB,1)-I,CO(FB,2)+J)
 640 NEXTJ,I:RETURN
 700 INPUT"DO YOU WANT MOVES LISTED";Q$:
IFLEFT$(Q$,1)<>"Y"THENEND
 710 INPUT"STARTING AT WHAT NUMBER";NA
 715 POKE517,1
 720 FORNB=NATONSTEP5:FORI=0TO4:PRINTME(
NB+I,1);ME(NB+I,2);
 730 NEXTI:PRINT:NEXTNB
 740 POKE517,0:GOTO110
 100 REM PROGRAM TO SET UP A NEW "AFILE"
 105 REM
 110 REM   1) USE "CREATE" TO MAKE A FIL
E OF ABOUT 4 TRACKS
 120 REM      CALLED "AFILE" FIRST, AND
RUN "ZERO" TO CLEAR IT.
 130 REM
 140 REM   2) NEXT RUN THIS PROGRAM TO I
NITIALIZE THE FILE
 150 REM      WITH 2 ANIMALS AND SET UP
STARTING POINTERS
 160 REM
 170 REM      REMEMBER TO RUN "CHANGE" T
O ALLOW ONE BUFFER
 180 REM      BEFORE TYPING THIS IN!!
 190 REM
 195 REM      IF YOU HAVE ANY PROBLEMS F
EEL FREE TO CALL ME:
 200 REM      GARY KAUFMAN 206 S.13TH ST
. PHILA,PA 19107
 210 REM      (215) 735-2841
 220 REM
 1000 INPUT"INITIALIZE ANIMAL FILE";A$
 1020 IFLEFT$(A$,1)<>"Y"THENEND
 1040 DISK OPEN,6,"AFILE"
 1060 DISK GET,0
 1080 PRINT#6,"2,4"
 1100 DISK PUT
 1120 DISK GET,1
 1140 PRINT#6,"2,3,DOES IT LIVE IN THE W
ATER?"
 1160 DISK PUT
 1180 DISK GET,2
 1200 PRINT#6,"!,MOOSE, "
 1220 DISK PUT
 1240 DISK GET,3
 1260 PRINT#6,"!,FROG, "
 1280 DISK PUT
```

```
 100 REM ANIMAL FILE GAME (MODELED AFTER
 APPLE VERSION)
 105 REM GARY KAUFMAN  206 S. 13TH ST. P
HILA,PA  19107
 110 REM (215) 735-2841
 115 REM *** REMEMBER TO USE CHANGE TO C
REATE ONE BUFFER
 116 REM *** BEFORE TYPING THIS IN !!
 117 REM
 120 C=1:R=0
 140 DISK OPEN,6,"AFILE"
 160 DISK GET,0
 180 INPUT#6,A$,L$
 200 A=VAL(A$):L=VAL(L$)
 220 FORI=1TO32:PRINT:NEXT:PRINTTAB(25)"
ANIMALS":PRINT:PRINT
 240 IFRTHEN380
 260 INPUT"INSTRUCTIONS";A$
 280 IFLEFT$(A$,1)<>"Y"THEN380
 300 PRINT:PRINT"YOU THINK OF AN ANIMAL,
 I WILL TRY TO GUESS THE
 320 PRINT"ANIMAL YOU ARE THINKING OF.
 340 PRINT"IF I DON'T KNOW THE ANIMAL I'
LL ADD IT TO MY FILE FOR
 360 PRINT"THE NEXT GAME.
 380 PRINT:PRINT"I KNOW "A" ANIMALS"
 400 DISK GET,C:INPUT#6,A$,B$,C$
 420 IFA$="!"THEN520
 440 WP=VAL(A$):RP=VAL(B$):PRINTC$:INPUT
A$
 460 IFLEFT$(A$,1)="Y"THENC=RP
 480 IFLEFT$(A$,1)="N"THENC=WP
 500 GOTO400
 520 PRINT:PRINT"IS THE ANIMAL A";
 540 A$=LEFT$(B$,1)
 560 IF(A$="A")OR(A$="E")OR(A$="I")OR(A$
="O")OR(A$="U")THENPRINT"N";
 580 PRINT" ";B$;:INPUTA$
 600 IFLEFT$(A$,1)="Y"THENPRINT"I GOT IT
 RIGHT!":GOTO1200
 620 PRINT"ALL RIGHT, I GIVE UP.  WHAT W
AS THE ANIMAL?"
 640 INPUTN$
 660 PRINT"WHAT WOULD BE A QUESTION THAT
 WOULD DIFFERENTIATE"
 680 PRINT"BETWEEN A";
 700 A$=LEFT$(N$,1)
 720 IF(A$="A")OR(A$="E")OR(A$="I")OR(A$
="O")OR(A$="U")THENPRINT"N";
 740 PRINT" ";B$" AND A";
 760 A$=LEFT$(B$,1)
 780 IF(A$="A")OR(A$="E")OR(A$="I")OR(A$
="O")OR(A$="U")THENPRINT"N";
 800 PRINT" ";N$:INPUTQ$:IFRIGHT$(Q$,1)<
>"?"THENQ$=Q$+"?"
 820 PRINT"WHAT WOULD BE THE CORRECT ANS
WER FOR A";
 840 IF(A$="A")OR(A$="E")OR(A$="I")OR(A$
="O")OR(A$="U")THENPRINT"N";
 860 PRINT" ";N$:INPUTT$
 880 DISK GET,C:INPUT#6,T1$,T2$,T3$
 900 DISK GET,L:A$=T1$+","+T2$+","+T3$:P
RINT#6,A$
 920 DISK PUT:L=L+1
 940 DISK GET,C
 960 A$=STR$(L-1)+","+STR$(L)+","+Q$
 980 IFLEFT$(T$,1)="N"THENA$=STR$(L)+","
+STR$(L-1)+","+Q$
 1000 PRINT#6,A$
 1020 DISK PUT
 1040 DISK GET,L
 1060 A$="!,"+N$+", ":PRINT#6,A$
 1080 DISK PUT:L=L+1
 1100 DISK GET,0:INPUT#6,A$,L$
 1120 A=VAL(A$)
 1140 A=A+1:A$=STR$(A)+","+STR$(L)
 1160 PRINT#6,A$
 1180 DISK PUT
 1200 C=1:R=1:GOTO140
```

CORRECTION by J. Kaposztas

There was a slight typographical error in David Kuhn's Reverse Video Mod in the Feb. issue of the Journal. The IC used as an XOR gate should be a 74LS86 not 74LS76 as mentioned in the diagram.

Also the place to cut the foil trace is on top of the 600 board about a quarter inch from pin #9 of U42 (the 74LS165) between the two plated through holes.

FOR SALE:

MITTENDORF HIGH RES BOARD, FULLY POPULATED. INCLUDES ALL JUMPERS AND DOCUMENTATION. MANY EXTRAS. $90.00 CALL JERRY MELE (216) 225-0157