# Open-Apple™

[redacted] [redacted]

ISSN 0885-4017
newstand price: $2.00
photocopy charge per page: $0.15

***Releasing the power to everyone.***

## Miscellanea

**An AppleWorks Rescue Routine** comes to us by a long and circuitous route through bulletin boards and user group newsletters. We've tested it and checked it out with its originator, Michael Wilks, author of Applied Engineering's *Super AppleWorks Desktop Expander Utility.*

The trick is useful if AppleWorks hangs while you have unsaved files on the desktop. It doesn't ALWAYS work, so don't depend on it completely. The routine was developed to speed testing of the desktop expander. When the routine does work, the only thing you should do is save what you were working on. Then reboot AppleWorks and reload your files.

When AppleWorks seems to hang, first wait for a few minutes. If the last command you gave was to remove or delete something, this can take awhile on an expanded desktop.

If nothing happens after a reasonable amount of time and you are convinced that AppleWorks is all twisted up in its own underwear, press control-reset until you break the knot and get into the Monitor. Reset will switch the screen to 40 columns and you will have an asterisk prompt on your screen. Then enter:

```
C073:0                              select bank 0 on big memory cards
3 control-P                         turn on 80-columns

   Note: if this doesn't put you in 80-columns, then:
       FF59G
       3 control-P
   If you can't get into 80-columns, you'll have to reboot.

2F0:2C 83 C0 2C 83 C0 4C 33 10      (Those are eights, not bees.)
2F0G
```

At this point you will be back to the AppleWorks main menu with your files intact. The choices on the main menu may be numbered strangely. If so, simply choose another menu (Add Files will do) and escape back to the main menu.

Remember, do nothing but save your files. For safety sake, save them on a different disk from the one you have been using—don't risk destroying your most recent back-ups. Then reboot and reload.

**The AppleWorks User Group**, mentioned here last August (page 61), has moved to Colorado. The group maintains a clearing house of templates, notes, reviews, and public-domain utilities for AppleWorks. At the moment they have 28 disk sides of stuff available for $3 per floppy (2 sides) or $7.50 per 3.5" disk (5 sides). The group also supports a bulletin board in Denver (303-756-5222) and is starting an AppleWorks discussion section on GENIE. For more information, send a stamped, self-addressed envelope to The AppleWorks User Group, P.O. Box 24869, Denver, CO 80224.

**Macros and mouse support come to AppleWorks** in Beagle Bros' newest offering, *MacroWorks*, by Randy Brandt. ($34.95 from Beagle Bros, 3990 Old Town Ave, San Diego, CA 92110). A "macro" allows you to enter a long string of characters or keystroke commands with a "single" keystroke. With *MacroWorks*, this single keystroke is a combination of the solid-apple key and just about any of the other keys on the keyboard.

Any sequence of keystrokes you enter so often as to be bothered by the idea of having to type it is a good candidate for macrification. Your name and address, the commands for printing a document, the keystrokes that save all the files on the desktop, the keystrokes for inserting a new month in a spreadsheet, the keystrokes for recording a dues payment from a member of your Nash Rambler User Group, the keystrokes for writing letters to your congressman...the possiblities are endless.

The package comes with a set of pre-defined macros. These are most useful in the word-processing module but will work in any of the three AppleWorks programs. You can use some or all of these supplied macros and you can write your own. The ten number keys, 0 through 9, can be defined and redefined while you are using AppleWorks. These ten keys are allowed macros of up to 70 keystrokes each.

Macros for other keys must be defined ahead of time, using an AppleWorks word processor file, and "compiled," using the *MacroWorks* program. These macros can be any length—the only limit is that all of them put together can take up no more than 4,095 bytes. One macro can call another. A macro can even call itself. An AppleWorks word processor starter file, holding the standard *MacroWorks* macros as well as comments and suggestions, is included in the package.

**MacroWorks** mouse support allows you to use the mouse to select menu items and to move the cursor around the screen in any of the three AppleWorks modules. By holding down the mouse button you can make the screen scroll.

A number of other utility programs are included in the package. One allows you to re-write the AppleWorks help screens. You can replace AppleWork's word processor-help screen with a helpful list of *MacroWork's* built-in macros, or you can create screens to help you with your own macros, or your assembler's directives, or your family's birthdates, or any other information you like.

The version of ProDOS supplied on this disk includes a program selector called *Bird's Better Bye*, by Alan Bird, in place of Apple's standard quit code. This is a beautiful little piece of work. When you quit from a ProDOS program (from Applesoft you do this with the BYE command), Bird's quit code gives you a list of the subdirectories and system programs on the current disk (rather than the standard ENTER PREFIX rigmarole). You can scroll through

the list and select the program you want to execute next by pressing Return. If you select a subdirectory, its contents will be displayed. If you press escape, the program switches to another volume. By repeatedly pressing escape you can flip through all the volumes that are online. The only limitation of this program is that it won't list or run Applesoft programs for you—it can only get you into system programs such as Basic.system. The big advantage it has compared to other program selectors, however, is that you don't have to configure it ahead of time—you can use it to find and start any system program in any subdirectory your computer can access.

*MacroWorks* works with all but the earliest versions of Applied Engineering's and Checkmate Technology's desktop expansion software. It does not, however, work with *Pinpoint*. It also doesn't support AE's printer buffer (no big loss—software-based printer buffers have never done anything for me but eat my files).

**Beagle Bros sent along some interesting statistics** derived from the first 500 *MacroWorks* registration cards it received.

```
Per cent of MacroWorks buyers using:

large RAM cards                     65%
   Applied Engineering   88%
   all other             12%
modem                               53%
mouse                               52%
color monitor                       35%
clock                               30%
UniDisk 3.5                         20%
hard disk                           12%
   Sider                 60%
   all other             40%
```

Early *MacroWorks* buyers are not representative of all Apple owners. Members of this group likely own more peripherals than the average Apple user. However, the proportions of peripherals owned are probably indicative of what is happening in the Apple world.

Two numbers stand out. First, large memory cards are by far the best selling peripherals on this list and Applied Engineering is by far the dominant supplier. Now you know how the company affords all those ads in the Apple magazines.

Second, the UniDisk 3.5 has quickly achieved a greater penetration than all hard disks put together, even though Apple's marketing wizards gave it the same name as their 5-1/4 inch drive, priced it about the same as a 10-megabyte Sider, and have refused to fix its major bug, which makes the write-protection tab on 3.5 inch disks dangerous to use (January 1986, page 98). Apple has known about the bug since before the UniDisk was released but has given absolutely no indication that it intends to fix it.

The UniDisk 3.5 has managed to overcome Apple's bureaucratic naming, pricing, and updating policies only because it is a wonderful little device. After using a couple of them for several months I have to admit they are quiet, rugged, and have lots of capacity. I hereby take back all the snotty comments I made about these drives in October (page 73), although I still think Apple was stupid for not building in DOS 3.3 support.

Hard drives have two advantages over the UniDisk—speed and maximum file size. Their disadvantages are that they are noisier, less reliable, and you can't remove the disk.

The speed advantage of hard disks is evaporating, however, because of all those RAM cards. A RAMdisk is faster than a hard disk, and, if backed up by batteries or an uninterruptable power supply, just as reliable. This leaves hard disks with just one advantage—the ability to hold very large files. You'll see that advantage melt away during the next couple of years as RAM prices continue to fall (more slowly than in recent months, perhaps, because of the strength of the Japanese Yen, but fall they will). The fastest, most reliable mass storage now available for the Apple II is a combination of an electrically-backed-up RAMdisk and a UniDisk 3.5.

Until recently RAMdisks weren't widely used in the Apple world for two reasons. One was expense. The second was that most of the available RAM cards became RAMdisks only with the help of RAM-based software patches and additions to the operating system. Apple crushed the second problem last fall with the introduction of its RAM expansion card, which included a RAMdisk driver in ROM on the card, where it couldn't be stepped on or otherwise destroyed.

**Now Applied Engineering has just given an incredible boost to RAMdisks** with its introduction of the RamFactor card. This card uses a standard slot, like Apple's. However, it has twice as much ROM as Apple's card, which gives its RAMdisk many additional abilities. Like a Sider hard drive, it is recognized as a storage device by ProDOS, DOS 3.3, Pascal, and CP/M. Like a Sider, the device can be partitioned, with certain segments of the space allotted to each operating system. With one megabyte of memory and a $179 battery back-up option, the card retails for $568, exactly the same as a UniDisk 3.5. It can be expanded up to 16 megabytes. The battery back-up option allows you to leave files on the card while the computer is turned off.

I think the introduction of this card marks the peak of auxiliary-slot schemes for adding memory to Apple IIs. Applied Engineering has modified its AppleWorks desktop expansion software so that it works with RamFactor—this software has been the major reason for getting AE's auxiliary-slot based card in the past. (AE's AppleWorks desktop expander even works on an Apple II-Plus with RamFactor and an 80-column card). The only disadvantage of the RamFactor card is that it requires a standard slot.

**The IIc—Apple Color Monitor interference problem** (February, page 2.8; April, page 2.23) made it to "read and post" service-notice status at Apple dealers as of May 1. As mentioned here in April, the problem is a missing shield on some IIc disk drives. Apple dealers are supposed to install a shield as a warranty transaction if your IIc is missing it. The same notice says that IIc stand-alone power supplies with date codes prior to 4584 (45th week of 1984) may induce waviness in the color monitor. Upcoming details from Apple will tell your dealer how to get you a replacement.

***The Apple II Plus/IIe Troubleshooting and Repair Guide*** is in its third printing already, though I just found out about it (the copyright date is 1984). It's by Robert C. Brenner and is the first Apple repair guide for non-technicians that I've run into. It has a great deal of general information on topics such as "Steps to Successful Troubleshooting," "Where Does Interference Come From?," and "Removing Solder."

But most importantly, it has page after page of very *specific* information on exactly which chips are suspect when your Apple suffers from various combinations of ill symptoms. It gives very detailed, conservative instructions on how to proceed if you want to make your own repairs. Anyone with responsibility for several Apples who's willing to use a chip puller should own this book. It was published by Howard W. Sams and Co and has a list price of $19.95; you can get it by mail order from S-C Software (P.O. Box 280300, Dallas TX 75228 214-324-2050) for $18 plus $2 shipping.

**Lost in the private domain.** Has Apple ever released an Applesoft support package for double-high resolution graphics? **Open-Apple** subscriber Bruce Ristow points out that it has—hidden away on a disk that came with Apple's auxiliary-slot RGB card, which is no longer sold. The package includes 18 ampersand commands for drawing and for printing 80-column text in double-high-resolution.

Also missing from Apple's most recent price lists is the *DOS (3.3) Programmer's Toolkit*. Now that Apple is listening to user groups, pressure from the right people might pop this stuff into the groups' public domain software libraries. Without the pressure, the stuff is totally lost—it's illegal to make copies and impossible to buy them.

**Apple Documentation Update:** A new packet of technical notes was published by the Apple II technical support team in early May. Here are the new items for those of you who want to update our March list of Apple documentation (page 2.11):

```
Technical Notes--January-April 1986
  ProDOS: 17
  IIe hardware: 1 through 8
  IIc hardware: 1 through 4
  Memory expansion card: 1
  Smartport (protocol converter): 1 through 3
  UniDisk 3.5: 1 through 4
  Apple II misc: 1 through 6
```

Most of these notes are pretty esoteric. *Apple II Miscellaneous Technical Note #3*, however, raises a problem at least a few **Open-Apple** readers have probably run into. Apple has recently discovered a bug in the Pascal protocol firmware on the Super Serial Card. The bug surfaced with the combination of Apple's *Access II* communications software, the UniDisk 3.5, and the Super Serial Card.

The problem has to do with the $C800-$CFFF ROM space that peripheral cards are supposed to share. The hardware part of the protocol is that each card should turn off its $C800 ROM whenever byte $CFFF is accessed and turn its $C800 ROM back on when its slot-dependent ROM space is accessed (slot-dependent ROM is at $CsXX—each slot gets $100 bytes in the range from $C100 to $C7FF). The software part of the protocol is that all uses of a card's $C800 ROM should be vectored through the card's slot-dependent

ROM and that the firmware on the card should tickle $CFFF before jumping into the $C800 area.

The Pascal entry points on the Super Serial Card, however, jump right into the $C800 ROM without referencing $CFFF. Apparently the bug hasn't surfaced before because the Super Serial Card could overpower most other cards whose ROM was still on. The UniDisk 3.5 card, however, uses the same chips to "drive the bus" as the Super Serial Card, and neither can overpower the other.

Thus, Pascal and assembly language programs that intend to use Pascal entry points with groups of cards that could include the Super Serial Card should do so something like this:

```
LDA ????      character out, status request number, etc.
LDX #$Cs      s=slot number of card (required by Pascal entry point protocol)
LDY #$s0                "                          "
STX MSLOT     MSLOT=$7F8, used by interrupt routines to fix $C800 ROM
STX $CFFF     turn off all $C800 ROMs
JMP entrypt   go to entry point (this turns the card's $C800 ROM back on)
```

Since Apple has promoted the Super Serial Card as a model that third-party developers should use when designing firmware for slot-based cards, it's likely that many third-party cards share this bug. However, the more commonly-used entry point on the card—the one Applesoft programs use—tickles $CFFF as it is supposed to and doesn't present a problem. The Pascal entry points have some advantages for assembly language programmers, however, and have been getting more use by developers recently.

## Picking Up Applesoft

### How much was that variable?

There are two situations in which it can be handy to get a list of all the variables a program is using and their current values. One is during program debugging. Scanning the values of a program's variables after a program-breaking error can be an immense help in finding the bug.

The second is when trying to figure out how programs that use ProDOS VAR files work. Apple's new system utilities program, for example, makes heavy use of VAR files. It's very helpful to have an easy way to find out what variables are in such a file and what their values are.

There are a couple of commerical programs available that, as one element of their many abilities, can display current variables and their values. One is Beagle Bros' *Double-Take*, by Mark Simonsen ($34.95, 3990 Old Town Ave, San Diego, CA 92110) and the other is Glen Bredon's *ProCMD* ($20, 521 State Rd, Princeton, NJ 08540). These are assembly language packages that are neatly hidden away from the Applesoft program you are working on.

Neither of them can dump the values in arrays, however. And neither of them teaches us as much about Applesoft as writing our own variable reader would.

As it happens, most of the background information needed to write such a program appeared in *Open-Apple* in April, in the article about using Applesoft with a RAMdisk (pages 2.17-2.21). Just for fun, let's build on that article and write a program called VAR.READER, that will work under either DOS 3.3 or ProDOS.

I've reprinted April's Figure 1 to jog *your* memory about *Applesoft*'s memory. Applesoft programs reside in the area from $800 (2048) to $9600 (38400) and are split into four major parts—the program image itself, a table of simple variables, a table of array variables, and a string storage area.

The variable tables hold the name of each variable and, in the case of numeric variables, the current value. For string variables, the table holds the length of the string and a pointer aimed at the string itself, which is usually stashed away somewhere inside the string storage area.

Reading the variable tables should be quite easy—all we have to do is scan them from beginning to end, dig out each variable's two-letter name, and talk Applesoft into displaying the current value.

For example, each item in the table of simple variables is seven bytes long. The first two bytes hold the two letters of the variable's name. By using a FOR loop with a step of seven, you can scan the table from beginning to end, pull out all the variable names, and determine what type each variable is. Then it's simply a matter of printing its value.

Since Applesoft keeps the address of the tables in standard two-byte assembly language pointers (with the high byte last), let's begin by devising a two-byte PEEK function that will return the value held in two-byte pointers. How about:

```
37700 DEF FN PK(ADR)=PEEK(ADR)+PEEK(ADR+1)*256
```

This statement defines a function you can call with such syntax as V=FN PK(X), or PRINT FN PK(X), where X can be a number or a formula that specifies the address of the first byte of the two-byte pointer you want to examine. The function returns the address stored in that two-byte pointer. (Note the difference between the address of the pointer, which is X in these examples, and the address the pointer is aimed at, which the function returns.) Once the PK function is defined, we can find the beginning and end of the simple variable table and scan it like this:

```
37811 TS=FN PK(105)   : REM TS = simple variable table's starting adr
37812 TE=FN PK(107)-1 : REM TE = table's ending adr
37820 FOR ADR=TS TO TE STEP 7 : REM scan 7-byte simple variables
```

As we scan the table, we want to find out the name and type of each variable. From April's article (page 2.19), we know that combinations of high- and low-ASCII characters are used to designate whether a variable is floating point, integer, a string, or a function.

To determine a variable's type, we can let T=0 as we start to examine it. If the first character is high-ASCII, we'll add 1 to T. If the second character is high-ASCII, we'll add 2 to T. Thus, T will end up 0 for low-low (floating point), 1 for high-low (functions), 2 for low-high (strings), and 3 for high-high (integers). Here's some definitions that will allow us to turn T into the normal variable identifier. They would appear at the beginning of our program, after the PK function definition:

```
37790 TYPE$(0)=" " : REM real
37791 TYPE$(1)=""  : REM function
37792 TYPE$(2)="$" : REM string
37793 TYPE$(3)="%" : REM integer
```

And here's a subroutine that actually digs out the name and type of the simple variable stored at ADR:

```
37980 REM * Read variable name and type *
37984 T=0
37985 C1=PEEK(ADR)   : IF C1>127 THEN T=T+1 : C1=C1-128
37986 C2=PEEK(ADR+1) : IF C2>127 THEN T=T+2 : C2=C2-128
37987 IF C2=0 THEN C2=ASC(" ")
37988 N$=CHR$(C1)+CHR$(C2)+TYPE$(T)
37989 RETURN
```

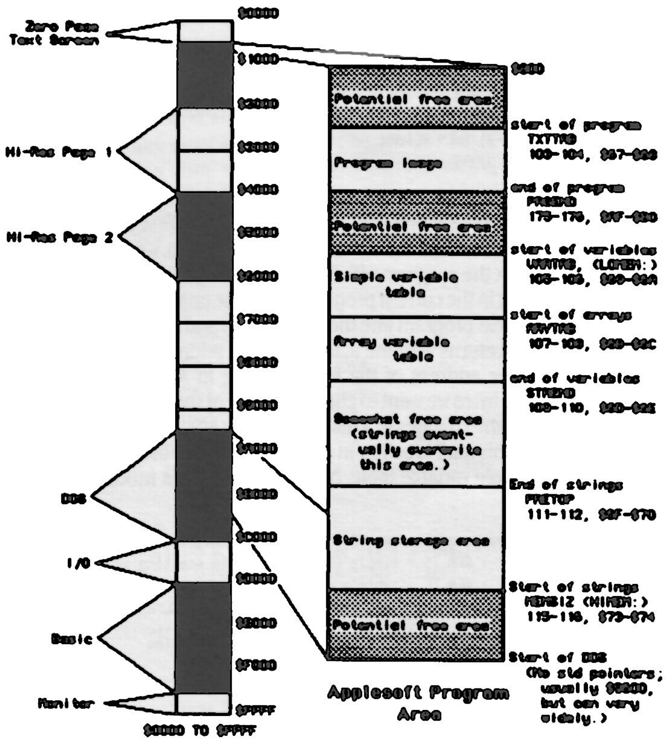


Figure 1

From April's article we also know that if a variable has a one-letter name, the second letter will be given the value $00 or $80. We need to change this to a blank—this is the reason for line 987. Notice that if a character was in high-ASCII, we subtracted 128 from it. This makes all characters low-ASCII for simplicity later on.

Since FUNCTIONs have no value associated with them, let's filter them out right away:

```
37822 : IF T=1 THEN PRINT "Function ";N$;" defined." : NEXT
```

Now we have captured the variable name in N$; all we have to do is print it and its associated value on the screen. PRINT N$;" = "; will get the name and an equal sign up on the screen, but how to do we tell Applesoft to print the value of the variable **in** N$ rather than the value **of** N$?

In Logo this is no problem. Logo has a function called THING that returns the value of a variable specified by another variable:

```
MAKE "X$ "CONFUSED?"          The Applesoft equivalent is X$="CONFUSED?"
MAKE "N$ "X$                   "        "            "    is N$="X$"
PRINT THING :N$               (No Applesoft equivalent available.)
CONFUSED?
```

Of course, in Logo this whole issue is of no significance anyhow—Logo has a built-in command, PONS (print out names), that prints out all active variable names and their values.

In Applesoft, there are two possibilities. We could poke the variable name somewhere in memory and then write an assembly language routine that would trick Applesoft into printing that value, or we could just poke the variable name right into a PRINT statement in our program.

When a program POKEs changes into itself it becomes *self-modifying*. Most professionals disdain such programs. That alone is probably reason enough to figure out how to do it.

Again, the April issue (page 2.18) has all the information we need to find a specific spot inside a program. For example, let's say the line we decided we wanted to modify looked like this:

```
37824 : PRINT N$;" = ";XX$
```

The idea would be to overwrite the XX$ with the name of the variable we really want to print. (At the moment the name of that variable is stored in N$.)

What we need is a subroutine that returns the beginning and ending memory addresses of any line in memory. In April we learned that the first two bytes of each program line, as they exist in memory, point to the next program line. The third and fourth bytes hold the line number of the current line. Applesoft aims a pointer called TXTTAB ($67, 103) at the first byte of the first line of the program. The last next-line pointer holds zeros. Thus:

```
37990 REM * Find line number L              *
37994 ACL=FN PK(103) : REM address of current line, 103=TXTTAB
37995 ANL=FN PK(ACL) : REM address of next line
37996 IF ANL=0 THEN PRINT "There's no line ";L;" in this program." : STOP
37997 IF L=FN PK(ACL+2) THEN RETURN
37998 ACL=ANL : GOTO 37995
```

Line 994 gets the address of the first program line out of TXTTAB. Line 995 gets the address of the second program line out of the first one. Line 996 looks to see if we've reached the end of the program; if so, we print an error message and break the program with STOP. Line 997 looks to see if the line number embedded in the current program line is the one we're looking for. If not, we make the next program line the current one and start over.

This subroutine returns with the actual memory address of the specified line in ACL and the address of the following line in ANL. In our current situation, the three bytes we want to change appear at the end of a line. Thus it's easiest to poke the changes in a few bytes below ANL.

Now let's put all this stuff together in one place—here's the entire loop for digging up the simple variable table. Note how line 824 is modified by line 823:

```
37810 REM * Display names and values of all simple variables *
37811 TS=FN PK(105)   : REM TS = simple variable table's starting adr
37812 TE=FN PK(107)-1 : REM TE = table's ending adr
37813 L=37824 : GOSUB 37990 : REM put address of line L in ACL and ANL

37820 FOR ADR=TS TO TE STEP 7 : REM scan 7-byte simple variables
37821 : GOSUB 37980 : REM get name and type of this variable
37822 : IF T=1 THEN PRINT "Function ";N$;" defined." : NEXT
37823 : FOR I=1 TO 3 : POKE ANL-5+I,ASC(MID$(N$,I,1)) : NEXT
37824 : PRINT N$;" = ";XX$
37825 NEXT
```

Now that we have the simple variables taken care of, let's look at what kinds of problems the array variables pose. As I said earlier, neither of the two commerical packages mentioned display the values of array variables, although Bredon's does report the dimensions of all active arrays.

Admittedly, dumping the array variables is more difficult than dumping the simple variables. The problem is that arrays have no set number of dimensions and no set number of elements per dimension. Any routine we write has to be quite flexible.

On the other hand, however, one advantage of working with arrays is that you *can* put most of an array variable's name in a second variable. For example, if A=1 and B=2, you can print the array element XX$(1,2) with the statement PRINT XX$(A,B).

Applesoft stores a pointer to the beginning of the array variable table at ARYTAB ($6B, 107) and points just past where the table ends with STREND ($6D, 109), so finding the table is no trouble. However, as we saw in April, the array table keeps getting moved to higher and higher addresses as new simple variables are used. For each new simple variable, Applesoft has to make room in the simple variable table. To do this, the array variable table has to be moved.

Thus, before we bother to determine exactly where the array table is, it's important that we activate all the simple variables that our array-dumping routine will use. By sneaking a look ahead, I know what they'll be, so here's how to find the array table (note that it's not necessary to reactivate the simple variables that have already been used to dump the simple variable table):

```
37911 A=0 : B=0 : C=0 : D=0 : E=0 : ND=0 : NV=0 : D1$="" : D2$=""
37912 NV=FN PK(107) : REM array table's start is Next Variable
37913 TE=FN PK(109) : REM table's ending address
```

Unlike simple variables, which are always exactly seven bytes long, array variables can be just about any length. It depends how many dimensions the array has and how many elements are in each dimension.

Consequently, the third and fourth bytes of each array table entry hold the distance to the next array variable entry. (The first two bytes hold the variable's name in the high/low-ASCII code we've come to expect.) Line 922 uses ADR to point at the variable currently being worked on and advances NV to point at the next variable:

```
37920 IF NV=TE THEN END : REM when there's no Next Variable, quit
37922 ADR=NV : NV=ADR + FN PK(ADR+2) : REM working pointer is ADR
37924 GOSUB 37980 : REM get name and type of variable at ADR
```

You'll find the number of dimensions in an array stored four bytes beyond the start of the array's entry. Since most arrays have five or fewer dimensions, and since our dump routine's complexity increases with each dimension we support, let's skip arrays with more than five dimensions, such as A(0,0,0,0,0,0):

```
37930 ND=PEEK(ADR+4) : REM get Number of Dimensions for this array
37932 IF ND>5 THEN PRINT "Array ";N$;" has ";ND;" dimensions." : GOTO 37920
```

Now, consider for a moment the line we'll use to actually print an array's value. Again, because I've cheated and looked ahead, I know it will go something like this:

```
37970 ::::: PRINT N$(1);N$(2);N$(3);N$(4);N$(5);") = ";XX$(A,B,C,D,E)
```

Of course, we'll have to overwrite the XX$ at the end of this line with the letters of the array we are actually working on, and we'll have to overwrite the (A,B,C,D,E) with the exact number of dimensions we have. For example, if we have a two-dimension array named PA%, after overwriting line 970 it should look like this:

```
37970 ::::: PRINT N$(1);N$(2);N$(3);N$(4);N$(5);") = ";PA%(A,B      )
```

That should be easy enough. Line 914 finds the addresses we need for poking changes into line 970, line 940 pokes in the variable's name, and lines 941-949 plug in either blanks or commas and letters:

```
37914 L=37970 : GOSUB 37990 : REM get ACL and ANL of line 37970

37940 FOR I=1 TO 3 : POKE ANL-16+I,ASC(MID$(N$,I,1)) : NEXT

37941 D1$=CHR$(32) : D2$=CHR$(32) : REM blank spaces
37942 IF ND>4 THEN D1$="," : D2$="E"
37943 POKE ANL-4,ASC(D1$) : POKE ANL-3,ASC(D2$)
37944 IF ND>3 THEN D1$="," : D2$="D"
37945 POKE ANL-6,ASC(D1$) : POKE ANL-5,ASC(D2$)
37946 IF ND>2 THEN D1$="," : D2$="C"
```

```
37947 POKE ANL-8,ASC(D1$) : POKE ANL-7,ASC(D2$)
37948 IF ND>1 THEN D1$=",": D2$="B"
37949 POKE ANL-10,ASC(D1$) : POKE ANL-9,ASC(D2$)
```

Now it's a simple matter of finding out how many elements each dimension has and building a loop that will step through each element. The number of elements in each dimension is stored beginning at byte 5 of the array description. The final dimension is stored first. And, believe it or not, these two-byte numbers are stored high-byte first—backwards from the backward method usually used. Let's collect the number of elements in each dimension in our own little array, D(    ):

```
37950 FOR I=1 TO ND
37952 D(I)=PEEK(ADR+3+(I*2))*256 + PEEK(ADR+4+(I*2))
37954 NEXT
```

The fact that we don't know how many dimensions the array has poses a few problems in building a loop to print the array elements, but IF statements can get around them:

```
37960 FOR I=1 TO 5 : N$(I)="" : NEXT : REM clear N$() array
37961 FOR A=0 TO D(ND)-1 : N$(1)=N$+"("+STR$(A)
37962 : IF ND>1 THEN FOR B=0 TO D(ND-1)-1 : N$(2)=","+STR$(B)
37963 :: IF ND>2 THEN FOR C=0 TO D(ND-2)-1 : N$(3)=","+STR$(C)
37964 ::: IF ND>3 THEN FOR D=0 TO D(ND-3)-1 : N$(4)=","+STR$(D)
37965 :::: IF ND>4 THEN FOR E=0 TO D(ND-4)-1 : N$(5)=","+STR$(E)
37970 ::::: PRINT N$(1);N$(2);N$(3);N$(4);N$(5);") = ";XX$(A,B,C,D,E)
37975 :::: IF ND>4 THEN NEXT
37976 ::: IF ND>3 THEN NEXT
37977 :: IF ND>2 THEN NEXT
37978 : IF ND>1 THEN NEXT
37979 NEXT : GOTO 37920: REM do next entry in array table
```

So much for dumping arrays.

The reason this program has numbers in the 37700-37999 range is that if you use it to debug your own programs you'll want it stowed away where it won't interfere with the program you're working on. The easiest way to proceed is to type VAR.READER into a word processor (the complete listing follows this article) and then save it in a text file. When you need it to help debug a program you are working on, EXEC it in while the program you are working on is loaded. Then RUN your program until it crashes (or install a STOP command to break it where you think you're having value trouble) and enter GOTO 37700. (Don't RUN 37700; that will clear all your variables.) VAR.READER will then give you a list of all your active variables and their values.

If you want to read a ProDOS VAR file, EXEC in VAR.READER, then enter RESTORE *pathname*. This will load the VAR file into memory. Finally, enter GOTO 37700 and the file will be listed. If you'd like a printed copy, just enter PR#1 or whatever first. If you want to dump the listing to a file, put OPEN and WRITE commands in lines 37701-37702 and add a CLOSE command before the END in line 37920.

To test the program, I recommend you take a look at the VAR files on Apple's ProDOS System Utilities disks. They are immense. Among other things, they contain all the menus that appear in the system utilities program. The advantage of using VAR files like this is that if you want to translate a program into another (human) language, all you have to change is the VAR file. The program itself can remain untouched.

Incidentally, creating a VAR file is simple. Use a word processor to create a list of variables and values that looks just like the one VAR.READER prints. At the top put the command NEW. At the bottom put the command STORE *pathname*. Save all this in a text file, then EXEC it into memory. The EXEC will create a new VAR file with all the variables and values in your list.

VAR.READER has one significant problem. It damages the values in variables that have the same name as the variables it itself uses. This is a direct consequence of its being written in Applesoft rather than assembly language. Assembly language would also be faster and could be much more transparent. On the other hand, fewer *Open-Apple* readers would be able to follow the logic of the program, it would become too long to print here, and it would take too long to write.

We can get around the problem by inserting a section at the beginning of the program that looks for any pre-existing variables that have the same names as the ones we'll be using. If we find any, we can print out their values before we tread all over them. Two long lines of dashes can be used to separate these from the other variables—if nothing appears between the lines you know that there are no variable-name conflicts. I've included lines 37710 through 37789 in the final program listing to take care of this problem. If you are faced with typing in the program, you could leave these lines out and cut your typing time in half if you are prepared to deal with the consequences.

If you type in VAR.READER and RUN it (not the normal operating procedure), here's what you'll see:

```
----------------------------------------
----------------------------------------
Function PK defined.
AD  = 5270
I   = 4
TS  = 5263
TE  = 5297
D  (0) = 0
D  (1) = 11
D  (2) = 0

   (27 more lines of array variables follow)
```

Note that, since we started the program with RUN, these are all variables used by VAR.READER itself. Only those simple variables activated through line 37812 show up in the display—line 37812 is where the end of the simple variable table is pinned down. Simple variables activated after that aren't listed.

Unfortunately, lines 37740 through 37762 activate all the program's array variables—they show up every time. When examining variables, don't get VAR.READER's mixed up with your own.

Just for fun, enter enter GOTO 37700 immediately after RUNning VAR.READER. Now all of VAR.READER's variables show up between the two dashed lines—the values shown with them are left over from the RUN. When these same variables appear *after* the second dashed line, the values shown are those of the current trod.

Here's the complete listing:

```
      *-------------------------------
      *     : VAR.READER
      *     :
      *     : by Tom Weishaar
      *     :     June 1986
      *     :
      *     : a public domain program
      *-------------------------------
37700 DEF FN PK(ADR)=PEEK(ADR)+PEEK(ADR+1)*256

37705 REM Lines 37710 to 37789 optional, see text

37710 PRINT "----------------------------------------"
37711 IF A >0 THEN PRINT "A   = ";A
37712 IF B >0 THEN PRINT "B   = ";B
37713 IF C >0 THEN PRINT "C   = ";C
37714 IF D >0 THEN PRINT "D   = ";D
37715 IF E >0 THEN PRINT "E   = ";E
37716 IF I >0 THEN PRINT "I   = ";I
37717 IF L >0 THEN PRINT "L   = ";L
37718 IF T >0 THEN PRINT "T   = ";T

37720 IF C1>0 THEN PRINT "C1  = ";C1
37722 IF C2>0 THEN PRINT "C2  = ";C2
37722 IF ND>0 THEN PRINT "ND  = ";ND
37723 IF NV>0 THEN PRINT "NV  = ";NV
37724 IF TE>0 THEN PRINT "TE  = ";TE
37725 IF TS>0 THEN PRINT "TS  = ";TS
37726 IF AC>0 THEN PRINT "AC  = ";AC
37727 IF AD>0 THEN PRINT "AD  = ";AD
37728 IF AN>0 THEN PRINT "AN  = ";AN

37730 IF LEN(D1$)>0 THEN PRINT "D1$ = ";D1$
37731 IF LEN(D2$)>0 THEN PRINT "D2$ = ";D2$
37732 IF LEN(N$) >0 THEN PRINT "N$  = ";N$

37740 FOR I=1 TO 5
37741 : IF D(I) > 0 THEN PRINT "D(";I;") = ";D(I)
37742 NEXT

37750 FOR I=1 TO 5
37751 : IF LEN(N$(I)) > 0 THEN PRINT "N$(";I;") = ";N$(I)
37752 NEXT

37760 FOR I=0 TO 3
37761 : IF LEN(TY$(I)) > 0 THEN PRINT "TY$(";I;") = ";TY$(I)
37762 NEXT

37789 PRINT "----------------------------------------"

37790 TYPE$(0)=" " : REM real
37791 TYPE$(1)=""  : REM function
```

## Ask (or tell) Uncle DOS

### UniDisk 3.5 * 3 + CP/M

I have an Apple IIc with an external disk IIc and a UniDisk 3.5. I'm considering buying a second UniDisk and I wondered if I could use all three of my extra drives at once.

Do you know of any way to use the UniDisk 3.5 with CP/M? I purchased Nordic Software's *ProFix* (December 1985, page 93), and it makes the 3.5 inch drives work terrifically with DOS 3.3. Is anything like that available for CP/M?

Laird Malamed
Los Angeles, Calif.

*On an Apple IIc, up to two UniDisk 3.5s and a single 5-1/4 inch drive can be daisy-chained together. The 5-1/4 inch drive must be at the end of the chain. The UniDisk 3.5s act as if they are connected to slot 5, the 5-1/4 inch drive appears to be in slot 6, drive 2.*

*Unfortunately, this doesn't work on the IIe. The problem is that, unlike the case of the IIc, there could already be a 5-1/4 inch drive actually connected to slot 6, drive 2. Consequently, the UniDisk 3.5 controller card doesn't allow 5-1/4 inch drives to be in the daisy chain.*

*Applied Engineering is reportedly working on a UniDisk 3.5 driver for the CP/M work-alike it provides with its Z-80 products. I don't know of any other work in progress along these lines—perhaps our readers can help.*

### UniDisk 3.5 eject

Is there a software command to cause a UniDisk 3.5 drive to eject the disk? I have a UniDisk 3.5 and am very happy with it. I think one advantage it has over hard disks is that it is simpler both electronically and mechanically and therefore less vulnerable to breakdowns. It's also LOTS quieter.

Stephen Bach
Scottsville, Va.

*Yes, there is a "Smartport" or "protocol converter" call for ejecting a 3.5 inch disk. Some month soon we're going to look at the protocol converter in detail, but the eject-a-disk question keeps coming up so I'll show you how to do that right now:*

```
100 REM ** Eject 3.5 inch UniDisk **
110 SLOT=5 : DRIVE=1
120 C$="300:20 0D C5 04 0A 03 8D 11 03" : GOSUB 500
121 C$="309:60 03 01 0F 03 04 00 00 00" : GOSUB 500
130 POKE 770, SLOT+192 : REM Cslot to $302
131 POKE 779, DRIVE    : REM drive to $30B
140 CALL 768
150 ERR=PEEK(785)
160 IF ERR=0 THEN END
161 IF ERR=39 THEN PRINT "Eject failed." : END
162 IF ERR=40 THEN PRINT "No device cncted." : END
163 PRINT "Error in eject listing." : END

500 REM Lam technique, space before and after N
501 C$=C$ + " N D9C6G"
510 FOR I=1 TO LEN(C$)
512 : POKE 511+I, ASC(MID$(C$,I,1))+128
514 NEXT
520 POKE 72,4 : CALL -144
530 RETURN
```

*The above program is for demonstration purposes mostly; as written it's pretty slow. Those of you who want more speed would probably like to see the assembly language code embedded in the program —and here it is:*

```
              EJECT
300:20 0D C5  JSR DISPATCH    P Converter entry
303:04         .DA #4         command 4, 'control'
304:0A 03      .DA CMDLIST    adr of command list
306:8D 11 03  STA ERR         store error code
309:60        RTS               (zero=no error)

              CMDLIST
30A:03         .DA #3         # of items in cmdlist
30B:01         .DA #1         use 2 for drive 2
30C:0F 03      .DA CTRLLIST   adr of control list
30E:04         .DA #4         cmd 4, eject disk

              CRTLLIST
30F:00 00      .DA 0000       # of CTRLLIST bytes

              ERR
311:00         .DA #00
```

*The "dispatch" address on both the IIe UniDisk 3.5 card and the IIc 3.5 ROMS is $Cs0D (s=slot the card is in, which is 5 on the IIc), however, you are really supposed to find it by grabbing the value in $CsFF, adding three to it, and using it as an index from $Cs00. Without adding three, incidentally, you'll get the card's ProDOS entry point.*

*If an error occurs, the carry will be set and the error number will be in the A register. If there is no error, the carry will be clear and A will hold zero. The important error codes to watch for are $27, FAILURE TO EJECT, and $28, NO DEVICE CONNECTED. Most other possible errors are related to invalid command tables. Apple's documentation doesn't say what could cause a FAILURE TO EJECT error or what to do about it if it happens.*

*The "device number" in the command list could theoretically be something other than one for Drive*

```
37792 TYPE$(2)="$" : REM string
37793 TYPE$(3)="%" : REM integer

37810 REM * Display names and values of all simple variables *
37811 TS=FN PK(105)   : REM TS = simple variable table's starting adr
37812 TE=FN PK(107)-1 : REM TE = table's ending adr
37813 L=37824 : GOSUB 37990 : REM put address of line L in ACL and ANL

37820 FOR ADR=TS TO TE STEP 7 : REM scan 7-byte simple variables
37821 : GOSUB 37980 : REM get name and type of this variable
37822 : IF T=1 THEN PRINT "Function ";N$;" defined." : NEXT
37823 : FOR I=1 TO 3 : POKE ANL-5+I,ASC(MID$(N$,I,1)) : NEXT
37824 : PRINT N$;" = ";XX$
37825 NEXT

37910 REM * Display names and values of all array variables *
37911 A=0 : B=0 : C=0 : D=0 : E=0 : ND=0 : NV=0 : D1$="" : D2$=""
37912 NV=FN PK(107) : REM array table's start is Next Variable
37913 TE=FN PK(109) : REM table's ending address
37914 L=37970 : GOSUB 37990 : REM get ACL and ANL of line 37970

37920 IF NV=TE THEN END : REM when there's no Next Variable, quit
37922 ADR=NV : NV=ADR + FN PK(ADR+2) : REM working pointer is ADR
37924 GOSUB 37980 : REM get name and type of variable at ADR

37930 ND=PEEK(ADR+4) : REM get Number of Dimensions for this array
37932 IF ND>5 THEN PRINT "Array ";N$;" has ";ND;" dimensions." : GOTO 37920

37940 FOR I=1 TO 3 : POKE ANL-16+I,ASC(MID$(N$,I,1)) : NEXT

37941 D1$=CHR$(32) : D2$=CHR$(32) : REM blank spaces
37942 IF ND>4 THEN D1$="," : D2$="E"
37943 POKE ANL-4,ASC(D1$) : POKE ANL-3,ASC(D2$)
37944 IF ND>3 THEN D1$="," : D2$="D"
37945 POKE ANL-6,ASC(D1$) : POKE ANL-5,ASC(D2$)
37946 IF ND>2 THEN D1$="," : D2$="C"
37947 POKE ANL-8,ASC(D1$) : POKE ANL-7,ASC(D2$)
37948 IF ND>1 THEN D1$="," : D2$="B"
37949 POKE ANL-10,ASC(D1$) : POKE ANL-9,ASC(D2$)
37950 FOR I=1 TO ND
37952 D(I)=PEEK(ADR+3+(I*2))*256 + PEEK(ADR+4+(I*2))
37954 NEXT

37960 FOR I=1 TO 5 : N$(I)="" : NEXT : REM clear N$() array
37961 FOR A=0 TO D(ND)-1 : N$(1)=N$+"("+STR$(A)
37962 : IF ND>1 THEN FOR B=0 TO D(ND-1)-1 : N$(2)=","+STR$(B)
37963 :: IF ND>2 THEN FOR C=0 TO D(ND-2)-1 : N$(3)=","+STR$(C)
37964 ::: IF ND>3 THEN FOR D=0 TO D(ND-3)-1 : N$(4)=","+STR$(D)
37965 :::: IF ND>4 THEN FOR E=0 TO D(ND-4)-1 : N$(5)=","+STR$(E)
37970 ::::: PRINT N$(1);N$(2);N$(3);N$(4);N$(5);") = ";XX$(A,B,C,D,E)
37975 :::: IF ND>4 THEN NEXT
37976 ::: IF ND>3 THEN NEXT
37977 :: IF ND>2 THEN NEXT
37978 : IF ND>1 THEN NEXT
37979 NEXT : GOTO 37920: REM do next entry in array table

37980 REM * Read variable name and type  *
37981 REM *    T=variable type           *
37982 REM *    N$=variable name          *
37983 REM
37984 T=0
37985 C1=PEEK(ADR)   : IF C1>127 THEN T=T+1 : C1=C1-128
37986 C2=PEEK(ADR+1) : IF C2>127 THEN T=T+2 : C2=C2-128
37987 IF C2=0 THEN C2=ASC(" ")
37988 N$=CHR$(C1)+CHR$(C2)+TYPE$(T)
37989 RETURN

37990 REM * Find line number L           *
37991 REM *   ACL=address of current line  *
37992 REM *   ANL=address of next line     *
37993 REM
37994 ACL=FN PK(103) : REM address of current line, 103=TXTTAB
37995 ANL=FN PK(ACL) : REM address of next line
37996 IF ANL=0 THEN PRINT "There's no line ";L;" in this program." : STOP
37997 IF L=FN PK(ACL+2) THEN RETURN
37998 ACL=ANL : GOTO 37995
```

*1 and two for Drive 2 if other protocol converter devices were daisy-chained in between the computer and the drives. Determining the actual device number of each drive from within a program is fairly complicated, however. Yet the ProDOS entry point on the UniDisk 3.5 card uses slot and drive parameters rather than a device number; the card figures it all out somehow. Doesn't it seem like ProDOS itself should support an eject-disk command? At the moment, however, all the ProDOS kernel can tell a device to do is to read, write, report its status, and format itself.*

## ASCII Express V4.20

I have version 4.20 of *ASCII Express*. I am using an enhanced IIe and I chose the "Apple IIe" selection for a "local console" when installing the program. It works at 1200 baud with no problems, contrary to your advice in April (page 2.23). Maybe the directive to choose "Auto" or "Pascal 1.1" applies to earlier versions.

James W. Patton
Littleton, Colo.

*The version of **ASCII Express** I use around here also says it's 4.20, but I swear I had to abandon the IIe local console option to get it to work on an enhanced IIe. From your information, it would appear the publisher has altered the program to support both old and new IIes without changing the version number.*

*One would think that software companies would use a new version number when they alter software. What's the sense of having version ID numbers in the first place if you can't tell different versions apart with them? Still, one has to applaud the fact that at least the publisher of **ASCII Express** appears to be trying to keep the program up-to-date.*

*Dennis Doms, who wrote the answer to the April letter, says to mention that there's one anomaly he forgot to mention that appears even after you follow the setup he recommended. It's that the cursor tends to leap around the screen during protocol transfers (this may also have been corrected in your version). Dennis says this is disconcerting, but doesn't seem to affect the file transfer.*

*Finally, on the "undocumented feature" front: **Open-Apple** subscriber John Morse reports that **ASCII Express** can operate at a maximum of 9600 baud with a Super Serial card—not 4800 baud. Just select "8" from the "baud rate" menu. Although 8 doesn't actually appear as a selection on the menu, AE will accept it and will proceed to run quite happily at 9600 baud. This seems to work on both the DOS and ProDOS versions.*

## ProDOS compilers

Any more news on ProDOS compilers?

Robert C. Heidreth
Rochester, N.Y.

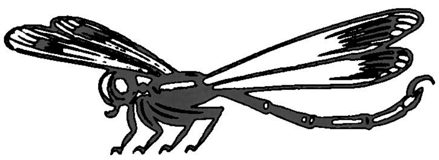
*While calling around in search of a ProDOS-based Applesoft compiler, we had an interesting conversation with Tom Burns at Roger Wagner Publishing, publishers of the **SpeedStar** DOS 3.3-based Applesoft compiler. Tom said that, at this time, he knew of no commercially available ProDOS-based compiler for Applesoft. He also told us Roger Wagner Publishing had no plans to produce one, and gave us their reasons.*

*1) ProDOS is so much faster than DOS 3.3 in disk operations that use of ProDOS and BASIC.SYSTEM, in itself, can speed up programs significantly. Therefore, there is less incentive for the use of a compiler.*

*2) Compilers, at the time they were introduced, were the easiest way of speeding up Applesoft programs. Programming sophistication has now produced machine language tools, which can be accessed from Applesoft programs, that provide much of the speed of compilers. In addition, these "ampersand utilites" and "ProDOS added commands" use much less memory than compiled code (DOS 3.3 Applesoft compilers greatly expand programs as they are compiled), are faster, and do not hamper development (the Applesoft program can still be debugged interactively, whereas use of a compiler necessitates re-compilation after every change).*

*3) DOS 3.3 Applesoft compilers were not big sellers. Given the additional reasons for not using a compiler with ProDOS, a ProDOS compiler would probably be a money-loser rather than money-maker.*

*The only remaining feature a compiler provides is protection against having the program LISTed and modified by the user. Of course, half the world thinks LIST protection is a bad idea, anyhow. At the moment it appears the other half will have to come up with some protection scheme other than compilation in the ProDOS world.*



## Black tape

Why are the ventilation slots on the Disk II covered (from the inside) by black tape?

Harvey S. Picker
Hartford, Conn.

*My guess is that The Disk II doesn't generate enough heat to need ventilation. Apple probably put the slots there to make the appearance of the drives match the Apple II and then added the tape to keep dust and dirt out.*

## Cheap color

I suppose everybody else in the world already knows this, but on the off-chance that they don't: You can use your color TV as a color monitor for the Apple if you have a video cassette recorder. Simply run a standard cable from the monitor jack on the Apple to the video-in jack on your Beta or VHS.

I use a two-way splitter from Radio Shack so that I can use both the monochrome monitor to read the text and the TV to see the pretty colors. (I use this double-combination for playing adventure games with graphics, mostly.)

Douglas Cuff
St John's, Newfoundland

*I didn't know that. I have heard people complain in the past that they couldn't record Apple colors on a video cassette with this trick, so I assumed it wouldn't work. Sounds like it does work, however, on at least some machines.*

*People whose goal is to get full-color Apple screen displays on video tape and who find this trick doesn't work should try running the Apple's video signal through an RF modulator (the standard way of getting Apple video to a TV) and then to the **antenna** inputs on the VCR. Who knows, it might work.*

## INPUT under EXEC

Here's a couple of tricks to liven up your readers' DOS 3.3 EXEC routines. Type this as a three-line text file, and then EXEC it:

```
POKE 118,0 : POKE 43699,0 : INPUT "ENTER
                        FILE NAME: ";F$ : POKE 43699,1
MONICO
POKE 118,0 : POKE 51,0 : PRINT CHR$(4);"OPEN";F$
```

The first line puts EXEC in deferred mode so that keyboard input can be requested by an EXEC routine (this trick was desgribed by Bob Schmidt in *Washington Apple Pi*, March 1985).

The last line persuades DOS that a BASIC program is running, so it will accept commands that normally cannot be executed in immediate mode. These are the same POKEs that are part of your instructions for using DOS from assembly language in *The DOSTalk Scrapbook* (pages 129-130).

I'm not sure that either of these is of earth-shaking importance, but it's nice to know you can do them if you want to.

On the subject of memory location 51, which contains the prompt character for keyboard input—in immediate mode, it contains 221, the right bracket prompt for Applesoft. In a Applesoft program, an INPUT statement changes it to 128 (null). Both of these make sense, but you'll get six (why six?), if you enter the following:

```
NEW
10 PRINT PEEK (51)
RUN
```

Paul Nix
Summit, N.J.

*Six is what ends up in the Y register when DOS 3.3 searches its command table for the word RUN—Y is used as an index for other table-lookup operations. DOS stores this six into the PROMPT location so that the location will later reveal that a program is executing. When the program ends and falls into immediate mode, Applesoft will change PROMPT to "]" (93, $5D) and Integer Basic will change it to ">" (62, $3E).*

*Why six? It just happened to be handy. A more logical value would require more code but serve no purpose other than aesthetics. If you run your test program under ProDOS you'll find a zero there.*

## Address sorting

I operate a small business in which I reference all of my customers by their street address. I write all of my software in Applesoft and have begun to expand the versatility of my programs but have a question about how to sort strings that contain both numbers and characters, such as 123 Main Street or 456 Main Street, and so on.

There are many sorting routines (both machine language and BASIC) that I can use for the actual sorting but my efforts to manipulate the street address string to sort the street name alphabetically and the house numbers in ascending order has not been successful.

Would you please discuss the most efficient way to accomplish this?

George Rolla
Redwood City, Calif.

*The basic problem here seems to be that you want to end up with your customers sorted by street in ascending house order. This would be an excellent sort for making deliveries, for example.*

*The only easy way Dennis and I can figure out to do this is to keep the house numbers and the streets in different strings. Combine the strings when you want to print someone's address with something like PRINT N(I); " "; S$(I).*

*While this may sound like a lot of trouble, trying to implement a sort routine that can recognize and compensate for numerical values in a string that is being sorted alphabetically (and vice versa) is going to be a real job. Assuming you already have the addresses in a string array, splitting them into a numerical array and a string array shouldn't be too complicated.*

*Once you have the numbers and names separated, you can use just about any sorting technique to put your database in house number order, then do a "bubble" sort to order the street names. Dennis says to use a bubble sort (normally disdained for its sloth) for the final sort because most faster sorting algorithms, such as the shell sort, will scramble the first sort while performing the second.*

## Double-res files

Recently I attempted to move some double-high resolution pictures from DOS 3.3 to ProDOS using CONVERT. The pictures came on the disk Apple provided with its auxiliary-slot RGB color card. The DOS 3.3 files were binary and 65 sectors long. The process seemed simple but the transfer attempt resulted only in a hung system with a bunch of trash on the screen and a few clicks of the IIe speaker just before that. As usual, Apple's no help.

I am also seeking the locations to poke, or an & routine, that would allow multi-color text on my RGB screen when using ProDOS. The only thing I have is for DOS 3.3 and it doesn't work with ProDOS.

Steve Perry
Santa Ana, Calif.

*Dennis tried CONVERT on Apple's RGB disk and experienced the same problem, then tracked down the cause. He says if he worked for Apple, he would be embarrassed to acknowledge the problem, too.*

*Whoever wrote the DOS 3.3 double-high resolution demos used a non-standard file format for the binary files holding the pictures. Specifically, the four bytes at the beginning of the first sector of the files, which are supposed to hold a binary file's loading address and length ("A" and "L") values, have been omitted. The files contain only the data for the picture. The programs on the disk use &LOAD and &SAVE commands to access these non-standard files.*

*There is nothing in the DOS catalog to indicate to CONVERT that the file is abnormal. (The programmer could have given these files one of the normally unused DOS 3.3 file types, for example.) Therefore, CONVERT tries to transfer the file using the information in the file. The file PIE.CHART appears to have a loading address of $8080 and a length of $8080 bytes. CONVERT gags on the file after copying 45 blocks.*

*CONVERT ends up by flipping through all the softswitches in page $C000 (hence the odd video and speaker effects you noted). The fact that CONVERT crashs like this rather than reporting an error does not give much reassurance about CONVERT's performance.*

**Copy II Plus 6.0** *can make the conversion without bombing; however, the starting address and file length are still taken from the erroneous data within the file—just as you would expect. The first four bytes of the converted picture are missing and all other bytes are shifted four positions forward. Not a pretty result.*

*Apple's programmer saved one sector of disk space per file by using the special storage technique. There are five of these files on the disk, a net saving of five sectors (the disk has twelve free sectors left). The same effect could have been achieved by leaving off the last four bytes of the file, which don't appear on the high-resolution screen, anyhow. The price of this misdirected file-size efficency is the inability to convert the files back and forth between DOS 3.3 and ProDOS. And this is supposed to be Apple's best example of how to program double-resolution graphics?*

*As far as I can determine, Apple has never officially defined how double-resolution files should be stored under DOS 3.3; all we have is this rather poor example. Beagle Bros spurned it and developed a format whereby double-res graphics are stored in two files—the main-memory portion in a binary file named, for example, PICTURE; the aux-memory portion in a file named PICTURE.AUX. Such files are readily transportable between DOS 3.3 and ProDOS.*

*While Apple has been no help with DOS 3.3, ProDOS technical note #13 defines a standard for ProDOS double-high-resolution graphics files. This standard uses a special file type called FOTOFILE (type $08), which was originally defined for the Apple III. FOTOFILEs have the aux-memory portion of the image in the first $2000 bytes of the file and the main-memory portion in the final $2000 bytes. In addition, byte $78 of the file holds a code that tells what kind of file it is. Here's a list of the codes:*

```
FOTOFILE graphics mode codes
  (found at file.start+120 or $78)
                                     page 1    page 2
280 x 192 std high-resolution          0         4
280 x 192 ''limited color''            1         5
560 x 192 dbl-high black/white         2         6
140 x 192 dbl-high 16-color            3         7
```

*FOTOFILEs make a poor choice for a standard, however. On the Apple II, there is no such thing as a "page 2" double-high-resolution picture, so codes 5, 6, and 7 apply to the Apple III only. The same goes for type 1, 280 x 192 "limited color," which never appears in Apple II documentation.*

*On the other hand, Apple II RGB supports a "mixed 560/140" mode, for which no FOTOFILE code has been defined. This mode is available with RGB equipment only.*

*Another problem with the FOTOFILE standard is that the graphics mode code is stored within the aux-memory portion of the file for double-resolution graphics, but in the main-memory portion for standard high-resolution files. If the main-memory portion of the graphic was at the beginning of the file, with the graphic-mode code embedded within it, you could load that portion directly into memory, then look at the code to see if there was an additional portion to go into aux-memory. Here's a simple example of how such a file would be loaded under ProDOS, using a file type I just made up and called PIC:*

```
1000 POKE 49153,0 : REM 80STORE on
1005 POKE 49239,0 : REM HIRES on
1010 POKE 49236,0 : REM PAGE2 off
1020 PRINT D$;"BLOAD pathname, A8192, L8192, TPIC"
1030 T=PEEK(8192+120) : REM get mode code
1040 IF T=(std res graphic code) THEN 1080
1050 POKE 49237,0 : REM PAGE2 on
1060 PRINT D$;"BLOAD pathname, A8192, L8192, B8192,
        TPIC"
1070 POKE 49236,0 : REM PAGE2 off
1080 ON T GOTO .... : REM flip RGB switches for mode
```

*Using FOTOFILEs, you either have to load the first part of the file in main-memory and then move or reload it into aux-memory if the picture turns out to be double-resolution, or you have to load the first part into aux-memory and move it to main if it turns out to be single-resolution. Come on, Apple—how about a new file type just for Apple II graphic files?*

*Apple's treatment of double-high resolution, in terms of documentation and support, has been shoddy since day 1. (One of many examples—memory maps in the Apple RGB manual all show a high-res graphics page 2 in auxiliary memory, even though there isn't such a thing.) There is no more obscure arena in the Apple II world than double-resolution. Apple has indicated that advanced color graphics will be a part of the next Apple II—let's hope they get their act together better this time or programmers will be too confused to use them.*

*Oh, and speaking of confusion, does your question about multi-colored text refer to the green-amber-blue-white choice for text screens; to 40-column, 16-color foreground, 16-color background text; or to colored text on the double-high-resolution screen? Actually it makes little difference—the switches for all these things are the same whether you're using ProDOS or DOS 3.3. For everything I know about RGB see the May 1985* **Open-Apple***, pages 35 and 36, and the July 1985 issue, page 54.*

**Open-Apple**

is written, edited, published, and

**© Copyright 1986 by Tom Weishaar**

| | |
|---|---|
| **Business Consultant** | **Richard Barger** |
| **Technical Consultant** | **Dennis Doms** |
| **Circulation Manager** | **Sally Tally** |

Most rights reserved. All programs published in **Open-Apple** are public domain and may be copied and distributed without charge (most are available in the MAUG library on CompuServe). Apple user groups and significant others may obtain permission to reprint articles from time to time by specific written request. Requests and other editorial material, including letters to Uncle DOS, should be sent to:

**Open-Apple**
**P.O. Box 7651**
**Overland Park, Kansas 66207 U.S.A.**

ISSN 0885-4017. Published monthly since January 1985. World-wide prices (in U.S. dollars; airmail delivery included at no additional charge): $24 for 1 year; $44 for 2 years; $60 for 3 years. All back issues are currently available for $2 each; a bound, indexed edition of Volume 1 is $14.95. Index mailed with the February issue. Please send all subscription-related correspondence to:

**Open-Apple**
**P.O. Box 6331**
**Syracuse, N.Y. 13217 U.S.A.**

Subscribers in Australia and New Zealand should send subscription correspondence to **Open-Apple**, c/o Cybernetic Research Ltd, 576 Malvern Road, Prahran, Vic. 3181, AUSTRALIA.

**Open-Apple** is available on disk for speech synthesizer users from Speech Enterprises, P.O. Box 7986, Houston, Texas 77270 (713-461-1666).

Unlike most commerical software, **Open-Apple** is sold in an unprotected format for your convenience. You are encourgaged to make back-up archival copies or easy-to-read enlarged copies for your own use without charge. You may also copy **Open-Apple** for distribution to others. The distribution fee is 15 cents per page per copy distributed.

**WARRANTY AND LIMITATION OF LIABILITY.** I warrant that most of the information in **Open-Apple** is useful and correct, although drivel and mistakes are included from time to time, usually unintentionally. Unsatisified subscribers may return issues within 180 days of delivery for a full refund. Please include a note from your parents or children confirming that all archival copies have been destroyed. The unfullilled portion of any paid subscription will be refunded on request. MY LIABILTY FOR ERRORS AND OMISSIONS IS LIMITED TO THIS PUBLICATION'S PURCHASE PRICE. In no case shall I or my contributors be liable for any incidental or consequential damages, nor for any damages in excess of the fees paid by a subscriber.

**Open-Apple** is neither affiliated with nor responsible for the debts of Apple Computer, Inc.; "tinaja questing" is a trademark of Don Lancaster.

**Source Mail: TCF238** **CompuServe: 70120,202**