Open-Apple™

April 1986
Vol. 2, No. 3

ISSN 0885-4017
newstand price: $2.00
per page photocopy charge: $0.25

Releasing the power to everyone.

Applesoft meets the RAMdisk

While the price of RAM chips has been dropping the last few years and Apple IIs using memory expansion cards have become common, the little bit of RAM that can be used by Applesoft has been getting more and more dear every year.

Applesoft's designers considered the Apple II to be a 48K machine, which was large at that time, and they gave Applesoft the ability to use every byte within that limit—but no more. As a result, no matter how many of today's RAM cards you plug into your computer, the original 48K bytes in main Apple II RAM are all that today's Applesoft is ever going to be able to take advantage of.

If you use cassette tape for program and data storage, the Applesoft memory area comprises a 47,103-byte range from $800 to $BFFF. In a cassette-tape environment this is a massive amount of memory. In fact, the memory is so massive and cassette tape is so cranky that it is well worth sacrificing as much memory as necessary to gain the DOS-given ability to save and load programs and data on disks instead.

The sacrifical RAM. So DOS came along and we gave it over 20 per cent of Applesoft's space—the 10,751 bytes from $9600 to $BFFF. In return, DOS gave us the ability to write longer programs and to store more data—and suddenly 48K wasn't so massive anymore. Users have been limited by the OUT OF MEMORY error ever since.

One answer to the problem, if you have a 64K or larger Apple, is to use a program that moves DOS 3.3 to "language card" memory. For example, the program DOS-UP on the Beagle Bros ***ProntoDOS*** disk does this. With such a set-up you can reclaim all but 256 bytes of the lower 48 for Applesoft. Compatibility problems often arise when DOS 3.3 is moved, however, and this trick doesn't work with ProDOS. The ProDOS kernel uses the whole language card area itself; Basic.system takes up all of DOS 3.3's memory space. And by the way, if your program needs to open files, Basic.system will demand an additional 1,024 bytes of Applesoft memory per open file. DOS 3.3 also does this in a smaller way—it needs an additional 595 bytes for each open file after the first three.

Another solution for the OUT OF MEMORY problem, if you have a 128K or larger Apple, is to use the EXTRA.VARIABLES program on the Beagle Bros **Extra K** disk. This program modifies Applesoft so that it stores all variables in auxiliary memory. Instead of the normal 36K that your programs and data must *share*, EXTRA.VARIABLES lets your programs have 35K and your variables 59K. That should be enough to solve tons of problems. (Note: of the 59K variable space, 16K of the 59K must be used for string storage. Only 43K can be used for *numeric* variables. That's enough for a floating-point array with 8,000 elements or an integer array with 22,000 elements.) However, *Extra K* doesn't help much if the bulk of your memory problem is a fat program rather than lots of variables. Programs that rely on *Extra K* won't run on older Apples that can't use the IIe/IIc auxiliary memory scheme. Other compatibility problems can arise.

Whether or not one of these solutions works for you, assembly language routines that add much-needed abilities and speed to Applesoft may also be encroaching on your program space. Just as DOS gave us vast storage improvements over cassette tape at the price of some memory, a huge library of assembly language routines has grown up around us that can give us Applesoft improvements at the price of just a few hundred bytes each. Basic.system even includes a standard way to obtain RAM for such routines—but the space comes out of Applesoft's allotment.

And don't forget that if you want to use a page of high-resolution graphics, you lose more than 20 per cent of the Applesoft memory that's left after DOS has seized what it needs. Two high-res pages will cost you 40 per cent. Either way, the memory remaining for Applesoft after HGR is divided into two small chunks rather than one large, contiguous area.

So while RAM *chips* have been getting cheaper, the demand for Applesoft's RAM has been increasing. The sudden appearance of large memory cards has been of no help to the Applesoft programmer. Their use has been limited to assembly language wizards, who by careful bank control, can take advantage of additional memory. But even the wizards find that the variety of cards and protocols, and the difficulty of bank switching and control, make using additional memory extremely difficult.

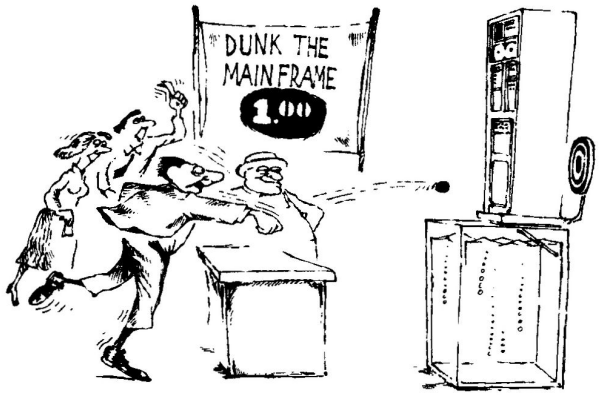
The RAMdisk alternative. The only easy way a programmer can use a big RAM card is as a RAMdisk. Let the RAM on the card appear to be a disk drive. Store segments of your program and data on this "disk" and let DOS load the segments as they are needed. Because DOS is used as an intermediary, speed will be lost compared to accessing the card directly, but it becomes DOS's responsiblity to deal with the variety of cards and protocols and with the bank switching and bank control.

A program that uses extra RAM as a RAMdisk, rather than accessing it directly, immediately becomes compatible with all the various RAM cards. Such a program doesn't require assembly language, but can be written in Applesoft. Such a program will run at speeds close to those obtained by using extra RAM directly. And such a program would even work, at slower speeds, with a hard disk or a floppy.

DOS can fulfill its responsiblity to deal with the variety of cards on the market fairly easily, because most cards include RAMdisk "drivers" for DOS 3.3 and ProDOS. A "driver" is an assembly language program that makes the RAM card look like a disk drive to DOS. Such drivers are built into the ROM on Apple's own slot-based memory cards. Auxiliary-slot cards and most standard-slot cards other than Apple's don't have the drivers in ROM. Instead, you are supposed to load them into what little RAM you have left.

The world awaits "universal" DOS 3.3 and ProDOS RAMdisk drivers that would recognize which, if any, of all the available RAM cards were installed in a computer and turn them into one large RAMdisk. A universal driver could be licensed to software developers for inclusion on their own disks. This would allow any commercial program to easily recognize and use any kind of RAM card.

While commercial developers and non-programmers wait for the universal RAMdisk driver, the rest of us can start with the drivers and cards we have now



and learn how to use them by writing Applesoft programs that are segmented. Both programs and data can be segmented and used a piece at a time. Basic.system even has built-in support for this with its CHAIN, STORE, and RESTORE commands. DOS 3.3 has a separate CHAIN routine for Applesoft (to say nothing of a CHAIN *command* that works only with Integer Basic). In addition, if you know what you are doing, you can use BLOAD and BSAVE to overlay portions of Applesoft programs and data.

The image of Applesoft. In order to understand how these commands and tricks work, it is important that you know what an Applesoft program and its variables look like in memory. Figure 1 is a slightly modified version of a memory map that originally appeared in the January 1985 ***Open-Apple*** (page 4). Compared to most Apple II memory maps you will see, this one is upside down. That's because when you examine memory bytes with the Monitor, the values appear on your screen with the lowest addresses at the top—just like in this map. Figure 1 shows how the main 64K bank of Apple memory was allocated in the Apple II-Plus. All later models look exactly like this when they are powered up.

The section from byte zero to byte $BFFF is RAM. Beyond that is ROM. The memory area for Applesoft programs and data, as shown, extends from byte $800 to $9600. Bytes zero through $7FF are used for the text screen, keyboard input buffer, stack, zero page, and other special stuff. Bytes $9600 to $BFFF are used by DOS. $C000 though $CFFF are used for softswitches and for addressing cards in slots. $D000 though $F7FF contain the Applesoft interpreter. This is the machine language program that contains the genes that make Applesoft come alive in your computer. Bytes $F800 through $FFFF hold the Monitor.

The memory area for Applesoft programs and data is broken into five sections. First there is the image of the program itself. Immediately after the program image is a table of simple (non-array) variables, followed by a table of array variables. After that comes the free memory that is left. The final section is a storage space for strings.

As shown in figure one, Applesoft keeps track of where each of these sections begins and ends by means of two-byte pointers stored in page zero. DOS uses the TXTTAB and PRGEND variables to determine where a program is when you execute a SAVE. Likewise, DOS places a program at TXTTAB and updates PRGEND when you do a LOAD or RUN.

You can create up to two additional sections within memory, for your own purposes, using the commands LOMEM: and HIMEM:. LOMEM: controls where the variable tables will begin by changing the value in VARTAB. HIMEM: controls where the string storage area will be by changing the value in MEMSIZ. By manipulating these and TXTTAB, you can create as many as three spaces for assembly language programs, for buffers, or for the high resolution graphic screens, as shown in Figure 1.

Using LOMEM: the space will appear between the end of your program and the beginning of your variable tables. Using HIMEM: and DOS 3.3, the space will appear between the string storage space and DOS. Using HIMEM: and Basic.system, which is not recommended, the space will start 1,024 bytes beyond your new HIMEM: value and extend that same distance beyond the previous HIMEM: value.

In any case, you must use these commands before you have used *any* variables in your program or you will screw things up. In addition, if you mistakenly try to set LOMEM: lower than its current level, which would usually cause your variable tables to overwrite your program, Applesoft will stop you with an OUT OF MEMORY error. On the other hand, if you set HIMEM: higher than normal, no error will stop you, and your strings will overwrite DOS. You must have a clear idea of what you are moving where in memory when you use these commands.

The program image. The image of your program itself is usually stored beginning at byte $800, although with a few tricks we'll look into in a moment you can make it start anywhere. In the October ***Open-Apple*** we looked at Applesoft program structure briefly (page 75), now let's look at it in more detail.

As you type in a program line such as 10 TEXT : HOME, the Applesoft interpreter scans it. First it looks for a line number. If it finds one, it assumes you are entering a line that is part of a larger program and it stores the hex equivalent of the line number in memory and continues scanning what you have entered. If it can't find a line number, on the other hand, Applesoft assumes you are entering a line that you want to execute immediately, and it will do so as soon as you press return.

Since immediate-execution commands aren't stored in memory, our only concern here is with entries that begin with line numbers. After the line number is found, Applesoft continues scanning the entry, looking for words that it recognizes. The words that it knows, which are called "reserved words," are converted into *tokens* and stored in memory. A token is simply a number between 128 and 234 that represents one of the commands or functions known to Applesoft. All other characters on the line are stored in memory as low-value ASCII characters. Because low-value ASCII is used, the ASCII characters all have their high bits clear, while the tokens, being numbers higher than 128, all have their high bits set.

A complete list of Applesoft's reserved words and their associated tokens is too long to print here. However, the following program will dig the reserved word table out of the Applesoft interpreter and show it to you. The table is embedded between $D0D0 and $D25E. It's in low-value ASCII, except for the final character of each reserved word, which is in high-value ASCII. The reserved words are stored in token-number order.

```
10 REM *** APPLESOFT RESERVED WORD AND TOKEN LISTER ***

100 TKN = 128
110 FOR ADR = 53456 TO 53854
120 : V = PEEK(ADR)
130 : PRINT CHR$(V);
140 : IF V > 127 THEN PRINT TAB(10);TKN : TKN = TKN + 1
150 NEXT
```

As stored in memory, each line of an Applesoft program begins with a byte containing a zero. The next two bytes point to the next program line. The next two bytes hold the line number in hex. After that come the tokens and ASCII characters. For example, if you enter the Monitor and examine memory after entering our little token peeker, here's what you'll see:

ZEROS

```
800- 00 3B 08 0A 00 B2 20 2A
808- 2A 2A 20 41 50 50 4C 45
810- 53 4F 46 54 20 52 45 53
818- 45 52 56 45 44 20 57 4F
820- 52 44 20 41 4E 44 20 54
828- 4F 4B 45 4E 20 4C 49 53
830- 54 45 52 20 2A 2A 2A 00
838- 44 08 64 00 54 4B 4E D0
840- 31 32 3B 00 59 0B 6E 00
848- B1 41 44 52 D0 35 33 34
850- 35 36 C1 35 33 3B 35 34
858- 00
```

ASCII EQUIVALENTS

```
800- 00 3B 08 0A 00 B2 20 2A        *
808- 2A 2A 20 41 50 50 4C 45  ** APPLE
810- 53 4F 46 54 20 52 45 53  SOFT RES
818- 45 52 56 45 44 20 57 4F  ERVED WO
820- 52 44 20 41 4E 44 20 54  RD AND T
828- 4F 4B 45 4E 20 4C 49 53  OKEN LIS
830- 54 45 52 20 2A 2A 2A 00  TER ***
838- 44 08 64 00 54 4B 4E D0      TKN
840- 31 32 3B 00 59 0B 6E 00  128
848- B1 41 44 52 D0 35 33 34   ADR 534
850- 35 36 C1 35 33 3B 35 34  56 53854
858- 00
```

NEXT LINE ADDRESSES

```
      $844   $838   $859
800- 00 3B 08 0A 00 B2 20 2A
808- 2A 2A 20 41 50 50 4C 45
810- 53 4F 46 54 20 52 45 53
818- 45 52 56 45 44 20 57 4F
820- 52 44 20 41 4E 44 20 54
828- 4F 4B 45 4E 20 4C 49 53
830- 54 45 52 20 2A 2A 2A 00
838- 44 08 64 00 54 4B 4E D0
840- 31 32 3B 00 59 0B 6E 00
848- B1 41 44 52 D0 35 33 34
850- 35 36 C1 35 33 3B 35 34
858- 00
```

CURRENT LINE NUMBER

```
       100    10    110
800- 00 3B 08 0A 00 B2 20 2A
808- 2A 2A 20 41 50 50 4C 45
810- 53 4F 46 54 20 52 45 53
818- 45 52 56 45 44 20 57 4F
820- 52 44 20 41 4E 44 20 54
828- 4F 4B 45 4E 20 4C 49 53
830- 54 45 52 20 2A 2A 2A 00
838- 44 08 64 00 54 4B 4E D0
840- 31 32 3B 00 59 0B 6E 00
848- B1 41 44 52 D0 35 33 34
850- 35 36 C1 35 33 3B 35 34
858- 00
```

TOKENS

```
   $81 (FOR)    $B2 (REM)
      $C1 (TO)     $D0 (=)
800- 00 3B 08 0A 00 B2 20 2A
808- 2A 2A 20 41 50 50 4C 45
810- 53 4F 46 54 20 52 45 53
818- 45 52 56 45 44 20 57 4F
820- 52 44 20 41 4E 44 20 54
828- 4F 4B 45 4E 20 4C 49 53
830- 54 45 52 20 2A 2A 2A 00
838- 44 08 64 00 54 4B 4E D0
840- 31 32 3B 00 59 0B 6E 00
848- B1 41 44 52 D0 35 33 34
850- 35 36 C1 35 33 3B 35 34
858- 00
```

Just to make sure we understand how all this works, let's change the *next line pointer* at the beginning of line 100 so that it points to line 120 instead of line 110. The next line pointers are aimed at each other, that is, they point one byte beyond the zero at the beginning of the program line. The zero for line 120 is at $858, so do this:

```
*838:59

*(control-C return)

]LIST
```

When you do this, you'll find that line 110 has disappeared from the listing. However, if you RUN the program, it will *still execute normally.* This is because the *next line pointers* aren't used when a program is executing sequentially from line to line. They are used by LIST, however, and by GOTO and GOSUB. You can actually mess up all the next line pointers in this program, since it doesn't use GOTO or GOSUB, and it will still RUN, though it won't LIST. We'll use this information later to split an Applesoft program into pieces. You can also create interesting effects by aiming the pointer for the first line at itself (use 801:1).

Simple variables. Now let's look at the variable tables. There are two tables, one for simple variables and one for array variables. Simple variables have a single element. In our earlier program, TKN, ADR, and V were all simple variables. These particular ones were also "real" or "floating point" variables. There are two other types of simple variables—integer variables such as A%, which can hold numbers between -32,767 and 32,767, and string variables such as A$.

In addition, functions defined by the user with the command DEF FN also use the simple variable table. When you define a function you give it a name, just like a variable name. If you DEF FN SHARE(X) = X * PCT, then the name of the function is SHARE. If, later in your program, you say B = FN SHARE(A), then the variable B will be set to the value of A times PCT. B and A, as well as PCT and X, also appear in the simple variable table as real variables. SHARE appears in the table as a function. X in this case is what's known as a "local" variable, because Applesoft saves its pre-existing value during the calculation and replaces it afterward.

Each item in the simple variable table is seven bytes long. The first two of these seven bytes are the two ASCII letters that make up the variable name.

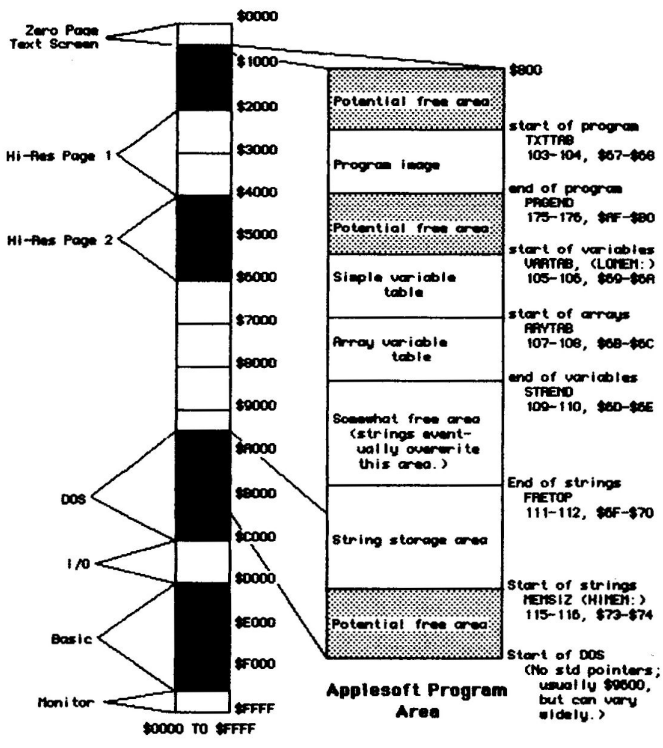


Figure 1

This is why variable names have just two significant characters, although you can make them longer for program readability if you want. If a variable name has just one letter, the second is given the value $00 or $80. Different combinations of low-value and high-value ASCII characters are used to designate whether an item in the table is a real, integer, or string variable, or a function definition pointer.

The following table summarizes the format of the various simple variables as they appear in the variable table:

FORMAT OF APPLESOFT SIMPLE VARIABLES

byte	reals	integers	strings	functions
0	low-ASCII	high-ASCII	low-ASCII	high-ASCII
1	low-ASCII	high-ASCII	high-ASCII	low-ASCII
2	exponent	high byte	length	low def address
3	mantissa	low byte	low address	high def address
4	"	unused	high address	low var address
5	"	"	unused	high var address
6	"	"	"	unknown

The value associated with a real or an integer variable is stored right inside the variable table. Real variables use five bytes to store the value; integer variables use two bytes. Note that integer variable values are stored high-byte first. This forward-thinking format is backwards from how two-byte numbers are usually stored in Apple memory.

The values associated with string variables and with function definitions are *not* stored in the variable table. Instead, the table holds pointers to the actual value. In the case of strings, one byte designates the string length (that's why Applesoft strings can't be longer than 255 characters) and two bytes point to where in memory the string is stored. This is usually in the string storage area, but it can also be inside a program if the program contains a statement such as A$="APRIL FOOL". In this case, the string address pointer in the variable table will be aimed at the ASCII characters inside the program image.

For functions, the "definition address" points to the first byte following the = in the function definition. The "variable address" points to the value of the major variable used in the function definition.

When Applesoft encounters the ASCII characters representing a variable in a program image, it searches for the current value of that variable by scanning the variable table from beginning to end. If the variable isn't found in the table, it is added and its value is set to zero. Thus, for maximum speed, variables that are used the most should be referenced early in a program so that they appear at the beginning of the table.

Another implication of the way variables are added is that real, integer, and string variables are all mixed together. Variables of the same type are not necessarily next to each other (unless a complusive programmer first references the variables in sorted order).

Array variables. Any array variables that are DIMensioned or encountered before simple variables must be moved higher in memory to make room for the new simple variable. Applesoft does this automatically, but it can take time. For maximum speed, all simple variables should be referenced before any arrays are dimensioned. Also note that arrays have a tendency to crawl around in programs that don't take this extra step. Just because you knew where an array was a few program lines ago doesn't necessarily mean you know where it is now.

The structure of the array-variable table is similar to the simple-variable table, except that it's completely different. There are three possible types of array variables—real, integer, and string. As with the simple variables, values assocated with real and integer arrays are stored within the array table itself; the actual strings associated with string arrays are not. For strings, the array table holds a length byte and a two-byte string address for each array element. For integer arrays, the table holds a two-byte value for each element; for reals, the table holds five-byte values.

Unlike simple variables, which always take up exactly seven bytes, the size of an array entry depends both on how many dimensions the array has and on the total number of elements in the array. By number of dimensions we mean how many different numbers appear in the parentheses in a statement such as DIM A$(10,10,10)—three in this case. By number of elements we mean the total number of values that can be stored in the array. Each dimension will hold the number of values you specify plus one. For example, DIM A%(4) creates an array with five elements, A%(0) through A%(4). For multiple-dimension arrays you multiply the number of elements in each dimension together. For example, DIM A$(10,10,10) has 11 * 11 * 11 or 1,331 elements.

The reason you have to DIMension an array is so that Applesoft knows how much memory space to allot for it. If you do not DIMension an array before you use it the first time, Applesoft will automatically dimension it for you with eleven elements in each dimension. Thus A$(0,0) = "ZERO" will do an automatic DIM A$(10,10) if A$ hasn't already been dimensioned. If you try to access an element of an array beyond the last element in a dimension, you'll get a BAD SUBSCRIPT error—the most frequent cause of this is forgetting to DIMension the array.

```
FORMAT OF APPLESOFT ARRAY VARIABLES

byte         real         integer         string

 0        low-ASCII     high-ASCII       low-ASCII
 1        low-ASCII     high-ASCII      high-ASCII
 2      low byte of distance to next array variable
 3     high byte of distance to next array variable
 4               number of dimensions
 5          size of last dimension, high byte
 6          size of last dimension, low byte
 7       size of next-to-last dimension, high byte
 8       size of next-to-last dimension, low byte
...
...
3+D*2       size of first dimension, high byte
4+D*2       size of first dimension, low byte
5+D*2       5-byte-       2-byte-          3-byte-
             values follow, element zero first
```

The string storage area. As mentioned earlier, strings that are defined within a program are left embedded there. Strings that are defined by INPUT from the keyboard or from disks, or by manipulating other strings, are stored in the string storage area. Thus, the value of string variables are split into two pieces—the length and location pointer, which are in the variable tables, and the actual string value, which can be just about anywhere.

When you change the value of a string with a statement such as A$=LEFT$(A$,4), the original value of A$ is left in the string storage area, but nothing any longer points to it. It becomes inactive. As you continue to use and redefine strings, the string storage space will expand into the remaining free memory until there isn't any left. At that point garbage collection will occur. During garbage collection all the strings that the variable table is pointing at are moved to the beginning of the string storage area and all other strings are electronically disintegrated. This can take awhile. For more information on garbage collection see ***Open-Apple*** for January 1985 (pages 4-5) and March 1985 (pages 17-19).

LOAD, RUN, and CHAIN. Each time you LOAD or RUN an Applesoft program, all existing variables are cleared to zero. If you are trying to break a long program into smaller segments that can be executed independently, however, you'll usually want to retain the values of your variables while switching between segments. Basic.system provides a CHAIN command, and DOS 3.3 provides a disk-based CHAIN routine, that allow you to switch between programs without losing the values of your variables.

The Basic.system CHAIN command has the same syntax as RUN. It must be followed by a pathname and may use optional parameters for slot, drive, and a starting program line (using the @ parameter). For example, a program that creates graphs and charts could be broken into a "program initialization and main menu" segment, a "data entry and editing" segment, a "graph creation" segment, and a "graph editing and printing" segment.

The program initialization segment should move all the program segments (including itself) onto a RAMdisk, if one is present (see the July 1985 ***Open-Apple***, pages 50-52, and October 1985, pages 74-76, for complete details on how to accomplish this). The main menu would then prompt the user and respond by CHAINing to one of the other segments with a command like this one:

```
4100 PRINT CHR$(4);"CHAIN /RAM/DATA.ENTRY"
```

Basic.system would then load and run the data entry segment. Suppose the main menu program had initialization instructions from lines 0 through 4999, and the menu itself began at line 5000. When the data entry segment returns to the menu, it wouldn't want to reinitialize everything, so it could use a command like this:

```
9000 PRINT CHR$(4);"CHAIN /RAM/MAIN.MENU, @5000"
```

Under DOS 3.3, an Applesoft CHAIN is not a built-in command, but is instead a 456-byte binary file found on the DOS 3.3 system master disk. Here's an example of how you would use it:

```
1000 PRINT CHR$(4);"BLOAD CHAIN, A$20"
1010 CALL 520"DATA ANALYSIS"
```

Do not put a space betwen the 520 and the opening quote in the second line of this procedure, or it will fail to work. The two lines load the CHAIN routine into the keyboard input buffer and page-3 free space ($208-3CF) and execute it. DOS 3.3 doesn't support the @ parameter.

Essentially the DOS 3.3 and Basic.system chain routines do the same thing. First, garbage collection is performed to concentrate the active strings at the DOS end of memory. Next, strings that are embedded inside the program segment that is about to be overwritten are also placed in the string storage area. Then both of the variable tables are moved so they lie against the string storage area. This maximizes the program area.

Once all the variables have been concentrated at the DOS end of memory, the new program segment is loaded over the top of the old one. Then the variable tables are moved back to the program end of memory and execution continues. When the variable tables are moved, CHAIN places them next to the new program segment, at PRGEND. Neither the DOS 3.3 nor Basic.system versions of CHAIN pay attention to whether you had reset LOMEM: to avoid overwriting the high-resolution screens. When CHAIN moves the variables back down, it always puts them at PRGEND, adjacent to the end of the new program segment.

Neither DOS 3.3 nor Basic.system correctly handle functions in CHAINed programs. You cannot use DEF FN in your main program and then use FN in a CHAINed segment, unless you redefine the function in that segment.

Another significant problem with the Basic.system version of CHAIN is a bug that causes it to forget the variable values if the variable tables are an exact multiple of 256 bytes long. This bug was first demonstrated in the June 1985 *Call-A.P.P.L.E.* (page 30) by Peter Meyer, and a solution was given in the November 1985 *Call-A.P.P.L.E.* (page 49) by Oliver Mayes. The problem is a bad branch instruction inside Basic.system. A simple poke will fix it. You should include this poke in any ProDOS-based program that uses CHAIN:

```
IF PEEK(49149)=1 THEN POKE 41859,3 : REM test for V1.1 and fix CHAIN
```

The command checks to make sure version 1.1 of Basic.system is active by looking at the system-program version number, which is kept at $BFFD in the ProDOS global page. If this is not a one, this POKE won't work. Instead it would mess up something else inside Basic.system, so we don't allow it.

STORE and RESTORE. Basic.system includes commands called STORE and RESTORE that are used to save and load a complete set of variables while leaving the current program intact. DOS 3.3 has no similar commands. STORE uses the same subroutines as CHAIN to concentrate all the variables at the DOS end of memory, then it saves the image in a special file type called VAR. RESTORE loads VAR files at the DOS end of memory, then calls the same subroutines as CHAIN to move the variable tables back to the program end of memory. Unlike CHAIN, however, RESTORE honors the current LOMEM: setting.

STORE and RESTORE are handy for quickly saving and reloading all the variables in a program. RESTORE can also be used to quickly initialize a program with a whole bunch of preset variables. Apple itself uses VAR files to solve the foreign-language problem. Some programs require the user to specify what language their menus and instructions should use—the programs then select and RESTORE a VAR file that has string variables that use that language. This is a neat trick.

Nonetheless, VAR files cannot easily be used to divide a large number of variables into groups for one-by-one processing. The problem is that STORE and RESTORE insist on handling all variables at once. What is needed is a way to save and load just a portion of the existing variables.

BSAVE and BLOAD with variables. You can easily do such tricks, with numeric variables anyhow, using BSAVE and BLOAD. String variables, on the other hand, are very difficult to manipulate because they are stored in two parts.

In order to save several simple variables, you only have to make their first reference sequential, so that they will lie next to each other in the variable table; calculate their length, which will always be the number of variables times seven; and determine the starting address. The starting address is easy to come by because Applesoft always leaves bytes 131 and 132 ($83-$84) pointing at the last variable it has used.

Retrieving this value can be a bit tricky, however, since any variable used in the statement retrieving the value becomes the "last variable used." But

suppose you want to save the values in three simple variables, D, F%, and I. Here's how to do it:

```
10 A=0 : D=0 : F%=0 : I=0 : REM first reference--puts these variables
                            in adjacent bytes, with A first

600 A=PEEK(131) + PEEK(132)*256 - 2
610 PRINT CHR$(4);"BSAVE /RAM/D.F.I, A";A; ",L";4*7
```

In this trick, we set up a dummy variable, A, as the first in the series of variables we want to save. Line 600 references A to get its address in bytes 131 and 132, then stores the result in A itself. Since this address is actually the starting address of A's five "value" bytes, we have to subtract 2 in order to back up over the variable name. Line 610 uses A to save the contents of the four variables in a binary file.

To reload the variables into another program or program segment, you must take all of the *exact same* steps, except change BSAVE to BLOAD (under DOS 3.3 you must also delete the length parameter from the BLOAD command). Note that you are BLOADing the variables' *names* as well as their values.

A similar trick can be used with numeric *array* variables, but it's simpler not to save the array name or dimensions. Refer to the zeroth element of the array to find where its values begin. The array's length is five times the number of elements in floating point arrays and two times the number of elements in integer arrays. But remember that the "number of elements" is one more than the number you DIMensioned the array with, because of the zeroth element. Here are some examples, where S represents the size of the array and AL its length:

```
 floating point arrays: DIM A(S)  : AL = (S+1) * 5
       integer arrays: DIM A%(S) : AL = (S+1) * 2
multi-dimension arrays: DIM A(S1,S2,S3,S4) :
                            AL = (S1+1) * (S2+1) * (S3+1) * (S4+1) * 5
```

Once you know the length of an array, you can determine its address as we did earlier. If you are unwilling to sacrifice the array's zeroth element as temporary holder of the array's address, here's another way to capture it.

```
AA=A(0) : POKE 78,PEEK(131) : POKE 79,PEEK(132) : AA=PEEK(78) + PEEK(79)*256
```

If you use this trick, make sure the simple variable that will hold the array address appears first, then the zeroth element of the array. If it is a multi-dimensional array, the reference should be, for example, AA=A(0,0,0,0). Also make sure you calculate the length of the array before finding its address.

The most common problem programmers have with putting arrays in binary files is using a simple variable for calculating length, for the first time, after finding the array's location. When the length variable hasn't been used before, it is entered in the simple variable table, which moves the array before it can be saved. Another mistake programmers tend to make is loading data into an array of different dimensions than the array the data was saved from. Don't do it.

One major advantage of these tricks is that they allow you to quickly save and reload a large number of numeric values. Binary files are much faster than standard text files under both DOS 3.3 and ProDOS. These tricks, used in conjunction with the CLEAR command, also allow you to do things like redimension arrays. Use BSAVE to retain the few variables you need to retain, use CLEAR to erase everything, then reinitialize and BLOAD the needed variables. Your slate will be clean.

Another big advantage of these tricks is that they allow you to segment your data and actually process more numbers than will fit in memory at one time. For example, you might want to work on an array that is 60 elements wide and 500 elements deep. Even if your program is relatively small, however, about the largest array you'll be able to get into memory at one time will be 60 by 100 elements.

But it's entirely possible to have five binary files, each holding a 60 by 100 piece of the data. All you have to do is sequentially load the five files into what Applesoft thinks is the same array for processing.

BSAVE and BLOAD with programs. While CHAIN can be extremely helpful in segmenting programs, it isn't perfect. In addition to the LOMEM: problem noted earlier, CHAIN replaces the entire program in memory with something new. But usually you'll have a set of subroutines and perhaps even a main menu that all of your program segments should share. It is a waste of disk space and of time to put these routines in each segment and constantly reload them.

By using binary files, however, you can overlay just a portion of an Applesoft program. In order to do this, all you have to do is get the next line pointers that are embedded in the overlays to match the memory area they will be loaded into, and to link the overlays to your main set of subroutines.

The easy way to proceed is to write all of your overlays so that they begin with the same line number, say 10000. Your main program, which will now *always* be ready in memory, should have line numbers lower than that. Use the final line of your main program to jump to the current overlay like this:

```
9997 ADR = 16384 : REM byte where all overlays start
9998 HI = INT((ADR+1)/256) : LOW = (ADR+1)-(HI*256)
9999 HERE = PEEK(121) + PEEK(122)*256 :
     POKE HERE+1,LO : POKE HERE+2,HI : GOTO 10000
```

Bytes 121 and 122 ($79-$7A) point at the zero in front of the line currently being executed. Line 9999 changes its own next-line pointer so that it points at the first next-line pointer in the overlay. The GOTO 10000 at the end of that line makes Applesoft start searching through the program, using the next-line pointers, looking for line 10000. When it gets to line 9999 it will be directed to line 10000 automatically.

The address you use in line 9997 can be right at the end of the main body of your program if you like, but I recommend you be easy on yourself and leave a few hundred bytes of empty space between your main program and your overlays so that you can add a few lines to the main program without having to move the overlay. If you want, you can move the overlay a greater distance, such as beyond the graphics pages. In fact, you can use this trick just to split a program in two even if you don't intend to use an overlay.

To get everything to work, use LOMEM: at the beginning of your main routines to move the variable tables out beyond the end of your longest overlay.

You also have to get the Applesoft program segments that you want to overlay into binary files. This way you will be able to BLOAD them without disturbing any variables or the main program. The easiest way to correct the embedded next-line pointers is to get DOS to LOAD the segment at the correct address, then BSAVE it. To do this, temporarily change TXTTAB at bytes 103 and 104 ($67-$68). This pointer tells Applesoft where a program's first next-line pointer is. DOS uses it to figure out where to load Applesoft programs.

After you change TXTTAB and load your program, you'll also need to know how long the program is. PRGEND at bytes 175 and 176 ($AF-$B0) holds the address of the end of the program. Here's how to use all this stuff to change an Applesoft program segment named OVERLAY into a binary file called OVERLAY.IMAGE that will execute at $4000, just above high-res page one:

```
ADR=16384 : POKE ADR,0                          overlay must start with 0
HI=INT((ADR+1)/256)                             calculate high byte
LO=(ADR+1)-HI*256                               calculate low byte
POKE 103,LO                                     change TXTTAB
POKE 104,HI
LOAD OVERLAY                                    load program segment
PRINT (PEEK(175) + PEEK(176)*256) - ADR         calc program length
1234                                            answer, use in BSAVE cmd
BSAVE OVERLAY.IMAGE,A16384,L1234
```

If you have several program segments, LOAD them one after the other, find their lengths, and BSAVE their images. To load an overlay, simply add a command like this one near the end of your main body of routines:

```
9996 PRINT CHR$(4);"BLOAD /RAM/OVERLAY.IMAGE.";N
```

If you name your overlay images .1, .2, and so on, your main routines simply have to set N to the number of the overlay you want and GOTO 9996. The overlay will be loaded and line 9999 will jump-start it.

Do not attempt to edit an overlay while it is linked to a main program like this. Applesoft's editing genie gets quite upset and will crash your system. However, the LIST genie doesn't mind a linked image and can show you an overlay—until you edit the main program. In that case, the editing genie will unlink the overlay and you won't be able to LIST it anymore.

If you attempt these tricks, your finished program should begin with a STARTUP program that takes the following steps:

```
* copy all program and variable segments to /RAM
* CLEAR copy variables, move LOMEM beyond end of longest overlay
* initialize or RESTORE main program variables
* fix the CHAIN command (ProDOS--IF PEEK(49149)=1 THEN POKE 41859,3)
* CHAIN to the main program segment
```

Now, finally, you've expanded Applesoft beyond its inherent 48K memory limitation.

Ask (or tell) Uncle DOS

& to the rescue

In the March issue of ***Open-Apple*** there were quite a number of questions that can be answered with ampersand commands from the Toolbox series of programs that I publish.

A number of years ago I started on a project that was originally called *The Routine Machine*. The system has evolved into four packages: *The Wizard's Toolbox, The Database Toolbox, The Video Toolbox,* and the *Chart 'n Graph Toolbox.*

All of the ampersand commands included in these packages can be added on a pick-and-choose basis. You can choose the commands you wish to add from any of the Toolbox disks in any order you want at any time while you're writing your program. Add a routine here; add a few lines of Applesoft; add another command; delete some lines; and so on. Any order, no constraints.

No knowlege of PEEKs or POKEs or other high-level black magic is required. You decide you want a new command; it takes just 30 seconds to add it; and you're back working on the Applesoft portion of your program. The program can be saved as a single Applesoft file, loaded the next day, even converted to ProDOS. No modified DOS, no pre-arranged BLOADs, no changing of HIMEM: are required.

A *huge* library of existing routines covers just about everything you'd ever want to add. Each Toolbox package has anywhere from 30 to 60 commands in it, for a total of hundreds of commands available for the Applesoft programmer.

For example, the problem of rounding errors from Linda Bunney (page 2.15) can be solved with the PRINT USING command from *The Wizard's Toolbox.* We have an older Toolbox disk called &Sampler I that has both a formula-evaluation routine like David Atkins is looking for (page 2.16) and the IF-THEN-ELSE that Nick Andritsakis wants (page 2.15). David Sparks has written some toolbox routines for 99 digit math. *The Video Toolbox* contains a number of specialized input routines, and although the specific "underline cursor" isn't one of them, the variety of functions that are provided would fill just about any need an Applesfot programmer would have.

Programmers can use any of the toolbox routines in commercial software with just a credit line and a completed license agreement from us. This is similar to Apple's procedure with DOS, except that we don't charge money.

I'm willing to make a special deal for your subscribers who are interested in investigating this further. I'll send anyone who mentions he or she read of the possibilities in your newsletter a free copy of our *Trial Size Toolbox* (normally $3). This disk includes eight ampersand commands, including a charting command-set with 12 sub-commands, a fixed-length input command, and a print with word-wrap command, usable under either DOS 3.3 or ProDOS. Also on the disk is a 50-page manual. The manual includes a tutorial for the toolbox system, a complete explanation of the commands included on the sampler disk, and a comprehensive listing of every command in each of our Toolbox packages.

Roger Wagner
Roger Wagner Publishing, Inc.
P.O. Box 582
Santee, Calif. 92071

Well, hello, Roger. I remember you. Once upon a time we were both sages for ***Softalk****'s IF-THEN-MAYBE column. You had sensible answers to readers' questions back in those days, too.*

As you point out (and as I forgot), ampersand routines that solve a number of the problems raised by readers last month are available.

In addition to your own large library of integrated routines, other major sources of ampersand commands are MicroSPARC (publishers of ***Nibble*** *magazine, 10 Lewis St, Lincoln, MA 01773) and Beagle Bros (3990 Old Town Ave, San Diego, CA 92110). MicroSPARC publishes* ***Ampersoft*** *utility libraries written by Cornelis Bongers in both DOS 3.3 and ProDOS versions. Beagle Bros includes ampersand commands on a number of its disks, including* ***Beagle Graphics, Extra K,*** *and* ***Big U.*** *Most Apple user groups have a public domain disk called* ***Binary Linking Utilities*** *(IAC #46) that includes a number of ampersand commands for DOS 3.3.*

Besides ampersand routines, ProDOS "added commands" can provide similar capabilities. The July 1985 ***Open-Apple*** *(page 52) discusses a couple of disks from Glen Bredon and from Night Owl Productions that can also help push back the limits of Applesoft.*

Even the double-precision arithmetic problem *that I had no answer to last month can be solved with ampersand routines. Bob Sander-Cederlof at S-C Software reminded me this month that he published an 18-digit, binary-coded-decimal arithmetic package for Applesoft in his newsletter,* ***Apple Assembly Line****, between May 1984 and February 1985. The package, which is called* ***DP18****, interfaces with Applesoft using the ampersand command. It includes all the math functions of Applesoft, as well PRINT USING and INPUT USING commands. It stores its variables in normal Applesoft arrays, except that each number takes two array positions (a one-byte exponent and nine-byte mantissa for 10 total bytes).*

S-C Software (P.O. Box 280300, Dallas, TX 75228 214-324-2050) was also the developer of the ***Double Precision Floating Point for Applesoft*** *package, once published by Hayden, that we mentioned last month. This package has 21 digits of precision, but doesn't use BCD arithmetic, so it has the same rounding flaws as Applesoft. It evaluates expressions from left to right, rather than giving precedence to multiplication and division, as Applesoft and* ***DP18*** *do. But, on the positive side, it uses about one-third the memory of the 18-digit package and has a more polished manual. Either package is available on disk, with instructions and source code, for $50.*

There are even non-ampersand answers to some of last month's questions. A technician at Apple called to remind me about a formula on page 18 of the original Applesoft reference manual that shows how to limit the number of digits that will print to the right of the decimal point. It uses rounding and is faster and more accurate than the string manipulations I demonstrated last month. The formula, where D is the number of decimal places you want, is:

```
X = INT(X * 10^D + .5)  / INT(10^D + .5)

   or, if P=10^D  (i.e., 10 for one place, 100 for
two places, etc.)

X = INT(X * P + .5) / P
```

This works as long as X >= 1 *and* X < 999999999.

In regard to the Apple DMP superscript blues problem, Apple's technician said that Imagewriters and very late DMPs have the carriage-return on linefeed bug removed and will do superscripts and subscripts normally. He also said all but very early DMPs came with a downloadable character set that provides small-letter superscripts and subscripts. Rather than using reverse linefeeds, you simply turn on the alternate character set and print either caps (for superscripts) or lower-case (for subscripts).

1200 baud characters lost

I use an original Apple IIe equipped with an Apple Super Serial Card at my office to communicate with IBM PC's and other computers such as our MicroVax II. I usually use ASCII Express as a terminal program. The Apple frequently loses characters sent from the other machines. I've tried switching baud rates but the problem exists even at 1200 baud, which is supposed to be within the capabilities of the hardware. The mystery is that my Apple IIc doesn't have this problem. What's going on? And, what I can do about it?

Dr. S. A. Klein
Madison, Wisc.

I have a Novation Apple-Cat II modem with a 212 upgrade card installed in my Apple IIe. My primary communications software is ASCII Express. I had been able, with that combination, to transmit and receive at 1200 baud. I recently had my IIe enhanced. I now find that I am unable to use the 1200 baud feature of my modem. While everything works fine at 300 baud, my telecomputing needs require the faster rate. Do you know of a fix for this problem?

Peter J. Paul
Middle Village, N.Y.

Modems and terminal programs aren't my specialty, so I asked ***Open-Apple*** *technical consultant Dennis Doms your questions and he provided these answers:*

To keep up with 1200 baud on ***any*** *Apple II, you must have your serial card or slot-based modem send an interrupt as each character is received. The interrupt tells the terminal program to stop updating the screen display and come get the waiting character. At 1200 baud the characters come in at the rate of about 120 a second, so the interrupts are fast and furious.*

However, on the original IIe, you will ***still*** *lose characters at the beginning of each line, unless the sending computer pauses after sending a carriage return. This is because the 80-column firmware on that machine disables interrupts as it scrolls the screen. A scroll takes longer than the time available between incoming characters at 1200 baud (but not at 300 baud). This problem was fixed on the IIc and enhanced IIe.*

If you have an enhanced IIe and you are losing characters, the problem is probably that you don't have interrupts turned on. On the Apple Super Serial Card, switch 2-6 controls whether the card sends an interrupt when a character has been received. This switch should be ON. ASCII Express and the Novation

modem are supposed to enter interrupt mode automatically. Check the ASCII Express "status" (!) display to find out if interrupts are enabled. If not, you may have a hardware problem. Pages 275 to 281 of the ASCII Express manual go into great detail on the interrupt support provided by various modems and serial interface cards (although the interrupt-enable switch given for the Super Serial Card is wrong—it really is SW2-6, not SW1-6).

I have used an enhanced IIe, a Super Serial Card, ASCII Express, and interrupts as a terminal for a DEC minicomputer at 4800 baud without losing characters. This is the highest baud rate ASCII Express supports. I am told that Apple's Access II supports 9600 baud, but I haven't seen it done myself.

***Open-Apple** has pointed this out before (page 38), but it's worth repeating that when switching between an original and an enhanced IIe you must run the ASCII Express INSTALL program and change the "local console" selection from "Apple IIe" to either "Auto" or "Pascal 1.1." The special IIe driver built into AE Pro assumes you have the flaky original IIe ROMs, and will not function correctly on the enhanced IIe.*

If, after taking all of the above steps, you are still getting occasional garbage, add the following characters to the "character suppression" list within the AE Pro INSTALL program: ^O, ^N, ^Q, ^U, ^F, ^@, ^#, ^', and ^[. Several of these characters interact with the 80-column firmware; control-O sets inverse display mode, for example, and ^[(the escape character) can activate MouseText and trigger other effects. Control-F causes the cursor to become hidden if you are using Pascal protocol. Control-Q switches to a 40-column display and control-U will turn the firmware completely off (CRASH!!).

*Control-caret is a new GOTO XY command in the enhanced IIe firmware. Using the ASCII code of the following two characters, minus 32, it moves the cursor to horizontal position X, vertical position Y. All other firmware control-codes were listed in the March 1985 **Open-Apple**, page 23. There are a few codes in my list that I have disabled for some reason I've forgotten. I'm not sure what their negative effects are. You may wish to experiment with those.*

AppleWorks page nos. (cont)

AppleWorks has a definite bug in its page-number processing. I recently finished a 368-page novel, and AppleWorks' automatic numbering screws up many numbers after 200. Some pages are numbered correctly, while others get totally wrong numbers. Any patches to fix this problem?

Chet Day
Metairie, La.

I don't know of a fix.

CONVERTing ProDOS

This letter concerns the letter published in your February 1986 issue (page 2.8) about moving the file PRODOS over to DOS 3.3. I did this some time ago and found that simply CONVERTing it was not quite enough to make it work. When I tried to run the file from DOS 3.3 I got a RELOCATION/CONFIGURATION ERROR.

I got out the old machine language thinking cap and did some digging. It only took about 5 minutes to figure out that ProDOS expects to find the boot slot times 16 in zero page location $43 (decimal 67). There are two approaches to fix this problem. One is to POKE 67, SLOT*16 before running the file (big hassle). The second is to change the instruction that looks in $43 for the slot to some instructions that look in the DOS 3.3 Input/Output control Block (IOB) instead. I personally like number two so I took the following steps to change the PRODOS file so that it would work:

```
BLOAD PRODOS,A$2000          Get the file in memory
CALL -151                    Enter the Monitor
1FF7:20 E3 03 B4 4E B5 4F A0 0F B1 4E
                             Get slot from IOB, not $43
BSAVE PRODOS,A$1FF7,L$3A0A   Save it back to disk.
```

And that's all that needs to be done. The code we put in disassembles as:

```
1FF7: 20 E3 03  JSR $3E3     Find IOB
1FFA: B4 4E     STY $4E        low byte, high-byte
1FFC: B5 4F     STA $4F      Last slot used * 16
1FFE: A0 0F     LDY #$0F       is 15 bytes inside IOB
2000: B1 4E     LDA ($4E),Y  Get it
```

The reason I did this in the first place was because of an article by Ken Manly in the August 1985 issue of *Nibble* (page 40). This article showed how to put both DOS and ProDOS on the same disk and presented a program that allowed you to select either operating system on booting the disk. It also showed how to put DOS 3.3 into a ProDOS system file. I wanted to be able to switch back and forth between the two so I also moved ProDOS over to DOS 3.3.

Martin L. Wallgren
Prophetstown, Ill.

Soft write-protect tabs

Is there a POKE that will allow me to save a file on a write-protected disk? I don't want to open up my IIc to jump a wire or something like that.

Daniel Charette
Montreal, Que.

If you are using DOS 3.3 at its standard 48K location, POKE 47158,234 : POKE 47159,234 will allow you to save files on a write-protected disk. If you want to initialize write-protected disks, then four pokes are necessary—the two already given plus POKE 48221,234 : POKE 48222,234. But note that if you boot from one of these disks the pokes will be intact within DOS and your write-protect tabs won't protect anything.

To change things back to normal operation, use POKE 47158,48 : POKE 47159,124 : POKE 48221,48 : POKE 48222,94. Similar pokes for those who like to play with fire are probably available for ProDOS, but I don't know what they are.

More disk interference

In reference to the two letters in your February 1986 issue (page 2.8), one does not need to upgrade to the UniDisk 3.5 in order to discover problems with the IIc's new color monitor. When I attached one to my unenhanced IIc, many copy-protected programs and some that aren't refused to boot, displaying the "Check Disk Drive" message and/or interminable spinning and occasional clacking noises. Those that did boot didn't always run properly, but did, however, when the old monitor was re-attached.

I now move the computer to the side, completely away from the front of the monitor, boot, and only then move it back. In this way, all programs will run, but I don't consider it a satisfactory solution. If Apple is serious about giving more help to its customers, it should correct this problem not only for future buyers but for those of us who already own the monitor.

The two technicians in the service department of the store at which I bought the monitor say they have never run into this before, but obviously I'm not the only one with this problem.

Bernice Eaton
Northridge, Calif.

The letters in the February 1986 issue (page 2.8) regarding color monitor interference sounded familiar to me, but involved different equipment. I have had problems getting disks to boot using both an Apple RGB monitor and an Apple Composite Monitor when the monitors are stacked on top of Disk II Drives. I have not had any similar problems with monochrome monitors.

The monitors are used in our high school computer lab, so I have been able to test different combinations of drives and have found that most of our drives are not affected by interference from these monitors. The problem seems to be related to individual drive characteristics rather than to production grouping (age, serial number, or manufacturer).

At the risk of re-opening a previous can of bugs, my worst partially-solved problems concern printing using the AppleWorks program. I have to use three "non-standard" but common printers (Comrex CR-1, Diablo D-25, and Okidata 82a) and three non-standard and somewhat troublesome solutions to get acceptable output, even though the first two printers are connected to Apple Super Serial Cards. I will be glad to share problems or solutions with anyone interested.

Thanks for providing a wealth of good information and a much-needed forum for exchange.

Robert Tighe
P.O. Box 1153
Chinle, AZ 86503

Brand new revisions to the technical notes Apple provides its service technicians say that failure-to-boot and other disk problems on the Apple IIc can be caused by an unshielded read/write head assembly inside the disk drive. On Apple drives, the magnetic read/write head pushes against the bottom side of the disk. Providing counter-pressure from the top is a non-abrasive "load button". The load button pops on and off easily so it can be replaced when it wears out. However, it is also supposed to hold a round, metal shield on the drive to protect the read/write head from stray magnetic energy. Apparently this shield is missing from some drives. Apple will send your dealer extra shields at no cost.

*Tell your technicians to look at the March 1986 revision of page 1.17 of the **Apple Technical Procedures Manual, Vol 1**. If this solves the problem, let us know—this shield may be missing from lots of drives, and not just on the IIc.*

HTAB 1 still broken

The new enhanced IIe ROMs do not correct all problems with the HTAB function.

After some experimenting I have found that HTAB 1 in 80-column mode does not work correctly unless the command is the first print-related instruction after a carriage return.

The problem can be demonstrated with a simple program such as:

```
10 FOR J = 1 TO 5
20 FOR I = 1 TO 5 : HTAB J : PRINT "A"; : NEXT
30 FOR I = 1 TO 5 : HTAB J : PRINT "B"; : NEXT
40 PRINT: NEXT
```

If this program is run in 80 column mode the screen will show a display of:

```
AAAAABBBBB
 B
  B
   B
    B
```

Using the source code provided in *The Programmer's Guide to the Enhanced IIe*, I have traced the problem to the instruction at $C857 in the new ROM. At this location the current horizontal cursor position according to HTAB, CH (36, $24), is compared with the value in OLDCH (1147, $47B). If the two values are not equal, the 80-column cursor location, OURCH (1403, $57B) is updated. If the two values are equal, on the other hand, the value in CH is ignored and OURCH isn't changed.

The value in OLDCH is kept at zero by code at $C8D1 after each character is placed on the screen. Thus, whenever CH also holds a zero (after HTAB 1, for example), OURCH is not updated. The variable OLDCH does not appear to have any other use.

There appears to be no simple fix for the bug except continuing to use POKE 1403,X-1 for 80-column HTABs. For example, try changing the HTAB J in the above sample program to POKE 1403, J-1 and see what happens.

D. G. Chapman
Winnipeg, Manitoba

I also tested your program on an Apple IIc in 80-column mode—it exhibits the same bug.

A bit too many

Do you know a simple way to send a text file from Applesoft through the Apple Super Serial Card and a modem to the outside world with the high bit OFF?

Lambert van Beers
Tokyo, Japan

Open-Apple

[redacted]

is written, edited, published, and

ISSN 0885-4017. Published monthly since January 1985. Worldwide 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; seven or more from any single volume $14 (postpaid). Index mailed with the February issue. ***Open-Apple*** is available on disk for speech synthesizer users from Speech Enterprises, P.O. Box 7986, Houston, Texas 77270 (713-461-1666).

Source Mail: TCF238 CompuServe: 70120,202 Tele.: off hook

Enclosed is a copy of an Applesoft program for printing graphics on an Imagewriter with a Super Serial Card. As you can see, the graphic appears to be underlined because I can't figure out how to turn off the eighth bit. Can you help?

Melvin Brubaker
Dillsburg, Pa.

*Both of your programs use PRINT CHR$(n) to send characters to the serial card one at a time. Your problem is that **Applesoft** always sets the high bit of any character it prints.*

To avoid the problem, it is necessary to send the characters straight to COUT (the Apple Monitor's print routine) rather than using Applesoft's PRINT statement.

This requires a very simple machine language program. Here's what it looks like:

```
0300:A9 00      LDA #$00    load character
0302:20 ED FD   JSR $FDED   send character to COUT
0305:60         RTS         back to Applesoft
```

To use the routine, POKE the character you want to send into the byte shown holding a zero, and call the routine. Here's a modified version of your graphics program that uses this trick.

```
10  REM *** GRAPHICS.TEST ***

20  REM set up routine to send correct code
30  POKE 768,169 : POKE 769,0
40  POKE 770,32  : POKE 771,237 : POKE 772,253
50  POKE 773,96

60  REM print out a line of graphics codes
70  PRINT CHR$(4);"PR#1" : REM activate printer
80  PRINT : REM set at left margin of paper
90  PRINT CHR$(27);"G0256": REM 256 graphics codes

100 FOR X = 0 TO 255 : REM X = code to print
110 POKE 769,X : CALL 768 : REM send char
120 NEXT

130 REM rehook screen
140 IF PEEK(978)=157 THEN POKE 43602,0 :
                          REM fake <CR> for DOS 3.3
150 PRINT CHR$(4);"PR#0"
160 END
```

The modem problem should be solvable by the same method, using the machine language subroutine to send the data. However, you must also be sure that the modem and the interface are set to pass 8-bit data and that the receiving terminal is prepared to receive it.

To the max

On a DOS 3.3 Disk, if my objective is to store a small number (less than seven) of large files, how do I free the absolute maximum number of sectors for the files? Why is it that a disk formatted without DOS still uses all of track zero?

Dan Strassberg
Arlington, Mass.

DOS 3.3 uses zeros to indicate "end of file" or "no more data." For example, the second and third bytes of each catalog sector point to the following catalog sector. If you change these to zeros, DOS 3.3 assumes that there are no more—that the current catalog sector is the last catalog sector.

To maximize DOS 3.3 disk space, use a disk zap utility to change the second and third bytes in the first catalog sector—track 17 ($11), sector 15 ($F) to zeros. After that, you will only be able to save seven files on this disk, but you will also be able to free up the remaining catalog sectors for data storage.

To do that, load the VTOC from track 17 ($11), sector 0 and change bytes 124 and 125 ($7C-$7D) to 127 and 254 ($7F FE). This frees up all the sectors in the catalog track except the VTOC sector itself and the single catalog sector you created earlier.

However, actually saving files in these newly freed sectors requires more tricks. DOS 3.3 always assumes that the catalog track is full when saving files—it doesn't even look there for empty space. To use your newly created space on this track you can copy files onto the new disk using FID, which makes no assumptions about the catalog track, or you can POKE 45714,169 with DOS 3.3 at its normal 48K location and it will begin to store files in the catalog track automatically.

*If you insist on using track zero for data storage as well, take a look at Clay Ruth's series on the subject in **Call·A.P.P.L.E.**. In March 1982 there was "Get the Most From a Diskette" (page 13), in April 1982 "Get Most Followup" (page 71), and in December 1982 "Yes, You Really Can Store Files on Track Zero," (page 57). Extensive DOS modifications are required—I personally don't think they're worth the trouble.*

Program not NORMAL

Here's an interesting problem I hope you can explain. I have enclosed a program one of my students has written. It contains several errors, but the one in line 140 has an unusual effect, to say the least. When the program is run, it crashes and control-reset is necessary to unlock the computer. When the program is listed, it appears totally destroyed—full of lower case letters and punctuation marks.

However, I can recover the original program by saving this garbage to disk, rebooting, and reloading the program. It then lists normally until it is rerun. Merely saving, loading and listing does not work—a reboot is required.

This situation occurs infrequently; perhaps one or twice a semester, but it is disconcerting to the student involved when it does happen. The method of saving, rebooting, listing, and repairing usually works, but not always. What is going on here? Can a seemingly simple syntax error cause so much trouble?

Guy Nelson
Rotan, TX

Do not adjust your set, nothing is seriously wrong. The program did not sabotage your hardware; it's just that reset didn't fix everything.

The problem is that when the program hung, Applesoft's FLASH command was active. FLASH causes a $7F to be stored in location $32 (INVMSK) and a $40 to be stored in $F3 (ORMSK). NORMAL sets these values to $FF and $00 respectively. INVERSE sets them to $3F and $00. As Applesoft sends a character to the screen or printer, the character is "masked" against INVMSK and ORMSK. This is what controls how the character actually appears on the screen—whether normal, inverse, or flashing.

When you press control-reset, the Monitor sets what it thinks is NORMAL mode by correcting the contents of INVMSK to $FF. However, the Monitor itself does not use or correct ORMSK. Thus you end up with NORMAL's $FF in INVMSK and FLASH's $40 in ORMSK. This particular combination generates the garbage you see on your screen when you list the program.

It isn't necessary to reboot to fix things. Simply type NORMAL. Rebooting fixes the problem because it re-initializes Applesoft; FP or NEW would have the same effect (but would erase your program from memory).