Open-Apple™

March 1987
Vol. 3, No. 2

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

Releasing the power to everyone.

Reading AppleWorks data bases

While a few of you are still hold outs, most folks in the Apple II kingdom have become very fond of AppleWorks in general and of the AppleWorks data base in particular. The great strength of the AppleWorks data base is speed. It sorts and retrieves data fast enough to make mainframes blush. Data entry and editing is smooth and quick.

The amount of data that will fit in one record, of course, is limited to 30 different categories and what will fit on a single screen. The number of records that will fit in a single file depends only on what version of which company's RAMcard expansion software you have.

For getting the data in your file out onto paper, AppleWorks allows you to define eight report formats. While these formats are sufficient for many types of reports, they pose significant limits for others.

You can't, for example, dump your AppleWorks data onto a pre-printed form if more than 15 lines separate the highest and lowest areas you must fill out. If you use the AppleWorks data base to fill out continuous credit card forms (as we do around here), you have to include your company name and merchant number in every record; there's no other way to tell AppleWorks to print the same thing—even so much as a comma between city and state—on every form.

There are a few fairly easy solutions to these problems. The mail merge features of *AutoWorks* and of the new AppleWorks 2.0 will solve both of the problems I've just mentioned. Another possibility is to open-apple-P(rint) your data base into a *file*, then read that file with your own program and manipulate the data any way you like. You could even update the data (for example, deduct today's sales from your toy store's inventory) and store the updated data in a DIF file. The DIF file could be loaded into AppleWorks as a new data base. The records in that new data base could be copied, using the clipboard, into the old data base and the old records deleted. This process is cumbersome, however, and not without some ill effects, such as the disappearance of hyphens from phone numbers and the inability to chronologically sort time and date categories.

Another way to solve all these problems is to figure out how to have your own program directly read AppleWorks data base files. Those of you who are programmers probably realize that if you could get into the file itself you could manipulate and print out the data any way you wanted. You could even update the data and store a new AppleWorks-format data base file on disk. The possibilities are so immense that readers have been asking me to explain how to do it since I was struggling with volume 1.

I've been reluctant to try, however. There's no straightforward way to read an AppleWorks data base file with Applesoft INPUT or GET commands. This leaves loading the file into memory and reading it with assembly language subroutines or with Applesoft PEEK loops. The problem with assembly language routines is that they take a long time to write and test and they are usually too long to publish. The problem with PEEK loops is that they are excruciatingly slow. For example, consider this little program:

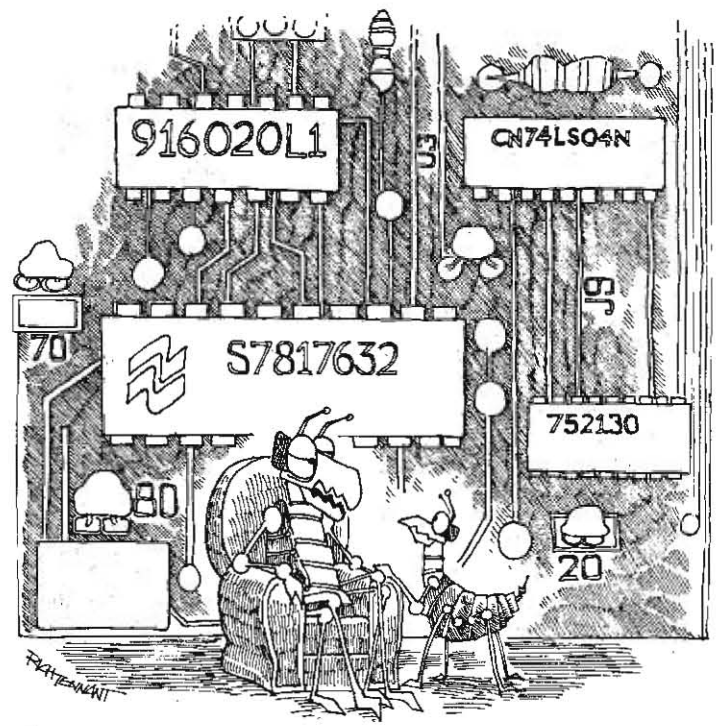
```
10 HGR
20 FOR C = 1 TO 4
30   POKE 8192,C*58 : FOR I = 8193 TO 16383 : POKE I,PEEK I-1 : NEXT
40 NEXT
50 TEXT
```

About all the program does is 32,768 POKEs and 32,767 PEEKs. Because it does them in the memory area devoted to the hi-res screen, with hi-res graphics turned on, it also displays some really ugly patterns. Under standard Applesoft the program takes more than three minutes to execute (190 seconds to be exact). On a IIgs running in fast mode you get a boost factor of 2.7—71 seconds total—still not very fast.

While pondering these problems (after receiving the latest request for information on how to read an AppleWorks data base file), I remembered reading in the thin (but excellent) little manual that comes with the *Beagle Compiler* that compiled programs should use integer values whenever they can, because integers execute much faster than floating point values (the compiler considers an integer to be any whole number between -32767 and 32767, whether stored in an integer variable, such as I%, or not). I wondered whether the compiler would speed up PEEK and POKE loops, which use such integer values. I compiled the above program and tested it. Running on a normal Apple II, the program finished in 19 seconds—10 times faster than normal. On a IIgs running in fast mode, the time was further cut to less than 8 seconds, which is 24 times faster than what we started with a paragraph ago and plenty fast enough to read a data base file.

So this month I'm doing it. Here's how to read an AppleWorks data base file. We'll save writing a new one for a later issue.

File structure. The first thing you need to know is that AppleWorks data base files have three main parts. The first part is called the header. Among the goodies you can dig out of the header are the number of records in the file, the number of categories per record and their names, and the number of report formats that have been defined. There's also a bunch of other stuff there that's important to AppleWorks but of little use to us, such as the format of the single-record and multiple-record screens and the current record selection rules.



"WHY, WHEN I WAS YOUR AGE I HAD TO TRAVEL FIVE MILES OVER A NON-COAXIAL CABLE TO A LITTLE RED PONG MACHINE IN A CORNER NEIGHBORHOOD BAR."

The header itself can be split into two parts. The second part holds the names of the categories. The first part holds everything else. The first part is 357 bytes long. The second part has 22 bytes for each category that has been defined. If all of the 30 possible categories have been assigned names, the second part is 22 * 30 or 660 bytes long, for a total length of 1017 bytes. The header can never be longer than this.

(The only advantage to having data base files with less than 30 defined categories is that you save 22 bytes of file and desktop space per category not defined. The disadvantage, of course, is that should you ever need to add a category to a file, your report formats will be deleted. It's far better, in the world of today's expanded desktops, to always assign 30 categories to new files. Name the ones you have no current use for something like "+", and use open-apple-L(ayout) to move them to a corner of the screen.)

So much for the header. The second section of an AppleWorks data base file holds the report formats. Each report format uses 600 bytes. Thus, the length of this section can vary from nothing at all, if no formats have been defined, to 4,800 bytes if the maximum of eight formats has been defined.

The third section of an AppleWorks data base file holds the actual data records. Records appear in the order in which you last sorted them. (Except that the first record always holds the file's "standard values.") Within each record, categories appear in the order in which they were defined with open-apple-N(ame), which is also the order in which the category names appear at the end of the header section.

Reading a category. The record-category data is all scrunched together. This part of the file is similar to a sequential text file rather than being spread out in equal-length, mostly-empty segments as are random-access text files.

The first two bytes of each record indicate how long the rest of the record is. Next comes the data for the first category. The first byte of each category is a "control byte." When this number is less than 128, it indicates how many bytes of data follow. Let's assume we have a variable called PNTR that points to the current control byte. The following subroutine will dig the data out of that control byte's category and store it in a string variable called C$(N):

```
5100 REM read a single category's data into C$(N)
5112 CBYTE = PEEK(PNTR) : PNTR=PNTR+1  : REM Get control byte.
5162 C$="" : FOR I=PNTR TO PNTR+CBYTE-1 : C$ = C$ + CHR$(PEEK(I)) : NEXT
5190 C$(N)=C$ : PNTR=PNTR+CBYTE      : REM Advance pointer to next category.
5195 RETURN
```

Line 5162 holds what I mean by a "PEEK loop." It begins by clearing a variable called C$, which will temporarily hold the category's data. Then it loops the number of times required to dig the data out of memory. The PEEK(I) part of CHR$(PEEK(I)) tells us what value is at byte I, then the CHR$ part immediately converts that value into an ASCII character.

You may find line 5162's "TO PNTR+CBYTE-1" puzzling. We have to subtract 1 from CBYTE because of the old "indexed from zero" paradox. If PNTR is at byte 100, and CBYTE says there are ten bytes of data, they would be stored in bytes 100 through 109. Looking in bytes 100 through 110 would return eleven bytes of data. A clearer way to write the statement might be FOR I=1 TO CBYTE : C$=C$ + CHR$(PEEK(PNTR+I-1)). Writing it like that would slow down execution, however, because of the additional calculations inside the PEEK statement that would have to be done on each pass through the loop.

Now suppose that the value in CBYTE is *greater than* 128. If CBYTE is 129, it means the next category is blank. If CBYTE is 130, it means the next *two* categories are blank. In other words, CBYTE-128 gives you the number of categories to skip. If CBYTE is 255, it means all remaining categories are blank and you have reached the end of the record. If there are no blank categories at the end of the record, there will still be a 255 marker.

For example, a completely blank record takes up three bytes of space, no matter how many categories there are. The first two bytes indicate the number of bytes in the rest of the record (1—$01 $00 in hex) and the final byte is a 255, indicating all remaining categories are blank. A record with 30 categories, all blank but the last, would begin with two length bytes, then have a control byte of 157 (128 + 29), followed by a control byte indicating the length of the data in category 30, followed by that data, followed by a record-ending control byte holding 255.

Let's add some lines to our previous subroutine to handle blank categories. "NB" is a variable that keeps track of the number of blank categories that should be skipped:

```
5100 REM read a single category's data into C$(N)
5110 IF NB > 0 THEN 5184              : REM In the middle of multiple blanks?
5112 CBYTE = PEEK(PNTR) : PNTR=PNTR+1  : REM Get control byte.
5114 IF CBYTE > 127 THEN 5180          : REM Start multiple blank categoriss.
5160 REM Category contains ASCII-string data.
5162 C$="" : FOR I=PNTR TO PNTR+CBYTE-1 : C$ = C$ + CHR$(PEEK(I)) : NEXT
5164 GOTO 5190

5180 REM Category is blank.
5182 NB=CBYTE-128                     : REM CBYTE-$80 is # of blank categories.
5184 C$(N)="" : NB=NB-1 : GOTO 5195

5190 C$(N)=C$ : PNTR=PNTR+CBYTE      : REM Advance pointer to next category.
5195 RETURN
```

One other detail we probably ought to consider comes up with categories that hold dates or times. AppleWorks uses a special storage format for dates and times to make them easier to sort. The first byte of a date category is 192 ($C0). The first byte of a time category is 212 ($D4). The first byte of any other kind of category is a low-value ASCII character, which will be less than 128.

Date entries consist of six bytes. The first is the ID byte (192 or $C0). The next two hold the year in ASCII characters. The next holds the month, where an ASCII "A" means January, "B" means February, and so on up to "L" for December. The last two bytes hold an ASCII day-of-month.

Time entries consist of four bytes. The first in the ID byte (212 or $D4). The next byte indicates the hour. An ASCII "A" means 00 (the hour after midnight), "B" means 01, and so on up to "X" or 23 (the hour before midnight). By adding the following lines to our previous subroutine, we can add the capability of reading dates.

```
103B DIM MO$(12)  : REM This array is for the names of the months.
1040 MO$ (1)="Jan" : MO$ (2)="Feb" : MO$ (3)="Mar"
1042 MO$ (4)="Apr" : MO$ (5)="May" : MO$ (6)="Jun"
1044 MO$ (7)="Jul" : MO$ (8)="Aug" : MO$ (9)="Sep"
1046 MO$(10)="Oct" : MO$(11)="Nov" : MO$(12)="Dec"

5116 T=PEEK(PNTR) : IF T<128 THEN 5160 : REM Date or time category?
5118 IF T=212 THEN 5130

5120 REM Category contains a date.
5122 YR$ = CHR$(PEEK(PNTR+1)) + CHR$(PEEK(PNTR+2))
5124 MO$ = MO$(PEEK(PNTR+3)-64)
5126 DY$ = CHR$(PEEK(PNTR+4)) + CHR$(PEEK(PNTR+5))
512B C$ = MO$ + " " + DY$ + " " + YR$ : GOTO 5190

5130 REM Category contains a time.
5132 M$ = "AM" : HR = (PEEK(PNTR+1) - 65)
5134 IF HR > 11 THEN M$ = "PM" : IF HR > 12 THEN HR = HR-12
5136 HR$ = STR$(HR) : IF HR < 10 THEN HR$ = "0" + HR$
5138 MI$ = CHR$(PEEK(PNTR+2)) + CHR$(PEEK(PNTR+3))
5140 C$ = HR$ + ":" + MI$ + " " + M$ : GOTO 5190
```

Line 5124 takes advantage of the fact that the ASCII codes for letters are sequential numbers. It turns the "A" that means January, for example, into a "1" by subtracting 64 from the ASCII code for "A" (65 or $41). "B" becomes a "2," and so on. The resulting value (1 to 12) pulls a month abbreviation out of the previously-defined array MO$(N).

Likewise, in line 5132, the letter codes for the hours are turned into numbers between zero and 23 by subtracting 65 from the ASCII letter code. As written, these routines convert dates and times to strings that look exactly like what AppleWorks itself displays. With slight modifications you could arrange dates or times into any alternative format you might prefer.

Reading a record. Now that we have a routine that reads categories, it is a simple matter to write a routine that reads whole records. The following subroutine assumes only that PNTR points at the correct byte of the file and that the number of categories for the file has previously been placed in the variable NC. It also checks for the presence of the end of record marker and jumps to an error routine at line 5900 if it is missing. Since any error is quite likely a program error rather than a file error, the error routine prints some helpful debugging information (this particular program, of course, has been tested thoroughly and worked fine, of course, just before I sent it to the typesetter, of course—I include these lines, of course, in case your own program, of course, based on this one, of course, requires some fine-tuning, of course):

```
5000 REM read record's categories into C$(1)...C$(N)
5010 RL = FN PK2(PNTR) : PNTR = PNTR+2 : REM RL is Record's Length
5030 NB=0                              : REM init # of blanks to 0
5040 FOR N=0 TO NC-1 : GOSUB 5100 : NEXT      : REM get category data
5050 GOSUB 5100 : IF CBYTE <> 255 THEN 5900  : REM get $FF at end of record
5060 RETURN
```

```
5900 REM The file doesn't look right--probably a program, not a file, bug.
5910 HOME : VTAB 10
5920 PRINT "I've encountered an error in the file's structure"
5930 PRINT "   in record ";R;" and category ";N;"."
5940 PRINT
5950 PRINT "The file buffer begins at ";BBG;" and ends at ";BEN;"."
5960 PRINT "   The buffer pointer is at byte ";PNTR;"."
5990 END
```

The "FN PK2(PNTR)" in line 5010 is a function that does a two-byte PEEK. The function is defined earlier in the program but later in this article. For more information on this trick see page 2.35 in our June 1986 issue.

Memory buffers. Everything we've talked about so far assumes that somehow we've loaded at least a part of the AppleWorks data base file into memory and that the variable PNTR points to the proper byte within the file. In order to do this we have to take care to set aside a block of memory we can use as a "buffer" and see to it that Applesoft doesn't accidentally try to use the same memory area. We also want this memory area to lie in an address range less than 32768 ($8000) so that a compiled version of our program can PEEK with integers and run at maximum speed. One good place to put the buffer is between the Applesoft program and its variable tables.

Figure 1 is a picture of how Applesoft uses your Apple's memory. Normally, Applesoft builds the variable tables adjacent to the end of the program image. By proper use of the LOMEM: command, however, we can move the variable tables and create an area of free space between the program image and the tables. This has to be done at the beginning of the program, however, before any variables have been used. (And in order to work with the *Beagle Compiler*, which doesn't update the PRGEND—program end—pointer at bytes 175-176 quite right, it has to be done without referring to PRGEND.) How about:

```
1000 REM Program initialization

1010 LOMEM: 16384 + PEEK(105) + PEEK(106)*256 : REM Create 16384-byte buffer.

1030 DEF FN PK2(ADR) = PEEK(ADR) + PEEK(ADR+1)*256 : REM 2-byte peek function.
1032 DIM CN$(30)  : REM This array is for category names.
1034 DIM C$(30)   : REM This array is for the information in categories.

1050 BBG = FN PK2(105)-16384 : REM BBG points to the beginning of our buffer.
1052 BEN = BBG + 16384/2     : REM BEN points to the end of our buffer.
1054 PNTR = BBG              : REM PNTR points to our position in the buffer.
1056 BYTE=0                  : REM BYTE points to our position in the file.
```

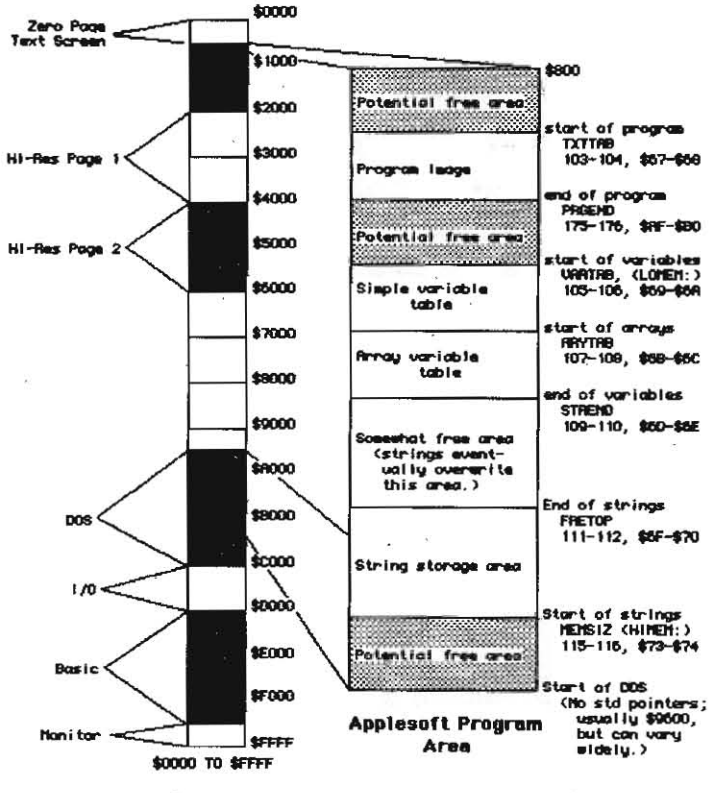


Figure 1

The PEEKs in line 1010 look up the current location of the variable tables; LOMEM: and the addition of 16384 move them $4000 bytes away from the end of the program. Lines 1030-1036 set up the two-byte PEEK function and dimension some arrays we'll need later. Lines 1050 to 1056 set up some pointers to the beginning and end of our buffer and to our position in the buffer and in the file on disk. In line 1052 we actually divide the $4000 bytes we have set aside into two buffers—only one will actually be used in this program, but we'll need the other in a couple of months when we give this program the ability to *write* AppleWorks files, too.

One of the beauties of ProDOS is the ease with which any type of file can be loaded into memory, even in small pieces. The upcoming subroutine for loading sections of AppleWorks data base files uses the BLOAD command, the A(ddress) and L(ength) parameters that DOS 3.3 programmers are familiar with, and the T(ype) and B(yte) parameters that can only be used with ProDOS.

The T(ype) parameter allows any type of file to be BLOADed. BLOAD is not restricted to binary files, as with DOS 3.3. Here, the file type we want to load is "ADB."

The B(yte) parameter allows you to begin loading a file into memory at some position other than the beginning of the file. Setting B to 1000, for example, will cause the BLOAD to skip the first 1000 bytes of the file. This capability is absolutely necessary—without it you can't access files that are larger than the buffer. With it, on the other hand, you can load the file into the buffer in pieces. Here's our subroutine:

```
5500 REM load section of file into buffer
5510 BYTE=BYTE+(PNTR-BBG) : PNTR=BBG
5520 PRINT CHR$(4);"BLOAD";F$;", TADB, L8192, A";BBG;", B";BYTE
5530 RETURN
```

Line 5510 calculates a value for the B(yte) parameter by determining the distance between our pointer, PNTR, and the beginning of the buffer. Then it adds that difference to the previous B(yte) value. When we initially load the first section of the file, these variables will cancel each other out and equal zero, so we'll begin BLOADing at byte zero. Later BLOADs will essentially move the byte that PNTR is pointing at from the end of the buffer to the beginning of the buffer.

Somewhere we need to check to see if PNTR is nearing the end of the buffer. A good place to do this is right after line 5010, where we find out the length of the record we are about to read. We need only add the record length to the PNTR and see if the result is beyond the end of the buffer. If so, we should reload the file. This line will take care of all that:

```
5020 IF PNTR+RL => BEN THEN GOSUB 5500 : REM Does record extend beyond buffer?
```

The header. After we have the beginning of the AppleWorks data base file BLOADed into our memory buffer and before we start reading the actual data embedded in the records, it's necessary to dig a few important pieces of information out of the header. The header length is stored in the first two bytes of the file. This length does not include the two length bytes themselves, however, so we need to add two to the result to arrive at the full header length.

The number of records in the file is a two-byte number and can be found at bytes 36 and 37 (where the first byte of the file is byte 0). The number of categories in the file is stored at byte 35. The number of report formats is stored at byte 38. And the category names are stored in 22-byte segments beginning at byte 357. Each name begins with a length byte and is followed by up to 21 ASCII characters. Here's the instructions for digging all this information out of the header:

```
1100 REM Load first section of file and dig stuff out of the header.
1110 GOSUB 5500 : REM load file

1120 HL = FN PK2(PNTR)+2 : IF HL > 1017 THEN 5900 : REM header length
1122 NR = FN PK2(PNTR+36)                         : REM # of records in file
1124 NC = PEEK(PNTR+35)  : IF NC > 30   THEN 5900 : REM # of categories
1126 NF = PEEK(PNTR+38)  : IF NF > 8    THEN 5900 : REM # of report formats

1140 PNTR=PNTR+357                  : REM get category names
1142 FOR N = 0 to NC-1
1144   C$="" : FOR I=1 TO PEEK(PNTR) : C$ = C$ + CHR$(PEEK(PNTR+I)) : NEXT
1146   CN$(N) = C$ : PNTR = PNTR+22
1148 NEXT
```

If you modify or amplify this program and you are pressed for space, you can make the buffer smaller. Don't make it smaller than 1K (1024 bytes), however, or you might not be able to read the whole header in one chunk. It's also possible, though unlikely, that a single record could hold slightly more

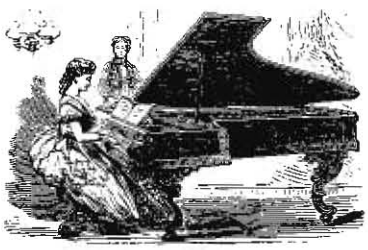
than 1K of data, so a 1.25K buffer is probably a safer minimum size. To make the buffer smaller, change "16384" in lines 1010, 1050, and 1052 and "8192" in line 5520.

After you've finished reading the header, PNTR will point to the first byte after the header. The next section of the file holds report formats, which we want to skip over completely. The following lines will do that:

```
1150 PNTR = PNTR + NF*600 : REM Skip over 600-byte-each report formats.
1152 IF PNTR => 8EN THEN GOSUB 5500 : REM Do formats extend beyond buffer?
```

That pretty much takes care of reading AppleWorks data base files. You now have everything you need to read such a file from an Applesoft program. Here are some additional program lines, however, that dress-up this demo a little so that you can easily read any AppleWorks data base file without actually running AppleWorks.

```
CAUTION: THIS PORTION OF THE PROGRAM DOESN'T INCLUDE THE PROGRAM LINES
   EMBEDDED IN THE ACCOMPANYING ARTICLE, WHICH ALSO MUST BE TYPED IN
   TO MAKE THE PROGRAM RUN. THE ORDER IN WHICH YOU TYPE THE LINES
   MAKES NO DIFFERENCE, JUST DON'T SKIP ANY. DON'T ENTER LINES 10 THROUGH 50.

 100 REM *** Open-Apple's ADB Reader ***
 101 REM   by Tom Weishaar, Feb 1987

1020 CLEAR : REM Restart point for reading another file.
1030 DIM TB(30,2) : REM This array is for category TAB positions on screen.

1060 PRINT CHR$(4);"PR#3" : PRINT : REM 80-column screen required.
1062 VTAB 10
1064 PRINT "What is the name of the AppleWorks database file you want to see?"
1066 PRINT
1068 INPUT F$ : HOME
1070 IF F$="" THEN END

1130 FOR N = 0 TO NC-1            : REM Get screen positions.
1132   TB(N,0) = PEEK(PNTR+186+N) : REM screen sequence left-top to right-bot
1134   TB(N,1) = PEEK(PNTR+114+N) : REM   horizontal screen position
1136   TB(N,2) = PEEK(PNTR+150+N) : REM   vertical screen position
1138 NEXT
1160 REM Just for fun, draw an AppleWorks-like screen for display.
1162 PRINT "File: ";RIGHT$(F$,20) : PRINT : PRINT "Selection: All records"
1164 VTAB 1 : POKE 1403,28 : PRINT "Open-Apple's ADB.READER"
1166 VTAB  7 : FOR I=1 TO 8 : PRINT "==========";  : NEXT
1168 VTAB 23 : FOR I=1 TO 8 : PRINT "----------";  : NEXT
1170 PRINT "Press spacebar to see next record.";

1200 REM Get records and display them on screen.
1210 FOR R=0 TO NR
1220    GOSUB 5000 : REM this loads C$(N) with record's data
1230 REM The rest of this just displays the data on the screen.
1232      VTAB 6 : POKE 1403,0
1234      IF R=0 THEN PRINT "Standard Values for this file:" : GOTO 1238
1236      PRINT "Record ";R;" of ";NR;":"; CHR$(29) : REM chr$(29) clears line
1238      FOR N = 0 TO NC-1
1240        VTAB TB(N,2)+1 : POKE 1403,TB(N,1)-1
1242        PRINT CN$(TB(N,0)-1);": ";C$(TB(N,0)-1); CHR$(29)
1250      NEXT
1260      VTAB 24 : POKE 1403,37 : GET A$ : PRINT A$; : REM wait for key
1270 NEXT

1300 REM End game.
1310 HOME : VTAB 12
1320 PRINT "That's all the records in ";F$
1330 PRINT
1340 PRINT "Would you like to see another? (Y/N) "; : GET A$ : PRINT A$ : HOME
1350 IF A$="Y" OR A$="y" THEN 1020
1360 END
```



Miscellanea

ProDOS 8, version 1.3, released to developers in mid-January and mentioned here last month, mistakenly includes a BRA (branch always) machine language instruction in a critical piece of code. This is a significant problem, because BRA isn't supported by the original 6502 microprocessor. Consequently, this version of ProDOS causes bad things to happen when run on Apple II-Pluses and unenhanced IIes. If you BLOAD PRODOS, TSYS, A$2000, then the bad instruction appears at $4CCD. A BEQ (branch on equal) would work just as well here, so you can fix things with a POKE 19661,240 and a BSAVE PRODOS, TSYS, A$2000.

Another significant problem I've encountered while running the newer versions of ProDOS on earlier machines has to do with interrupts. ProDOS 1.1.1 and earlier versions disabled interrupts. An alien static- or bad-luck-caused interrupt signal caused no problem with these versions of ProDOS unless some software that used interrupts had enabled them. Newer versions of ProDOS, on the other hand, leave interrupts enabled at all times in order to support some new features of the IIgs. If you use these versions on an earlier computer, an alien interrupt will lock it up with the message "INSERT SYSTEM DISK AND RESTART" at the bottom of the screen. At least that's my diagnosis of why I've seen that maddening message more in the last month than in all my previous incarnations. I've reverted to using ProDOS 1.1.1, patched as described in our November issue, on everything around here except the IIgs. I suggest you do so as well until further notice.

The problems that Apple's SCSI card had with the IIgs have been solved. There is a revision B EPROM now available for the card (part # 341-0112-B); contact your dealer for details.

You might be surprised at the kind of things you can get on continuous forms. The advantage of continuous forms, of course, is that they are easy to process through any printer that has a tractor feed, which is just about any printer nowadays. Besides the continuous credit card forms mentioned in this month's main article, you can also get such things as mailing labels, envelopes, invoices, purchase orders, statements, letterhead stationery, checks, and even Rolodex cards on continuous forms. Spend a few minutes wandering around your local office supply store to see what they have that you could use. Wouldn't life be easier if you designed a "labels style" report format for AppleWorks that could print your address-phone number data base on Rolodex cards?

If you live in the U.S., read this and act quickly. U.S. federal tax forms for 1986 are available on AppleWorks spreadsheet templates for $23.95 from Personal Financial Services, P.O. Box 1401, Melville, NY 11747 516-261-8652; for $42.45 from Sky Computer Resources, P.O. Box 204, Portland OR 97207 503-234-7291; or for $50 from Island Computer Services, 3501 E Yacht Dr, Long Beach, NC 28461 800-826-7146. All three packages include the main 1040 form, as well as the forms for Employee Expenses, Depreciation, Itemized Deductions, Interest and Dividend Income, Profit from a Business or Profession, Capital Gains, Supplemental Income, Self-Employment Tax, and the Married Couple Deduction. All three packages produce IRS acceptable print-outs for all forms except those that are color-coded (the IRS does want you to use green-bar or lined paper, however).

Personal's package, called *1040Works*, also includes forms for Farm Income, Income Averaging, Credit for the Elderly, Sale of Residence, Child Care Expenses, Moving Expenses, 10-Year Income Averaging, and Alternative Minimum Tax. If you have enough memory for a 256K AppleWorks desktop, pay $3 more and ask for *1040Works-X*; you'll get all this stuff in one large spreadsheet.

Sky's disk also includes form 1040A and the schedule for Farm Income. Sky can provide any other form or schedule printed by the IRS for an additional $5 each.

Island's disk also includes the forms for Child Care, Moving Expenses, Income Averaging, and Credit for the Elderly, as well as tax planning templates for 1987. Island is the only company of the three that accepts credit cards.

If you don't live in the U.S., (or Canada, Mexico, Australia, or New Zealand) read this. Because of a bad scale at our mailer's, our December issue, mailed near the end of November, went out with only enough airmail postage for 1/2 ounce. Since the newsletter actually weighed slightly more than that, the U.S. Postal Service seems to have kindly forwarded that issue to you by surface mail. Except for that issue, we have *paid for* 1 ounce of airmail on each of your newsletters each month. If you have received any other issues by surface mail, or if you receive an issue by surface mail in the future, please send the empty envelope back to Sally Dwyer at our Overland Park address so she can figure out where our intentions have gone astray.

Ask (or tell) Uncle DOS

Man talks, Apple listens

I was sick of reading how wonderful your newsletter is in all the computer magazines.

I was sick of getting little notices in all my new Beagle Brothers software telling me you were still alive.

I was sick of getting postcards in the mail asking me for a good reason not to subscribe to ***Open-Apple***.

So I took you up on your offer of a free issue. And now, damn it, I'm a member of the insidious "Gee, one suggestion alone was worth the entire cost of the subscription" club.

Ok, enough of that. Let me tell you what I'm doing with my Apple IIe. At the very least, I'd like to learn what others might be doing in a similar vein.

I'm the statistician for a group of baseball fanatics in what we call the Duckball League. (About the name, well, it's a long story.) Anyway, we draft major league players onto our teams and compete with each other in a number of statistical categories. And since I'm the one with the computer, I'm the statistician. Funny how things work out that way.

I use AppleWorks and about a 74K spreadsheet to keep track of everything. Normally, it takes me about two hours a week to enter the data on the almost 300 players in the league. The most difficult part is looking back and forth from the tiny type in USA TODAY (our bible), to the keyboard, to the screen, and back again. I figured I could speed things up by eliminating one of those three elements. Enter voice recognition.

After checking the few manufacturers of Apple voice recognition systems I could find, I decided on Intravoice II from the Voice Connection in Irvine, California. It consists of a main plug-in circuit board, a microphone, and a couple of smaller boards into which you plug several of the Apple IIe's ICs. All in all it's pretty easy to install.

The voice input module itself is quite easy to use and can be trained to recognize just about any word. That word can then be used to as a substitute for any keystroke or string. It supports open- and solid-apple commands as well, so it's ideal for my use in entering Duckball stats into the AppleWorks spreadsheet.

I have encountered one and only one problem in my entire use of Intravoice II to date. When using a phrase ("PinPoint") to substitute for a solid-apple-P keystroke to invoke the PinPoint Desktop Accessories, AppleWorks crashes into the monitor. I'm talking dead. Once I'm in PinPoint, voiced solid-apple commands work fine. But somehow when AppleWorks is active it doesn't work.

Rex Creekmur
Grand Rapids, Mich.

Your letter leaves me speechless. The address of The Voice Connection is 17835 Sky Park Circle, Suite C, Irvine, CA 92714, (714) 261-2366.

The elusiveness of relative power

In *Compute!'s Apple Applications* Vol. 4 (Fall/Winter 1986), William Mensch, the one-man 65816 progenitor, states that the 65816 (and presumably the 6502) runs in such a manner that one of its bus cycles is equal to four of anybody else's (page 18). He states that a 65816 running at 6 megahertz is "equivalent to the IBM PC running at 24 megahertz" and "a 2- to 3- megahertz Apple has the same kind of performance as an 8 megahertz Macintosh."

This information, if correct, is of critical importance to a typical hobbyist, such as myself, ready for his next generation computer. The marketing power of this alleged fact is exponential. Many times in the last three years I have pondered to myself something like, "My IIe is a wonderful machine but I wish I had the speed and power of the 8088".

It seems to me that the 6502's longevity and the allegiance it commands may be due as much to its unsung power and speed as to marketing forces over the past ten years. Maybe it's not ancient and obsolete but instead was ahead of its time when created and is still incredibly capable in its present form. If this conjecture has some credibility, then it follows that even though the 65816 is panned by blue loyalists as a 16-bit introduction at the dawn of a 32-bit era, the brute-force power and sophisticated characteristics may well approximate, parallel, or even outshine the favored sons of Mac and Charlie. Regardless, the technology for building a San Francisco skyscraper is quite different than that for building the Golden Gate Bridge. Both are indispensable.

Steve Cranney
Fallon, N.Y.

*Microprocessors are complex devices. Just as you can control the superiority of quarter-horses over Thoroughbreds by shortening the race, microcomputer loyalists can devise benchmarks that show their favorite microprocessor is the "most powerful." That's why the benchmarks we published here (December 1986, page 2.88) were the standard benchmarks **Byte** has been using for years—we didn't want to be accused of favoritism. Yet that's not to say we would have been so eager to publish the results if the 6502 and its progeny hadn't come out looking so good.*

*Dennis tells me Mensch's comments (great interview, by the way, congratulations to **Compute!**) are based on the fact that the 65xxx series uses a scheme called "pipelining" that allows it to grab the next piece of data it will need while it's still working on the last piece and on the fact that all instruction codes in the 65xxx series are just one byte long. This places some limits on the number of different instructions but allows faster execution.*

*The general philosophy of the 65xxx series is that the processor spends less time finding out what it is supposed to do and more time actually doing it. This translates into faster execution of common tasks at the expense of the inclusion of more powerful commands, such as multiply or divide instructions. This same philosophy has been used to develop a new breed of "reduced instruction set" computers that actually execute only a few instructions but do so **very** quickly. The IBM PC-RT is an example of this kind of machine.*

*In a practical example, the 65816 is faster than a 68000 (at the same clock rate) for simple load accumulator and save accumulator operations, which would favor the 65816 in a benchmark based on that ability. The 68000 should kill the 65816 in a math benchmark, however, because the 68000 has math instructions that the simpler 65816 instruction set does not. Nonetheless, if a benchmark **doesn't use** the 68000's math operations and instead does math "manually," as the 65816 does, then the comparison would favor the 65816. I suspect this is essentially what happened with the **Byte** benchmarks we reported.*

Dennis recommends you take a look at the series of books Adam Osborne has written on microcomputers for further information.

In the final analysis, the most important thing to a hobbyist shouldn't be a microprocessor's power, anyhow. You should be looking for a computer with a good, reliable, overall design, with good software that does the kinds of things you want to do, and with good documentation (very important, but often overlooked). An example of the importance of documentation is the first commercially available 16-bit microcomputer, the Texas Instruments 99/4. TI wanted to develop all the software for the machine itself and locked hackers out. No documentation, no hacking; no hacking, no programs; no programs, no customers. What was the advantage of owning the most "powerful" microcomputer of its day if you couldn't figure out how to do anything interesting with it?

RGB and TV too

I have a Sony KV-1311CR monitor hooked to my Apple IIgs. It's a 13-inch, cable-ready, remote-control TV, with analog and digital RGB inputs and video inputs and outputs. It has a dandy picture, both in normal TV use, and when used with the GS analog output. KV-1131s can be had mail order for less than the Apple monitor (the latest price I've seen is $399). I also like it because I can hook up the GS to the analog input, a II-Plus to the video input, and cable TV to that input, and have them all there for use in one neat little package.

The pinouts in the Sony book are as if you are looking at the mating cable connector, not at the jack on the TV. On the monitor, pin 1 is at the bottom, towards the back. On the IIgs, pin 1 is on the top, towards the on-off switch. Incidentally, the numbers don't match the pin numbers molded on DB-15s, or at least not on the one I used. Here's how to run the wires:

Sony Pin	GS Pin	Function
8	1	Red Ground
25	2	Red Signal
26	5	Green Signal
10	6	Green Ground
27	9	Blue Signal
12	13	Blue Ground
30	3	Composite Sync

You can also enable the IIgs audio output, but this is less straightforward. The Sony requires +2.5 to 5 volts on pin 34 to enable the audio input. Otherwise, you'll get TV audio all the time. Apple neglected to put +5 volts on its jack. However, there is +12 volts. You can have a technician build you a voltage divider out of a couple of resistors to cut back the +12 volts, which comes out of pin 8 on the IIgs. The audio signal

comes out of pin 11 on the IIgs and goes into pin 24 on the Sony.

An alternate option is to use the voltage divider to automatically switch on the video RGB when you turn on your computer. To do this, hook the +5 volts to pin 33 on the Sony.

Chris Arndt

*So, you **can** get a "free TV" by carefully selecting a IIgs monitor. The March 1987 issue of **Consumer Reports** compares 175 color television sets from 21 different companies; nine of these sets reportedly have RGB inputs. The nine are J.C. Penney model 2220, Magnavox models RF4254WA and RG4378BK, Quasar models TT6290XE and TT6298YW, Sanyo models AVM210 and 12C700, Sears model 42701, and Sony model KV-20XBR. There are, no doubt, others—the set you have, for example, isn't among the 175 sets listed.*

Slashed zeros and 8 bits

I have spent many hours trying to get my Imagewriter II to print slashed zeros. I've tried using the control codes given in the manual; it works fine except that I get a double-spaced printout.

As I read the printer manual, the printer codes required for slashed zeros are:

```
Slash on:  ESC D Control-@ Control-A
Slash off: ESC Z Control-@ Control-A
```

What am I doing wrong?

Richard E. Breininger
FPO Miami, Fla.

ESC D and ESC Z allow you to change the Imagewriter dipswitch settings under software control. ESC D is used to turn switches on. ESC Z is used to turn switches off. There are two eight-switch groups of switches, called A and B, for a total of 16 switches.

*After the ESC D or ESC Z the Imagewriter expects to see 16 bits of data that tell it **which** switches to turn on or turn off. Apple's sequence for turning on the dipswitch for slashed zeros is (hex values) "1B 44 00 01". This causes switch 1 on dipswitch B to be turned on. This tells the Imagewriter to use slashed zeros and slashed zeros we have.*

Unfortunately, however, Applesoft takes the liberty of setting the high bit of each byte printed (for a complete discussion of this problem see "Control-D(efeated)" in our December 1986 issue, pages 2.84-86 and "A bit too many" in April 1986, page 2.24). Thus, the actual values sent are "9B C4 80 81." Consequently, we actually turn on two additional switches—switch B-8, which is not used by the printer (no problem) and switch A-8, which adds a linefeed after carriage return if it is on (Presto! Instant double-spacing).

One answer is not to send the command sequence through Applesoft—see our December issue for the details. In this case, however, we don't have to get so elaborate. We can just turn the linefeed option back off after turning slashed zeros on. In other words, after sending the codes ESC D control-@ control-A, also send ESC Z control-@ control-@. Your printer will see "9B C4 80 81 9B DA 80 80" and will begin giving you slashed zeros without extra linefeeds.

Here's a good tip for those of you having problems similar to this one—it comes from a letter in our May 1985 issue, page 39. Turn your Imagewriter off, press and hold down on the linefeed button, and turn it back on. It will now print the hexadecimal code for each character it receives instead of doing normal printing. This feature can be quite useful for diagnosing printer command-code problems.

PRINT TAB alternatives

Tell me how to get a PRINT TAB statement to look like it does on your Apple IIc screen when you're printing to an Imagewriter II. Help—I'm a desperate woman!

Robin Boscia
Pittsburg, Penn.

I have a problem with an Imagewriter II, Super Serial Card, and enhanced IIe: my Applesoft program has tabs and the printer will not go to the correct columns. Tabs range from column 1 to column 103. Can you help?

David E. Brewer
Cincinnati, Ohio

Amazing. I've spent all morning playing with PRINT TAB on an enhanced IIe with a Super Serial Card and it hardly does anything right. After trying all kinds of combinations, including printing with the screen display in 40 and 80 columns and printing with video echo on and off (control-I I), I finally found a combination that worked: before turning on your printer, PRINT CHR$(21) to the screen to turn the Apple 80-column firmware off. Then, after turning the printer on, PRINT CHR$(9);"T E" (space required between T and E) to enable the Super Serial Card's "Basic Tab" command, which I don't remember ever hearing of before this morning's search through instruction manuals. That, anyhow, made this program work (using ProDOS):

```
10 HOME : PRINT CHR$(21) : REM Turn off 80-columns
20 PRINT CHR$(4);"PR#1"
25 PRINT CHR$(9);"T E" : REM Turn on ''Basic Tabs''
30 FOR I=1 TO 8 : PRINT SPC(9);T; : NEXT : PRINT
40 FOR I=1 TO 8 : PRINT "1234567890"; : NEXT : PRINT
50 FOR I=1 TO 4
60   PRINT TAB(30);"ROW";TAB(40);"OWR";TAB(50);"WRO"
70 NEXT
80 PRINT CHR$(4);"PR#3"
82 END
```

A significant problem with this trick is that the Apple IIc serial port command set doesn't include the "control-I T E" command.

Using PRINT TAB while both the 80-column firmware and the printer are turned on doesn't work at all. This is because something, probably the firmware but I'm not sure what, forces the values in CH (byte 36 or $24) and OURCH (byte 1403 or $57B) to zero. CH holds the horizontal cursor position for 40-column firmware and OURCH the position under 80-columns.

But Applesoft expects these locations to also reflect the horizontal position of the printer. Yet when both the printer and 80-column mode are on, both bytes always hold zeros (that's why an initial PRINT TAB works fine but later ones work just like PRINT SPC). Any readers who can provide more information on what the bug is or how to make TAB work right are encouraged to write.

Meanwhile, here are two ways to do tabbing without using PRINT TAB. Let's say you want to print a table in three columns. Assume the widths of the columns are stored in the variable array CW(c). The stuff you want to print in the columns is stored in the variable array T$(r,c). In this case, try the following:

```
10 FOR R=1 TO NR   : REM For row 1 through # of rows
20   FOR C=1 TO NC : REM For col 1 through # of cols
30     C$=LEFT$(T$(R,C),CW(C)) : REM truncate
40     PRINT C$; SPC(CW(C)-LEN(C$));
50   NEXT : PRINT
60 NEXT
```

In line 40 we print the data, then use the PRINT SPC function to fill out the column with spaces. We determine how many spaces to print by subtracting the length of the string just printed from the column's width. To use this trick we have to make sure the data is never wider than the column. If it is, the subtraction will produce a negative number and the program will stop with an ILLEGAL QUANTITY ERROR in the SPC function. That's why line 30 truncates everything to the width of the column. The biggest disadvantage of this method is that all these string manipulations will slow your program down—you may find your printer is waiting on the computer rather than the other way around.

The second alternative is to use your printer's tabs. This presents two problems. First, you have to figure out how to set the tab stops. Then you have to figure out how to get the control-I character, which most printers use for a TAB command, through your interface card and out to the printer.

*For the Imagewriter, the command sequence "ESC (" tells the printer you are sending a list of tab stops. This list comes immediately after the command and is a sequence of three-digit numbers, separated by commas and terminated with a period. The numbers will be sent as ASCII characters, so we don't have the high bit problem mentioned in the last letter. To set tabs at columns 30, 40 and 50, for example, use PRINT CHR$(27);"(030,040,050." (If you're having trouble with the conversions of things such as ESC into things such as CHR$(27), just memorize the "ASCII Control Code Rosetta Stone" on page 85 of the November 1985 **Open-Apple**.)*

The Imagewriter also allows you to clear a list of tab stops—just use "ESC)" instead of "ESC (". You can clear all tab stops with "ESC 0". When setting tabs, the left-most print position is tab stop 1. The maximum permitted setting depends on the width of the character set you are using. Once you have set a tab stop, however, it remains in the same absolute position on the page, (at least with the Imagewriter) even if you change character widths or the position of the left margin.

There are two ways to sneak a control-I through an interface card. You have to sneak them through because most cards use control-I as a "wake-up, here comes an interface card command" code. Consequently the cards eat any control-Is they see rather than passing them on to the printer. Some cards, including the Super Serial Card, will pass a single control-I on to your printer if you print two of them in succession. Otherwise, you have to change the card's command code to something other than control-I. Do this by sending a control-I followed by any other control-character. If you do this, however, change the command code back to control-I before you turn the printer off.

Try this program to see how an Imagewriter handles tabs with your interface card:

```
10 PRINT CHR$(4);"PR#1"
15 PRINT CHR$(27);"0"; : REM clear all tab stops
20 PRINT CHR$(27);"(030,040,050.";
30 FOR I=1 TO 8 : PRINT SPC(9);I; : NEXT : PRINT
40 FOR I=1 TO 8 : PRINT "1234567890"; : NEXT : PRINT

50 TB$=CHR$(9) : Z$=CHR$(26)
52 PRINT TB$;Z$ : REM change cmd code to ctrl-Z
54 GOSUB 90
56 PRINT CHR$(27);"L020";"Left margin at col 20."
58 GOSUB 90
60 PRINT CHR$(27);"L000";"Left margin at col 0.
62 PRINT CHR$(27);"P";"Proportional type"
64 GOSUB 90
```

```
66 PRINT CHR$(27);"Q";"17 chars per inch."
68 GOSUB 90
70 PRINT Z$;TB$ : PRINT "TAB with two ctrl-Is."

72 TB$=TB$+TB$
74 GOSUB 90

80 PRINT CHR$(4);"PR#3"
82 END

90 FOR I=1 TO 4
92   PRINT TB$;"Nice";TB$;"columns,";TB$;"right?"
94 NEXT : PRINT : RETURN
```

GET out of text files

I have been working on a program that creates downloadable characters for my Imagewriter printer. I had it save the characters in a text file. When I tried to read the data back I discovered something unusual. My read routine was similar to this:

```
10 GET A$
20 A = ASC (A$)
30 GOTO 10
```

If you type this in and run it, it works fine most of the time. The problem arises when you try to read a $00 (or $80 as it is stored in the file). If you try control-@ in the above program you get an ILLEGAL QUANTITY ERROR in line 20. Any idea why? Can I make it quit doing that?

Steve Carder
Liberty, Mo.

GET will not read an ASCII $00 or $80 (control-@) character; it interprets it as a null string. The GET does return to your program, however, so here's how to detect an ASCII $00:

```
10 GET A$
15 IF LEN(A$) = 0 THEN A = 0 : GOTO 30
20 A = ASC (A$)
30 GOTO 10
```

Nonetheless, a text file is probably not the best choice for saving data like this—text files always set the high bit. Try POKEing your characters into a buffer and BSAVEing them. Or, if you insist on a text file, convert the characters to ASCII numbers while saving them with PRINT STR$(A) and read them back with INPUT A$: A=VAL(A$). This will probably quadruple the size of your file, however.

Multiplan to DIF

In regard to the question in your November 1986 issue (page 2.79) about converting *Multiplan* files to *SuperCalc* files, a recent issue of *Computist* includes a program by D.W. Walkey that converts *Multiplan's* SYLK files to DIF files (issue #37, page 20). I've used the program to move several spreadsheets to AppleWorks.

J.D. Holdeman
N. Ridgeville, Ohio

***Computist's** address is P.O. Box 110846, Tacoma, WA 98411. DIF files, of course, will move only the **values** from one spreadsheet to another, not the underlying formulas.*

Editing data files

In your November 1986 issue (page 2.77), Lawrence Pratt mentions that he edits programs and data files, even random access files, with *Apple Writer*. Can you give more information about how this is done? Is this a better method than using GPLE or simply a different way of doing the same thing?

Thomas E. Militello
Rancho Palos Verdes, Calif.

*Everyone has his or her favorite method of writing programs. **GPLE** and similar programs are "line-oriented" editors—you edit one line at a time. Many of us who are addicted to "screen-oriented" word processors, however, keep what we know about line-oriented editors and ox-drawn vehicles on the same index card.*

*To use a word processor to write programs, simply make sure you enter a return at the end of each line. Save the program in a text file (from AppleWorks, open-apple-P(rint) the program to "A text (ASCII) file on disk.") Exit your word processor, enter Applesoft, and EXEC the text file you have written. Uncle DOS will type your program in for you. For more on this, including how to get programs you have already written into text files you can edit, see the May 1985 **Open-Apple**, page 36.*

*If you happen to use **Apple Writer** for program editing, CondiCom's program **OpenAppleWriter** is very handy (December, page 2.87). Another possibility is **Program Writer** from the Software Touch (September, page 2.62c), which is a full-powered, screen-oriented program editor.*

*Using a word processor to edit data files is similar to editing program files. In order to edit random-access files, however, the file should be completely filled with blanks or other characters before being used the first time. For much more information on this subject, see my column in the May 1984 **Softalk**, page 164.*

Only exit FOR-NEXT at NEXT

The insertion sorts printed in your November 1986 issue (page 2.80) had a common erroneous method of exiting a FOR—NEXT loop.

It should read:

```
  first routine:
940 IF A(E) <= T THEN E=E1 : GOTO 960


  second routine:
950 IF A(K,D%(E)) <= A(K,T) THEN E=E1 : GOTO 965
```

Before you exit a FOR—NEXT loop, you need to close out Applesoft's handling of that loop to free up the stack. The amended lines will set the loops' index variables to their maximum values and GOTO the loops' NEXT statements, so that Applesoft will close out the stack handling for the FOR—NEXT before continuing.

Craig Willford
Whittier, Calif.

*You are correct. I talked about this problem in the April 1984 **Softalk**, page 52. Using GOTO from inside a FOR—NEXT loop to a line outside the loop leaves Applesoft expecting a NEXT that never comes. Exit like this from eleven loops with different index variables in a row and you'll get an OUT OF MEMORY error. Applesoft won't say so, but it means it's out of **stack** memory.*

As a practical matter, many programmers use the same index variable for all loops, so they rarely see the bug. Applesoft fixes the stack automatically when a loop, or any other loop that uses the same index variable (E in this case), is re-executed. Nonetheless, there is a limited amount of stack memory in the Apple II (256 bytes), consequently programmers should try to keep it free of muck such as unresolved FOR—NEXT loops.

More TransWarp experiences

I read with interest the letter from Pat Marnett in your January issue, page 2.96, with reference to problems with a TransWarp board. I used an Ace 1000 with a TransWarp and Peachtree's *Back to Basics* accounting package with no problems. Recently, I upgraded(?) to the Ace 2100. Upgrading my *Back to Basics* program to the IIe configuration and using the TransWarp, I lose all data on the second drive. Removing the TransWarp, everything is perfect. Thank the Lord I copy data regularly! I've called AE before about the TransWarp and came away feeling like a stupid ass from the condescending attitude. I don't think the ProDOS patch would work because *Back to Basics* is a DOS 3.3 program, so my TransWarp is sitting on the shelf.

Frank Drew
Seminole, Fla.

I have been reading your letters about ProDOS zapping disks. Although I never experienced that, I have experienced three zaps of some significance to me and I wonder whether you have heard of similar complaints.

Briefly, I have zapped three disks when I tried to disable my TransWarp, using the escape key, after power-up. The three disks were all copy-protected and non-ProDOS—the reading program *Smart Eyes*, a *Moebius* scenario disk, and a program called *MicroTest* from Harper and Row that I use in my sociology class.

My computer has PinPoint's IIe upgrade kit and a Checkmate 768K memory board.

Larry Davis
Bedford, Texas

We've received a couple of other letters reporting problems with TransWarps since publishing Marnett's letter. My own IIe even started acting flakey last week (spreadsheet cells in columns to the right of my work area having nonsense formulas in them, characters I hadn't entered appearing in unusual places on the screen, computer locking up) and all the problems went away when I replaced my TransWarp with an older SpeedDemon.

The big problem, of course, is that once you're used to working with a TransWarp, it is very difficult to go back to something that's slower. Another difficulty is that the problems come and go—Dennis put my TransWarp in his IIe and hasn't had a bit of trouble.

I suspect the root problem is that the TransWarp pushes the IIe hardware to its limits. My guess is that weak chips or noise in the computer, rather than in the TransWarp, are causing most of these problems. Another potential culprit is dirty contacts on the TransWarp's edge connector. These are just guesses, however.

RAM, compiler comparisons

I recently called Applied Engineering and ordered the latest version of their desktop expander software so I could compare it with the software that came with my Checkmate MultiRAM CX card (for the Apple IIc). As far as I can tell there does not seem to be any problem running Applied Engineering's $10 software with Checkmate's card, except for the AppleWorks Desktop Expander itself, which doesn't work.

Other comparisons between the two disks have left me favoring Checkmate's $5 software. Checkmate's RAMdisk, called /MRAM, is configured for slot 3 drive 1 and leaves /RAM in slot 3 drive 2 intact. AE's RAMdisk, on the other hand, expands the the size of /RAM in slot 3 drive 2. I prefer Checkmate's way of creating the RAMdisk with a SYS file rather than a file that must be run from an Applesoft program as with AE's software.

AE's disk did have a nice program that uses auxiliary memory for a a one-pass disk copy (5.25 disks only). I don't see any advantage to using it, though — since it must first format the disk it is going to use it isn't any faster than *Copy II Plus*. Both disks have very similar auto-load programs for moving files to the RAMdisk. I did not compare the two except to notice that both seem very easy to customize. I always use *MouseDesk*'s auto-load feature so I don't need another auto-load program.

I have two compiler bugs to report. Interestingly enough, they were both discovered while using two different compilers on the same program and both problems were similar enough to affect the same program line.

The first one is in the *Beagle Compiler* (version 1.0) and is illustrated in the following example:

```
10 D = 23: D$ = STR$(D): DX = VAL(D$): PRINT DX
```

The above statement will print 23 with Applesoft and print 0 when compiled. The other bug is in *Micol BASIC* (version 2.2) and can be demonstrated as follows:

```
10 D$ = " 9": D% = VAL(D$): PRINT D%
```

This statement will print 9 with Applesoft and print 1024 with *Micol BASIC*.

I've sent letters to each company and I would expect they will implement fixes shortly. The *Micol BASIC* bug can be worked around as follows:

```
10 D$ = " 9": D = VAL(D$): D% = D: PRINT D%
```

Anyone who uses the *PinPoint Desktop Accessories* may want to remove some accessories from the list or rearrange them. This can be done by loading the text file PINPOINTPROFILE from the install disk; then arrange the file the way you want it; then save the file back to the disk and run the install program. You may even be able to install your own favorite binary programs (perhaps a game) as accessories by adding it to the list along with its length and highest memory address.

David Stevens
Eden Prairie, Minn.

Our experience and a growing pile of letters from our subscribers indicate Checkmate also has much better technical support.

The ***Beagle Compiler*** *is up to version 2.0. In addition to a fix for the STR$ bug, the latest version lets you automatically store strings and arrays outside of main memory. This gives you more room for programs and much more room for data. The extra memory supported includes the second 64K bank of a 128K IIe, a IIc, or a IIgs; auxiliary slot RAM cards from Applied Engineering and Checkmate; and slot 1-7 memory cards from Apple, Applied Engineering, and Cirtech.*

Memory, Apple Writer gotchas

In your discussion of the various kinds of RAM in the December 1986 issue, you said "Interestingly, none of these switches (HIRES/LORES, STORE40/STORE80, PAGE1/PAGE2) actually do anything to the current display that appears on your monitor." Well, that's almost true. If text or graphics page 1 is already being displayed, then turning on STORE80 will not change the display. But if page 2 is being displayed, turning on STORE80 will change the display to page 1 as the PAGE1/PAGE2 switch takes on its new meaning.

On the older unenhanced Apple IIe, this little interaction can cause a minor display annoyance in certain programs — specifically programs that display text on hi-res graphics page 2 and rely on monitor routines to help with the text bookkeeping. This is because a few monitor routines in the unenhanced IIe, including the scroll routine, turn on STORE80 for a brief moment even with 40-column operations. While STORE80 is on, the display changes to graphics page 1, causing a brief "flicker" on the screen. The enhanced IIe and IIc do not access STORE80 during 40-column screen operations and so do not have this problem.

You suggest that the "best" use of the auxiliary bank 64K of memory or an auxiliary slot memory board is as a RAMdisk. Do you mean from an application programmer's point of view or a user's? As a user, I find both the built-in ProDOS RAMdisk and the third-party auxiliary-slot memory board RAMdisks less than convenient to use. Sometimes the most convenient way, or in some cases the only way, to get from one applications program to another is to press control/open-apple/reset to reboot. But the contents of these auxiliary memory RAMdisks are lost whenever you reboot. You have to be careful to use any Quit options provided when moving from program to program.

In addition, for the auxiliary-slot memory boards, you have to run a special program every time you start up in order to "install" the RAMdisk driver code into memory. I've talked to too many customers who believe that they should be able to cold-boot any ProDOS program and have that program automatically recognize their auxiliary slot memory board as a RAMdisk. Since their memory board works so well with AppleWorks, they believe there's a bug in any program that can't do that.

You mention in the article that *Apple Writer* doesn't follow the documented protocol for disconnecting the auxiliary memory RAMdisk. Here's another interesting and obscure little tidbit — *Apple Writer* also diddles in silly ways with the Super Serial Card.

The printing set-up file that comes with *Apple Writer* includes the setting "CR1", which means *Apple Writer* will supply its own linefeed character after each carriage return. But the printer cards on most Apples are already set to supply their own linefeed after carriage return. They need to be set that way for printing from Applesoft to work.

So how does *Apple Writer* avoid double spacing? It first checks if the printer interface is either a Super Serial Card or a IIc printer port. If it is, *Apple Writer* pokes new values directly into the screenholes reserved for the printer interface to turn off the interface card's linefeed. Now *Apple Writer* can add its own linefeeds, and every SSC or IIc owner is happy.

Well, almost... *Apple Writer* doesn't repoke the old values back into the screenholes and the Super Serial Card doesn't forget previous pokes unless reset is pressed. The other day I printed a document using *Apple Writer*, then used "control-Q" to exit "gracefully." I switched to another ProDOS program (never needing to press reset), then tried printing something out to the printer from this program. The printer did not advance the paper. Line after line was printed one on top of another, since the Super Serial Card would not issue linefeed characters after carriage returns. The change that *Apple Writer* had made was still in effect. Remembering these problems, I pressed control-reset to force the Super Serial Card to reinitialize itself, then printed again. This time it worked fine.

Phil Thompson
Portland, Ore.

So ***that's*** *why my spelling checker prints differently every time I use it. Apple is always harping at developers to put everything back just like they found it (no problem here, I agree with them), but Apple's own software is as bad as any at not following the rules.*

The ProDOS quit feature, combined with program selection software, has created a trend toward not rebooting machines between applications. Thus, it is becoming more and more important that each application program undo any changes it makes to a computer or its attached devices.

My comments about it being "best" to use large memory cards as RAMdisks were aimed at both programmers and users. It's true, however, that a significant problem with the auxiliary-slot cards is that the software that makes them work must be "installed" every time the user reboots. However, neither Applied Engineering's nor Checkmate's aux-slot RAMdisks will lose their contents during a reboot as long as bank 0 is locked out. I'm still looking for a universal, licensable RAM disk driver that would recognize any type of RAM and convert it all into a large RAMdisk if no other driver was already installed. Commercial developers could include such a driver within their software to make memory management totally transparent to users. The advantage to developers, of course, is that all RAM cards (to say nothing of hard disks and whatever storage devices the future holds) look and act alike, even though they are in reality quite different.

Open-Apple

[redacted]

is written, edited, published, and

Business Consultant Richard Barger
Technical Consultant Dennis Doms
Circulation Manager Sally Tally
Business Manager Sally Dwyer

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

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 single back issues are currently available for $2 each; bound, indexed editions of Volume 1 and Volume 2 are $14.95 each. Volumes end with the January issue; an index for the prior volume is included 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 from Speech Enterprises, P.O. Box 7986, Houston, Texas 77270 (713-461-1666).

ISSN 0885-4017 **Source Mail: TCF238**
Printed in the U.S.A. **CompuServe: 70120,202**