Open-Apple™

June 1987
Vol. 3, No. 5

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

Releasing the power to everyone.

Making AppleWorks relational

Since my March article on how to create your own program that can *read* AppleWorks data base files, requests have been coming in for the promised companion article on how to *write* such files. This month we're going to fulfill that promise.

One of our subscribers who's in a hurry to see this companion article pointed out that auxiliary programs able to read and write several data base files at a time could turn AppleWorks into a *relational data base system.* While I'd heard the words "relational data base" many times and understood this system was a powerful way to store data, I didn't really know much about it. My interest was tweeked, however, and I started nosing through a book called *Database: A Primer,* by C.J. Date (Addison-Wesley).

According to Date, the "relational model" is concerned with three aspects of a data base: its structure, its integrity, and the ways it can be manipulated.

The *structural* part of the relational model specifies that all data must be kept in tables. The AppleWorks data base does, in fact, store your data that way. You can see it by looking at one of your data base files with the multiple-record display, which puts each of your records in its own separate row and each of your categories in a column. (Usually, however, you can't actually see all of the columns because some are beyond the right edge of the screen. Unlike the AppleWorks spreadsheet, the data base doesn't allow you to scroll sideways to see the other categories. You can always use the open-apple-L(ayout) command, however, to move any specific column onto the screen so you can see it.)

In addition to the general table structure, the relational model also specifies that each row must contain *exactly one* value for each column. AppleWorks allows you to put any number of values in a row-column cell. For example, look at Smith's "administer at" cell from this sample data base, which is called "patient-medication-times":

patient	medication	administer at
Smith	Naprosyn	08:00, 20:00

To conform to the relational model, this AppleWorks file must be modified so that only one value appears in each "administer at" cell. One way to do this would be to change the data to look like this:

patient	medication	administer at
Smith	Naprosyn	08:00
Smith	Naprosyn	20:00

Tables with exactly one value in each cell are said to be "normalized." According to Date, normalized tables are "a special case of the construct known in mathematics as a *relation*....The whole relational model is founded on elementary mathematical relation theory."

While a data base with just one table could follow all the rules of a relational data base, the power of relation theory doesn't come into play until you have more than one table. While AppleWorks does allow you to have more than one table on its desktop at once, it doesn't have any ability to manipulate or view data from two or more tables simultaneously. This is the major thing that prevents AppleWorks from being a relational data base system. Auxiliary programs with the ability to read and write AppleWorks data base files, however, could unite several AppleWorks tables into a true relational data base—a data base consisting of a number of tables related to each other by means of "keys."

The *integrity* part of the relational model has to do with these keys. There are two kinds of keys, primary keys and foreign keys. "Every table should have a primary key—that is, a field, or field combination, that serves as a unique identifier for the records in that table," Date says.

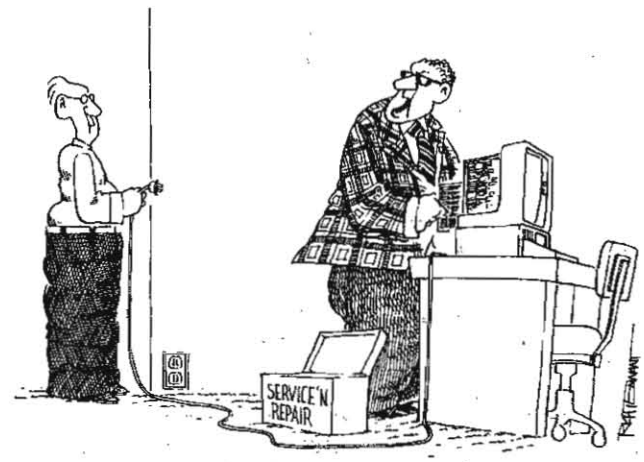
Account numbers, it would appear, were invented to fill the need for primary keys—for unique identifiers. Paradoxically, most people view account numbers as impersonal, yet their function is often ultra-personal—to uniquely identify a specific person in a data base. Names usually aren't used as a primary key because more than one person can have the same name.

What is the primary key in the table shown earlier? Even if we used a patient number instead of "Smith," that column couldn't be the primary key because both records refer to the same patient. The only column in the table that has unique values is the "administer at" column, yet it's easy to imagine the table growing to something like this:

patient	medication	administer at
Smith.073	Naprosyn	08:00
Smith.073	Methocarbamol	08:00
Smith.073	Methocarbamol	16:00
Smith.073	Naprosyn	20:00
Smith.073	Methocarbamol	24:00

Now, no single column has unique values. However, Date did say that a primary key could be a "field combination." The primary key of this table, in fact, is a combination of *all* of its columns.

It's important to understand that the word "key" doesn't mean the same thing as "column." While, as mentioned earlier, no row-column *cell* in a relational data base can hold more than one value, a *key* can be made up of several values. The important thing about the primary key is that no other row in the table can have the same key value. The value in the primary key, whether made up of one or several columns, must uniquely identify each row. (It follows from this, and Date makes a point of saying it, that a row's value for the primary key can never be "null" or blank.)



"FIRST I'LL SWITCH OUT THE MOTHERBOARD, THEN WE'LL TRY YOUR IDEA."

A "foreign key" ties one table to another table by linking with the second table's primary key. Foreign keys do not have to be unique (several rows in the "foreign table" may be related to the same row in the "primary table.") However, every value in the foreign key of a table must either be equal to a value in the primary key of the other table or be null. Said another way, a foreign key links each row in its own table with exactly one row in another table. A foreign key value that doesn't exist in the primary key of the other table is illegal. A blank foreign key, which means that the row doesn't link with anything in the second table, is allowable, however.

In our sample "patient-medication-times" table, for instance, the "patient" column could be defined as a foreign key capable of linking that table with a "patient-census" table that has exactly one row for each patient. The patient-census table might look something like this:

patient	room	physician	primary nurse	allergies
Smith.073	209	Jones.04	Doe.01	yes

In addition, the "room," "physician," and "primary nurse" columns of patient-census could all be foreign keys linking this table with other tables. Even the primary key of this table, "patient," could be a foreign key into another table that also had "patient" as a primary key (that table might hold non-medical information about the patient, such as address, phone number, and insurance company).

The *manipulative* part of the relational model "consists of a set of operators known collectively as the *relational algebra*. Each operator of the relational algebra takes either one or two tables as its operands and produces a new table as its result." These operators do things such as combine two tables (called JOIN), extract certain rows from a table (called SELECT or RESTRICT), and extract certain columns from a table (called PROJECT).

AppleWorks has some of these powers. It particular, it has two tools, open-apple-R(ecord select) and open-apple-F(ind), for SELECTing certain rows from a table. In conjunction with the clipboard, you can even use these tools to move the extracted rows to a new table.

Open-apple-L(ayout) allows you (forces you, in fact) to extract certain columns from a table. Open-apple-L(ayout) is not a true PROJECT, however. The PROJECT operator is also supposed to ignore any redundant rows. For example, if we were to PROJECT our patient-medication-times table "over" patient and medication, the result should include only two rows, not five:

patient	medication
Smith.073	Naprosyn
Smith.073	Methocarbamol

AppleWorks can't do that.

In addition, AppleWorks includes no way to join two tables. The JOIN operator takes two tables and makes a new, wider table out of them. JOIN requires that you specify a column in each table. It concatenates all rows in which the specified columns have equal values. For example:

A	a	b	c
x	a1	b1	c1
x	a2	b2	c2
y	a3	b3	c3

JOINed with

A	d	e	f
x	d1	e1	f1
y	d2	e2	f2
y	d3	e3	f3

on "A" is:

A	a	b	c	d	e	f
x	a1	b1	c1	d1	e1	f1
x	a2	b2	c2	d1	e1	f1
y	a3	b3	c3	d2	e2	f2
y	a3	b3	c3	d3	e3	f3

The result shown here is called a "natural" join. If the "A" column appeared in the resulting table twice, instead of once as here, that would be an "equijoin."

(Notice that JOIN is just a mathematical operator that works whether you have legitimate primary and foreign keys or not. Column A isn't a legitimate primary key in any of these tables because the values in that column aren't unique. Consequently, Column A isn't a legitimate foreign key in any of these tables, either. A foreign key can't exist without a legitimate primary key to link to. The foreign key is supposed to link its rows to *exactly one* row in the primary key's table. If the values in the "primary key" aren't unique, this isn't possible.)

In essence, then, a *Relational Data Base System* is software that supports the storage of data in tables made up of columns of categories and rows of records. Each table must have a primary key that is made up of the values in one or more columns. The values in the primary key must uniquely identify each row in the table. Tables may also have foreign keys that tie each row in the table with the exactly one row in another table by linking to that second table's primary key. The software must support the relational operators SELECT, PROJECT, and JOIN for creating new tables.

AppleWorks provides a small subset of a relational system—it stores data in a table and it provides a SELECT operator.

In his book, Date doesn't address the advantages relational systems have over the older *hierarchical* and *network* data base systems. He simply posits that the merits of the systems have been well aired and that "most database professionals now believe that relational technology is the way of the future." It appears to me, however, that the advantages of relational systems must be that they are efficient in terms of storage used, that they are relatively easy to expand or redesign, and that they provide powerful tools for quickly extracting specific information from a large data base.

When designing a relational data base, Date says, you need to first decide what "entities" you are trying to organize and give each its own separate table. In our previous example, for instance, "patient" is an entity that should get its own table (patient-census). Each table consists of its primary key, which is a field or field combination that uniquely identifies the rows in the table, and other fields, each of which "represents a fact about the key, the whole key, and nothing but the key."

When a field in a "wide" table needs to hold more than one fact about the key, the best strategy is to create a new table. (When a "narrow" table needs to hold more than one fact about a field, the best strategy may be to make that field part of the key and duplicate the row—as we did in our patient-medication-times table, for example.) In our patient-census table, one of the fields is for "allergies." A patient can have more than one allergy. If we follow the same "narrow" strategy that we used for patient-medication-times, we get:

patient	room	physician	primary nurse	allergies
Smith.073	209	Jones.04	Doe.01	penicillin
Smith.073	209	Jones.04	Doe.01	raspberries

There are two problems with this strategy. The first is that it uses up lots of storage for redundant information (why should room, physician, and primary nurse take up space for each allergy?). The second is that "allergies" must be included as part of the primary key to keep each row unique. But if "allergies" is part of the primary key for this table, then our patient-medication-times table, in order to link to this one, would have to look like this:

patient	allergies	medication	administer at
Smith	penicillin	Naprosyn	08:00
Smith	raspberries	Naprosyn	08:00

Adding allergies to this table just so we have a foreign key to link back to the patient table's primary key makes no sense, either. So, instead, we incorporate "allergies" into our data base by using the "wide" strategy of forming a new table. Then we have:

patient	room	physician	primary nurse	allergies
Smith.073	209	Jones.04	Doe.01	yes

and

patient	allergies
Smith.073	penicillin
Smith.073	raspberries

Using two tables is very efficient and powerful from the data base system's perspective. From a *user's* perspective, however, tables such as these may be the best way to organize the data:

patient	allergies	administer at	medication
Smith.073	penicillin	08:00	Naprosyn
	raspberries		Methocarbamol
		16:00	Methocarbamol
		20:00	Naprosyn
		24:00	Methocarbamol

```
administer at    room    patient        medication
================================================
08:00            205     Smith.073      Naprosyn
                                        Methocarbamol
                 210     Singh.029      Tetracycline
```

A user-oriented table is called a "view." A view extracts pre-selected data out of a number of different physical tables and presents it in a format useful to a user. Different users usually have different views of a data base. A primary nurse requires views that are different from those the accounting department needs.

There is a sense in which a view is a "report format" printed to screen. However, in an ideal system, users could add to or modify the data in the underlying tables by editing the values in one of their views. You can't do that with printed reports.

Data base views are powerful tools. They allow data base tables to be structured in a way that is best for data storage, with no concern at all for users. On the other hand, they allow users to see and manipulate data in a way that is best for them, with absolutely no concern about the actual structure of the underlying tables.

The relational system described by Date appears to be a non-existent ideal—he regularly points out features that are missing from the four relational software packages he discusses in his book. Nonetheless, it doesn't sound impossible to create a simple, elegant, AppleWorks-like relational data base for the Apple II. ProDOS certainly needs (and deserves) a sophisticated data base program.

The software packages described by Date mostly use a "command line" interface (like Applesoft's), though one provides a form that you fill out to access data. None of them take the more AppleWorks-like approach of showing you the actual data in a table you can scroll through and allowing you to manipulate things by *pointing* at what you are interested in. I suspect that a data base program that worked like that, that had the speed and simplicity of AppleWorks, and that was *programmable* (so that specific systems for primary nurses or for user-group secretaries or for Apple's software licensing department could be designed and sold like templates) would be a hit.

Meanwhile, as we wait for one of *Open-Apple*'s subscribers to write this software, the rest of us may get some mileage out of manipulating AppleWorks data base files with our own programs.

You may remember from our March discussion of how to *read* an AppleWorks data base file that the fundamental idea is to BLOAD a section of the file into memory and PEEK at it. To *write* an AppleWorks data base file we do just the opposite—POKE values into a range of memory and then BSAVE that range.

The range of memory we PEEK or POKE is called a *memory buffer*. It must be protected so that Applesoft doesn't accidentally step on it. The following Applesoft program lines, which also appeared in my March article, set up the buffer and initialize the program:

```
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.

1050 BBG = FN PK2(105)-16384 : REM BBG points to the beginning of our buffer.
1052 BEN = BBG + 16384       : 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.
```

In the March article, the 16384 in line 1052 was divided by 2. This created space for two buffers. At the time, I said the second one would be used when I showed you how to write AppleWorks files. Well, that's what I'm doing now, but we're still using just one buffer, so there's no need for the division either here or back in March.

When I wrote the March program, I figured this month's demonstration would read a record from an AppleWorks file in the first buffer, do some interesting manipulations to it, and write it into the second file using the second buffer. In fact, this month's demonstration gets its input data from a text file, not an AppleWorks file. Thus, this month's routines have to deal with just one AppleWorks file at a time—the one being written to—and require just one buffer.

(A program that tried to implement the relational SELECT or PROJECT commands would have to deal with two files (one input, one output); a program implementing JOIN, three files (two input, one output); while a program implementing the view feature might have to deal with a dozen or more files. It's not that confusing to deal with more than one file at a time by setting up multiple buffers, trust me. I just don't want to complicate things by getting into it this month.)

File structure. You may remember from March that an AppleWorks data base file has three main parts. In the beginning is the header, which tells how many records are in the file, how many categories are in each record, what the names of the categories are, and other interesting stuff. Next comes a section that holds report formats, if any have been defined. Finally comes the actual data.

Each data record begins with a two-byte record length. Next comes a *control byte* for the record's first category. When this byte is less than 128 it indicates how many bytes of ASCII data are in the category. When this byte is greater than 128, it indicates the next category or group of categories is blank. A value of 129 indicates one blank category; 130, two blank; and so on. If the first category holds ASCII data, the data is immediately followed by the control byte for the second category, and so on. A control byte of 255 ($FF) marks the end of the record; if any categories remain when the $FF is reached they are all blank. For a more detailed description, see March's article.

The reason this month's program gets its data from a text file is that ***Open-Apple*** has a new 800-number answering service. New subscribers will call this number (we hope) to order subscriptions. The answering service operators enter the caller's name, address, and credit card number into a computer. Every so often these "records" are sent to us by modem. We needed a program that would transfer this data into the AppleWorks data base file we use for processing credit card transactions.

The strategy I used for converting the text file data into an AppleWorks file was to read one record from the text file, poke that record to the AppleWorks file image in memory, read one, poke one, and so on, until the end of the text file was reached. This works better than reading the whole text file and then poking the whole AppleWorks file because it uses a consistent, manageable amount of memory. Here are the arrays I used to hold the data:

```
1031 DIM I$(30) : REM this array is for the categories of the input file.
1032 DIM O$(30) : REM this array is for the categories of the output file.
1033 DIM CB(30) : REM this array is for control bytes in the output file.
1034 DIM TD(30) : REM this array is for time/date ids in the output file.
```

I$(n) is where I put the data I read from the text file. O$(n) is where I put the data I wanted to poke into the AppleWorks file. In each case, the (n) indicates a category. My program filled I$(n) by reading one record from the text file; manipulated that data and transferred it to O$(n); poked the data in O$(n) into memory; then went back to read a new set of I$(n) values.

The TD(n) array was filled just once—it indicates which output categories are in the special AppleWorks time/date format. In my own case, all of the values in this array were zero. If you want output category 5 to hold a date, however, put a 192 into TD(5). For a time category use 212. We'll look at this, and at the CB(n) array, more later.

Data manipulation. If there was no need to manipulate the data, it would have been easier to simply use AppleWorks itself to load the 800-number ASCII text files into a new data base file. From there we could use the clipboard to copy everything into our credit card data base file. However, the order in which the 800-number people take each category of information is different from the order in which the categories appear in the credit card file. Consequently, this didn't work. In addition, I wanted to check each credit card number for the correct number of digits and make sure it had blanks in all the right places.

Since ***Open-Apple***'s own 800-number program is of little general interest, I don't intend to reprint the whole thing here. I'll skip the part that reads the text file (see the January 1985 ***Open-Apple***, page 2 for more on how to do that) and I'll skip most of the part that transfers data from the I$(n) strings to the O$(n) strings. Just to give you a taste for that part of the program, however, here are a few lines you might like to look at:

```
3544 O$(0)="bad card number" : O$(17)="ERR" : O$(25)=""
3545 IF LEFT$(T$,1)="4" THEN O$(17)="VISA" : GOTO 3550
3546 IF LEFT$(T$,1)="5" THEN O$(17)="MC"   : GOTO 3550
3547 O$(25)=I$(1)+" is not a VISA/MC number." : GOTO 3600
3550 REM continue with program

3600 REM continue with program
3602 O$(2)=I$(2)
3619 O$(19)=I$(4)
```

Line 3602 simply transfers input category 2 into output category 2. That category accidentally appears in the same position in both files. It's the caller's name. Line 3619 transfers input category 4 into output category 19. That category, which holds the caller's street address, changes positions between files, as most categories do.

In lines 3544 through 3547 I did some fancier manipulations. Output category 0 is for the credit card number, 17 for the credit card type (VISA or

MC), and 25 for an *error message* that will actually appear on the AppleWorks data base screen and explain any problems with this caller's record.

This program segment begins in line 3544 by assuming the credit card number will be bad and setting output categories 0 and 17 to reflect that. Any previous error message in category 25 is erased, however. Line 3545 checks to see if the first digit of the credit card number is a 4. If it is, the card is a VISA card and output category 17 is set to show that. Control would then pass to the part of the program that formats the card number.

If the first digit of the credit card number isn't a 4, line 3546 checks to see if it is a 5, which would indicate a Mastercard number. If the first digit of the number is anything else, line 3547 assigns an error message to output variable 25 and skips the part of the program that would replace the "bad card number" message in output field 0 with the correctly formatted number.

Writing the file. The strategy I used for writing (poking) the output categories into memory involved two passes at the record. During the first pass I calculated the total length of the record and the value for each control byte. During the second pass I actually POKEd the control bytes and ASCII data into memory. Here's the first pass:

```
6000 REM Write one record to file F
6001 REM   category data expected in D$(n)
6002 REM   time/date TDs in TD(n)

6010 RL=1 : REM minimum record length (final $FF byte)
6015 NB=0 : REM counter for blank categories
6020 FOR N=0 TO NC-1 : REM for each category:
6025   CB(N)=LEN(D$(N)) : REM calculate control byte
6030   IF CB(N) > 127 THEN ERR=1 : GOSUB 6900 : REM data too long error
6035   IF CB(N) = 0 THEN NB=NB+1 : GOTO  6060 : REM category is blank
6040     REM category is not blank
6042     BFLAG=0 : REM clear blank flag
6044     IF NB>0 THEN CB(N-1)=128+NB : BFLAG=1 : NB=0
6046     IF TD(N)=192 THEN CB(N)=6 : REM date category
6048     IF TD(N)=212 THEN CB(N)=4 : REM time category
6050   RL=RL+BFLAG+1+CB(N) : REM add category length to record length
6060 NEXT
```

The critical variables in this section of the program are RL (record length) and CB(n) (an array for the control bytes). RL is assigned an initial value of 1 in line 6010. This represents the final $FF byte.

Between lines 6020 and 6060 there is a loop that examines each category. Line 6020 sets the control byte for the current output category to the length of its ASCII data. Line 6030 checks to see if that length is greater than 127; if it is, a subroutine shown later truncates it and sends a message to the screen. The AppleWorks data base file format doesn't allow more than 127 characters per category.

If a category is blank, the control byte will hold a zero in line 6035. In this case, NB, a variable for counting consecutive blanks, is incremented and we immediately skip to the next category. If the control byte isn't zero, on the other hand, the category isn't blank and lines 6040 through 6050 will be executed.

Line 6042 makes sure that a flag we'll use later is correctly initialized. Line 6044 looks to see if there were any blank categories immediately before the current category. If so, the control byte for the *previous* (CB(N-1)) category is set to indicate how many blank categories there were.

So, for example, if categories 3 through 6 are blank, their control bytes will all hold zeros until, while processing the non-blank category 7, we get to line 6044. We leave the control bytes for categories 3 through 5 at zero, but we set the control byte for category 6 to 132, which indicates four blank categories (132-128=4).

When we go to poking all these control bytes into memory we'll skip the ones set to zero, so the simple 132 will represent all four categories in the file image. If the *final* categories of a record are blank, all of their control bytes will hold zeros. In this case, we don't need to include a control byte in the file—the record ending $FF telegraphs that all the remaining categories are blank.

In addition to setting the control byte for the previous category to "blanks," line 6044 sets BFLAG to one and sets NB, the number of consecutive blanks, to back to zero. BFLAG makes its final appearance in line 6050, where we sum the lengths of all the categories to determine the length of the whole record. BFLAG makes sure that all the "blanks" control bytes get included in the summation.

Lines 6046 and 6048 look at TD(n), the time/date array, to see if the category under examination holds an AppleWorks-format time or date. (For more information on this format, see the March article.) If so, it forces the control byte to 6 for a date, or to 4 for a time. The length of a date includes an ID byte and five ASCII characters; a time an ID byte and three ASCII characters.

Finally, line 6050 adds together the BFLAG, a 1 (for the control byte), and the control byte value itself (which holds the length of the data). After looping through all the categories, RL will hold the length of the record and CB(n) will hold the control bytes for each category.

The next step is to poke the record into the memory buffer:

```
6100 REM Poke record into memory buffer.
6110 POKE PNTR+1,RL/256 : REM poke in record length
6112 POKE PNTR,RL - PEEK(PNTR+1)*256
6114 PNTR=PNTR+2
6120 FOR N=0 TO NC-1 : REM for each category
6125   IF CB(N) = 0 THEN 6160 : REM if blank, then skip to NEXT
6130   POKE PNTR,CB(N) : PNTR=PNTR+1 : REM poke in control byte
6135   IF CB(N) > 128 THEN 6160 : REM if blanks, then skip to NEXT
6140   IF TD(N) > 0 THEN POKE PNTR,TD(N) : PNTR=PNTR+1 : CB(N)=CB(N)-1
6145   FOR I=1 TO CB(N) : POKE PNTR+I-1,ASC(MID$(D$(N),I,1)) : NEXT
6150   PNTR=PNTR+CB(N)
6160 NEXT
6170 POKE PNTR,255 : PNTR=PNTR+1  : REM write $FF at end of record
6180 NR=NR+1 : REM number of records
6190 RETURN
```

Lines 6110 and 6112 poke the record length into the buffer. Lines 6120 through 6160 form a loop that pokes in each category's data. In line 6125 we check for a control byte of zero; if found, we do no pokes and instead skip ahead to the next category. Line 6130 pokes any other control byte value into the buffer.

Line 6135 looks to see if the control byte just poked was a "blanks" byte. If so, we skip the rest of the loop. If not, we next look to see if this is a date/time category. If so, the date or time ID byte is poked into the buffer. Then the control byte for that category is reduced by one; this is necessary because the actual ASCII data for a date or time is one byte shorter than the real control byte value indicates (it includes the length of the date/time ID byte as well as the length of the ASCII data).

Finally, line 6145 is a poke loop that puts a category's data into the memory buffer. Line 6150 advances the buffer pointer. When all of a record's categories have made it to the buffer, line 6170 pokes in the record-ending $FF. Line 6180 then increments the number-of-records counter.

The header. Everything we've said so far just kind of assumes that we've already poked a data base file header into the memory buffer. In fact, this isn't done with pokes but by stealing a header from an existing file:

```
1070 F$="TEMPLATE" : REM data base file we'll steal a header from

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
```

These lines, with the exception of 1070, come straight from March's ADB.READER program. The subroutine used by line 1110 loads the AppleWorks file in F$ into the memory buffer. The rest of the lines dig some interesting values out of the header.

The next thing we need to do is CREATE the data base file we'll be storing data in. The BSAVE commands our routines use won't do this automatically because we are specifying a special file type (ADB—AppleWorks data base). These lines will do that:

```
1200 W$="NEW.FILE"
1210 ONERR GOTO 1216
1212 PRINT CHR$(4);"CREATE";W$;", TADB"
1214 POKE 216,0 : GOTO 1220 : REM turn off ONERR
1216 IF PEEK(222)=19 THEN PRINT CHR$(4);"DELETE";W$ : RESUME
1218 PRINT "ERROR #";PEEK(222);" IN LINE ";PEEK(218) + PEEK(219)*256 : STOP
```

In line 1215 there is a reference to error 19—DUPLICATE PATHNAME ERROR. If this occurs, it means a file named W$ already exists. We solve that problem by deleting it. Line 1216 simply handles any other error that occurs while our ONERR routine is active between line 1210 and the POKE 216,0 in line 1214, which turns ONERR back off.

The next thing we have to do is move the buffer pointer from the beginning of the file past the header, past the report formats, and past the first record of the file, which holds "standard values." Here's how to do that:

```
1220 REM move pointer to data, resaving file if necessary
1230 PNTR=PNTR+357+(NC*22) : REM skip past header
1240 IF NF=0 THEN 1290     : REM skip past report formats
1250 FOR I=1 TO NF
1260   IF PNTR+600 > BEN THEN GOSUB 6500 : GOSUB 5500
1270   PNTR=PNTR+600
1280 NEXT

1290 REM skip over standard values
1291 RL = FN PK(PNTR)
1292 IF PNTR+RL+2 > BEN THEN GOSUB 6500 : GOSUB 5500
1293 PNTR=PNTR+RL+2

1295 NR=0 : REM reset number of records to zero
```

The only trick to all of this is making sure that we don't overrun our buffer. (With the big 16384-byte buffer we're using this month it's no trick, but you may want to use a bunch of much smaller buffers in your own program.) As in March, we assume a minimum buffer size of about 1.25K. This means the entire header will fit in the buffer and solves some problems. In line 1230 we move the pointer past the end of the header to the beginning of the report formats.

Back in line 1126 we dug the number of report formats out of the header and placed that value in NF. Lines 1250 through 1280 form a toe-in-water loop that works like this. For each report format that has been defined we add 600 (the length of each report format) to the buffer pointer and see if the total is beyond the end of the buffer. If it is, then before actually advancing the pointer we call the subroutine at 6500 to save the portion of the file that's in the buffer to disk. Next we call the subroutine at 5500 to load the next portion of the original file into the buffer. A similar toe-in-water scheme is used in lines 1290 to 1293 to skip over the "standard values" record, which is the first record in the data section of the file.

The routines that do the actual buffer loading and saving look like this:

```
5500 REM load section of file into buffer
5510 BYTE=BYTE+(PNTR-BBG) : PNTR=BBG
5520 PRINT CHR$(4);"BLOAD";F$;", TADB, L16384, A";BBG;", B";BYTE
5530 RETURN

6500 REM save contents of buffer to disk
6510 L = PNTR - BBG
6520 PRINT CHR$(4);"BSAVE";W$;", TADB, B";BYTE;", A";BBG;", L";L
6530 BYTE=BYTE+L : PNTR=BBG
6540 RETURN
```

The "load" subroutine appeared in our March article (I did change the L parameter to 16384 from 8192 because of the larger buffer we're using this month, however). Both subroutines take full advantage of the extra parameters Basic.system allows with BLOAD and BSAVE commands. The "T" parameter allows us to manipulate non-binary files. The "B" parameter allows us to begin loading or saving at a specific byte in the file.

More information about the "load" subroutine can be found in our March article. The "save" subroutine works by first calculating the length of what's about to be saved in line 6510. This length is the distance in bytes between the beginning of the buffer and the current pointer position. Line 6520 saves the image in the buffer into the file, beginning at the byte specified by the BYTE variable. The first time the buffer is saved, BYTE will equal zero. Line 6530 advances BYTE for subsequent saves. It simply adds the length of image just saved onto the previous value in BYTE. This line also resets the buffer pointer to the beginning of the buffer.

One final place we need a toe-in-water scheme is when we are actually poking categories into the memory buffer. How about these lines, which fit in between those that determine the length of a record and those that actually poke a record into memory:

Ask (or tell) Uncle DOS

Our corrections this month come from Ingle Farm, South Australia, whence ***Open-Apple*** *subscriber Lili Gray points out that back in December 1986, on page 2.85, half way down the third column, the correct address of the Basic.system input hook is 43690-91, not 43607-08 as written. Gray also reminded me to mention another error several of you noticed in our recent April issue—in my introduction to that month's letters (page 3.18), the page I should have encouraged you to turn to was 3.5, not 3.7. Now that our corrections are overlapping on themselves, I really feel like an assembly language programmer.*

I am aghast and agape at the response *we got to last month's call for Apple's officers to take a stand on the issue of Apple's software not following Apple's own protocols ("Slot 3 RAMdisk rules," page 3.30). Rather than change a few bytes of code, Apple quickly decided to solve the problem by moving all its application software into a separate company and spinning the new company off to the public, according to the April 29* ***Wall Street Journal*** *(page 4). Since large software companies have never followed Apple's published protocols anyhow, there will now be no need to modify Apple products such as AppleWorks,* ***Apple Writer,*** *or* ***Instant Pascal*** *to follow them, either.*

Although the ***Wall Street Journal*** *didn't mention the May* ***Open-Apple*** *as the primary factor in Apple's decision, the facts speak for themselves. Instead, the newspaper quoted John "Scully" (whoever he is—the president of Apple has an "e" before the "y" in his name) as saying that the focus of the new company will be to "create and market innovative software for Apple's Macintosh and Apple II computers."*

I for one will be even more aghast and agape than I am now if this new company produces anything innovative for the Apple II. The company will be headed by William V. Campbell, who is now Apple's top marketing executive. Campbell's contributions to the Apple II have so far been nil to negative. Under his leadership Apple has refused to promote the Apple II (and occasionally actively discouraged sales) in business and university markets. Under his leadership Apple dropped most of its Apple II dealers outside of major metropolitan areas. He is a Macintosh yuppie from stem to stern and I predict that any innovative stuff that comes from his company will run only on a Macintosh.

Paradoxically, his company's biggest product off the starting line will be AppleWorks, although none of the computer trade papers have picked up on that startling fact yet. According to the ***Wall Street Journal,*** *neither the new company nor its products will use the "Apple" trademark, which may mean that even the product names "AppleWorks" and "****Apple Writer****" will soon disappear from sight.*

What all this means to the typical ***Open-Apple*** *reader is that the Apple II software market is on the verge of a renaissance. By totally dominating this market since the introduction of AppleWorks, Apple accidentally squeezed many one-time Apple II developers into IBM or Macintosh products (or into squash). I expect the best of those developers will come back and join both surviving Apple II software companies and new companies we haven't heard of yet in the next great software gold-rush.*

When I couple Apple's upcoming market withdrawal with the recent introduction of the IIgs, I see more opportunity for Apple II software developers than even during the boom years of 1980-83. Get out there and develop something innovative. I smell buyers in the wind for a change.

This issue of Open-Apple is the first *to be sent to more than 10,000 paying subscribers. We appreciate your support and encouragement. Things are going so well I've decided to come out from underground and start advertising in Apple II magazines. I even hired an answering service with an 800 number to take orders.*

Our first ads will appear in the July issues of ***Call-A.P.P.L.E., Nibble, inCider,*** *and* ***A+*** *and will continue monthly thereafter. Look for them—each ad includes a* ***different*** *tip from a previous issue of* ***Open-Apple****. In addition, each ad will also include some of the more complimentary things you've written in your letters to us. We've always gotten lots of blush-inducing fan mail from our readers but have included only snippets of it here because it's not what you're paying to read. I've always hated to waste all that good stuff, however, so I'm really pleased I've finally got someplace to publish it.*

I occasionally bill ***Open-Apple*** *as the "meeting place of the world-wide Apple II community." Even so, even I was amazed when we actually did a count last month and found paying subscribers in 54 different countries. Just to celebrate, I made a point of selecting a number of world-wide letters for Uncle DOS this month. (Our subscribers outside the U.S. are an intelligent and active bunch—they're forever sending us questions we can't answer.)*

The only bad news I have is that I've run out of time and space for the exposition on the IIgs memory manager that I promised you last month. Just give me 30 more days....

Power to the Apple II

I live in West Africa where the 220 volt, 50 cycle power is less than dependable. The power here is full of spikes, dips, brown-outs and all manner of trash. To

```
6090 REM Will record fit in buffer?
6095 IF PNTR+RL+2 > BEN THEN GOSUB 6500
6096 IF PNTR+RL+2 > BEN THEN ERR=2 : GOTO 6900
```

That is pretty much everything you need to write AppleWorks data base files. Not so hard, was it? There are also a couple of error routines that I've placed at the end of this article. In addition, you'll find it helpful to see the actual guts of a file-writing program:

```
1500 PRINT CHR$(4);"OPEN TEXT.FILE"
1510 GOSUB 2500 : REM get next record from text file
1515 IF EOD=1 THEN 1550 : REM if at end of text file, save buffer & quit
1520 PRINT "Record ";NR+1 : REM assure user something is happening
1530 GOSUB 3000 : REM manipulate data
1540 GOSUB 6000 : GOTO 1510 : REM write record, continue

1550 PRINT CHR$(4);"CLOSE TEXT.FILE"
1560 POKE PNTR,255 : POKE PNTR+1,255 : PNTR=PNTR+2 : REM two file-ending $FFs
1570 GOSUB 6500 : REM save buffer to disk
1580 POKE BBG,NR : REM fix number-of-records byte
1590 PRINT CHR$(4);"BSAVE";W$;",TADB,L1,B36,A";BBG
1600 END
```

Line 1500 opens the text file that holds the data we want to get into an AppleWorks file. Line 1510 calls a subroutine not shown here for reading one set of categories — one record — from that file. If this subroutine encounters an END OF DATA error, it should set the EOD variable to 1 before RETURNing.

First follow what happens when EOD comes back 0, meaning there are more records to read. Line 1520 prints the current record number to the screen, just to assure the user that something is happening. Line 1530 calls the subroutine, not shown here in full, that does all the data manipulations. Line 1540 calls the subroutine that pokes a record into the memory buffer, then returns to line 1510 for the next record.

When we get to the end of the text file, EOD comes back 1 and we go to line 1550. First we close our text file. Then we poke two additional $FFs at the end of the file, which is the final detail of the AppleWorks data base file format. In line 1570, we save the memory buffer to disk.

Lines 1580 and 1590 again use the BSAVE command, but this time with B set to 36 and L set to 1, to update just one byte of our new file. This is the byte in the header that holds the number of records in the file.

And that brings us to END.

```
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

6900 REM error handler for write instructions
6905 ON ERR GOTO 6910, 6920
6910 EMSG$="Category "+STR$(N)+" in record "+STR$(R)+" was truncated."
6912 O$(N)=LEFT$(O$(N),127) : CB(N)=127
6914 GOTO 6990
6920 EMSG$="I can't write record "+STR$(R)+" because it's"+CHR$(13)+
          "   longer than the memory buffer."
6922 GOSUB 6990 : STOP
6990 PRINT EMSG$ : RETURN
```

---

keep my system healthy, I run the power through a 220 to 110 volt transformer and then through a U.S. model 15 amp automotive battery charger. After the charger I have a cheap 12 volt car battery. The battery connects to a Tripp-Lite 550C Powerverter and my computer plugs into that.

Some people here have run the same system, but with a less-expensive 500 watt TrippLite that is not frequency controlled. The Apple doesn't seem to care.

Even my Sider hard disk works perfectly off the square wave power my Powerverter provides. So I also have a very serviceable hard disk system that will not crash. We have had several full power failures and the screen didn't even jump. I have as much time as I want to finish a document, save it, and park the heads. I could probably run an hour or so without power, but that would risk discharging the battery.

A 15 amp charger seems to be the minimum necessary to keep the battery fully charged and the system up. Previously I had an 8 amp 220 volt charger but I couldn't run the system continuously. I would charge overnight to get about five hours of up time. Unfortunately, with that system it was very easy to discharge the battery by getting carried away on a long night. You only get a few full discharges of an auto battery. So now I'm on my second battery, but I've had no problems with the bigger charger.

If the current is running closer to 240 than 220 the inverter gets a little too much voltage from the charger and produces close to the 132 volt upper limit for the Apple. This makes the screen wiggle a little. To get the right output voltage, I can turn the charger to trickle mode for awhile or adjust the transformer to a different tap.

I should mention that my IIe has a Disk II card, Thunderclock, CP/M Plus, CCS serial card, Sider card, and Apple 80-column card. I also have an Apple Monitor III and System Saver fan plugged into the Powerverter. When I tried to run my NEC Spinwriter off the inverter I got a wiggly screen. The NEC just thrives off the 50 cycle power from the transformer, however. I put a Transector surge protector between the transformer and the printer to clean the power up a little.

Bruce Slater
Niamey, Niger

I called Jerry Shepherd at Shepherd Marketing (PO Drawer 681339, Schaumberg, IL 60168 1-800-AC-SPIKE) to ask how typical your solution was and to find out what "frequency controlled" means. Shepherd specializes in selling uninterruptable power supplies, inverters, and similar items. He's a fountain of information about this stuff. (See "Reviewer's Corner" in our January 1986 issue, page 98, for more on uninterruptable power supplies.)

Shepherd said your system is a common solution for dirty 220 volt, 50 cycle power. He suggested that you try to get an RV-Marine battery, rather than an automotive battery, however. He said that marine batteries are designed to provide a steady stream of power for long periods of time. Automotive batteries, on the other hand, are designed to provide short bursts of power for starting engines on cold days. The voltage of an automotive battery starts to drop when only 20 per cent of its stored energy has been used. The voltage of a marine battery, on the other hand, doesn't drop until about 80 per cent of the stored energy is gone.

Shepherd also said that it's usually cheaper and easier to buy a battery charger locally than to use a 110 volt U.S. model. And he thinks it's best to stick with "frequency controlled" inverters, such as the one you have. This means it provides 60 cycle power. Shepherd said the Apple II itself will work just fine without frequency control, but some monitors and other peripherals won't.

Finally, Shepherd said that any computer or peripheral that has a modern "switching-type" power supply works fine (even a bit cooler) with the square wave power your inverter provides. Only older equipment that uses a "linear" power supply needs sine wave AC. The TrippLite 550C Powerverter currently sells for $229.

International AppleWorks I

About half a year ago I installed a Checkmate's MultiRAM CX/512 in my German IIc. It's a pity but I'm only able to use it as a RAMdisk rather than to expand AppleWorks. I sent the warranty card to Checkmate together with a question about how to expand the German AppleWorks version 1.2, but I haven't gotten any answer.

I guess the same problem is occurring with the simple program from Alan Bird, which you published in November (page 2.75, which is misnumbered as 2.78), that keeps AppleWorks from stopping on the way to the desktop. Bird's program doesn't recognize my German version of AppleWorks either. Is there anyone able to help me?

Hans-Juergen Kuehne
Neisser St. 2
D-3007 Garbsen 9 West Germany

We have seen a couple of the international versions of AppleWorks and, yes, their "guts" vary from the U.S. versions our patch programs assume. Dennis talked to Ron LaMee at Checkmate about your problems. LaMee said he has seen five international versions of AppleWorks: German, French, Italian, Spanish, and Hebrew.

Checkmate has attempted to produce expanders for some of the international versions, which are apparently not always numbered in exact correspondence with their U.S. counterparts. Specifically, the French and German versions are supported, although LaMee says he's not sure Checkmate has received a disk containing the most recent German version to verify the expander patches on. Since your previous letter seems to have slipped through the cracks, he will try to get a copy of the German version 1.2 expander to you; if you don't hear from him soon the European distributor for Checkmate is Pandasoft, Apple Vertragshandler, D-1000, Berlin 12. You should be able to get the upgrade software through them also.

As far as Alan Bird's patches go, multiple versions create multiple problems. A ***patch****, like the ones we publish here in* ***Open-Apple****, is usually a small one- or two-byte change to a program's machine language instructions. The difficulty is that the instructions a patch modifies usually move around inside a program between one version and another. Often the identical one- or two-byte change will work, but only if someone can figure out where the instructions to be changed have been moved. Consider that AppleWorks has been released in five official U.S. versions, to say*

nothing of international versions, and you'll immediately see why we'd get nothing else done if we took every patch request we get seriously.

Nonetheless, Dennis and I think we need to do ***something*** *to make the patch instructions we publish more universal. What we're considering doing is to give you a few bytes of the original code on both sides of any patch we print. You should be able to use a "disk zap" program such as ProSel's BLOCK.WARDEN, Quality Software's* ***Bag of Tricks 2****, Beagle Bros'* ***ProByter****, Central Point's* ***Copy II Plus*** *or even the IIe or IIgs Monitor to search for the code segment and change it, no matter where it's been moved to.*

The problem with this technique is that you, gentle readers, would have to learn how to patch programs this way. It's more difficult than typing in an Applesoft program, such as Bird's original, but not greatly so. But we can't help you a great deal—the specific procedure would depend on exactly which program you used to make the changes. Questions? Comments? Jokes?

A further complication is that some code doesn't stay exactly the same when it's moved; addresses embedded in a segment can change as well. In the case of Bird's patches, even the relevant machine language instructions change between U.S. AppleWorks versions 1.3 and 2.0, which kills all our hope of giving you a universal way to find the spot that needs to be changed.

ProDOS directory confusion

I am encountering a lot of problems when I try to store three or four programs (AppleWorks/Pinpoint, etc.) on the same 3.5 inch disk. Can you help?

Bruce Parry
Rawda, Kuwait

The strategy we use when moving sets of files to a large disk is to create a subdirectory for each set of files (APPLEWORKS for AppleWorks, APLWRITER for ***Apple Writer****, and so on). Then we copy all of the files on the original program disk into the appropriate subdirectory.*

To start up a program, it is usually necessary to first set the prefix to the subdirectory the program is in, then "dash" the system file. For example, with a disk named /UNI:

```
to run Apple Writer:
   PREFIX /UNI/APLWRITER
   -AW.SYSTEM

to run AppleWorks:
   PREFIX /UNI/APPLEWORKS
   -APLWORKS.SYSTEM
```

If the program being started is written correctly, it will use the existing prefix to find the rest of its files. Unfortunately, not all programs do this. Some look in the main volume directory for their files; some force you to enter pathnames to their various additional files. The latter are really a pain if you move the files to another disk; once moved, the program configuration must be re-entered to fix the pathnames.

Some programs don't remember which subdirectory they were started from if you change the prefix after startup. An example is ***Apple Writer****, which always looks for the file it needs to initialize a disk in the currently active subdirectory rather than in its own subdirectory. That should be classified as a bug.*

International AppleWorks II

I use the AppleWorks spreadsheet to display columns of figures. I would like a "blank when zero" option so that fields that contain a zero value display as spaces instead of "0.00," which I think looks untidy in a report. If you are scanning a report with a lot of zero-filled fields it is easy to miss a single field set to, say, "0.80." My answer so far is to print the report to the clipboard, copy it into the word processor, then edit out all the offending values, but is there a better idea?

Although AppleWorks was designed for the U.S. market, it is also marketed internationally. In spite of this there is no way to enter a date in anything other than the funny U.S. month-first format—nobody else in the world does it this way. Does anybody have a patch to make the date appear in a more generally accepted format such as dd-mmm-yy?

Another "U.S. only" gripe is that the spreadsheet formatting options allow for a currency layout but the only currency symbol supported is the dollar sign. Does anyone know of a patch to change this to another character? Currently, I either have a special column just to hold the currency symbol, or I print to the word processor and replace all the dollar signs.

Has anyone found a way to cause the word processor to do proper right justification? By this I mean a straight right edge with a ragged left edge. This layout is useful for adding page headers to the top right or bottom right of a page respectively and can also be used in a letter to put the date against the right hand margin (where it always used to be before people started using word processors).

On a different but related subject, I recently noticed that the chars-per-inch setting that is in effect when you set a header or footer in AppleWorks remains in effect when the header of footer is printed, even if you have since changed it. For instance, if you define a header while in 8 cpi, then print your page in 12 cpi, the header is printed on the second and subsequent pages at 8 cpi without having to reset the pitch. I know it *should* work like this but it was still a surprise to see it.

Tony Bond
Herts, U.K.

A simple way to get blanks instead of zeros in calculated cells is to set up an @IF statement that displays "NA" if a cell's value is zero. Then start up your favorite disk zap program, find the NA inside the AppleWorks program files, and change it to blanks. Of course, this means you can't use NA itself anymore, and it won't work with your data cells, but it should help. To find NA, use a disk zap program to search though the AppleWorks SEG.M1 file for the hex values 02, 4E, 41. This sequence occurs only once in the versions we've looked at. Change the 4E and 41 to 20s (blanks).

One of our relentless Australian readers, David Grigg, complained in October 1986 about the AppleWorks date format ("Of mice and macros," page 2.68), then went out and devised himself a patch to fix it. The patch is too long to print here, but we'll send a photocopy along to any subscriber who asks for one before we lose it. Australian readers can send a stamped self-addressed envelope directly to Grigg at 1556 Main Road, Research, Victoria 3095.

We thought finding the currency symbol would be easy until we discovered that the ASCII code for "$" also happens to be the oft-used 6502 BIT instruction. However, by process of elimination, Dennis figured out that if you look through SEG.M1 for the hex sequence A9, 24, 9D you'll find the byte you're looking for. Change the $24 to a $23 to start seeing pounds sterling.

We've always had right justification on our wish list, too.

Hard hardware questions

I have just started using *VIP Professional* and I am impressed—except that I end up seeing purple after a long session with the program and my green monitor. I think *MultiScribe* is very considerate in providing an invert screen function. Is there something else I can do to invert the screen short of buying a color or amber monitor?

John Mamutil
Westmead, NSW

A few months ago my second disk drive started crashing every ten minutes or so. Considering static electricity as a cause, I used an ohm meter to see if the chassis of the drive was grounded. Well, low and behold, the great thinkers at Apple hadn't seen fit to provide the drives with an earth ground. I tied a small wire to one of the screws on the bottom of each drive, and tied these to the chassis of my Apple II Plus, which does have an earth ground. Since that time I've never had an unexplained disk crash. Does the preceding sound logical to you, or just the rantings of an old farmer?

Leonard Simas
Hanford, Calif.

I have an Apple IIe, about four years old, which I use almost exclusively with *Apple Writer*. Usage varies quite a bit but typically I'll have it up and running one or two times a day for 30-60 minutes. What are the pros and cons of leaving the system on 24 hours per day?

William J. Mills
Heiskell, Tenn.

Can RGB cards or monitors bought in the U.S. be used in Europe? I find the cassette connections on my IIe quite unutilised. It there any way of putting them to real good use?

In my opinion the Snapshot card with *Shuttle* software from Dark Star Ltd in England is the best thing that has happened to the Apple after RAMWorks. It allows one to divide memory into four workspaces and then load in four different programs at one time. You can switch between them by simply pressing a button—no rebooting. You restart from exactly the point you left. I regularly use a 551K desktop AppleWorks, Copy II+, Beagle Graphics, and a chess program all in the RAM and switch between them at will.

Dr. S.S. Datye
Akureyri, Iceland

Neither Dennis nor I know enough about electronics to answer any of these questions, so Dennis called Apple II hardware guru Jim Sather, author of Brady Book's ***Understanding the Apple II*** *and* ***Understanding the Apple IIe*** *and asked him.*

In regard to inverting the monitor picture: Sather said the easiest solution is to get a monitor with inversion capability built in. A second solution would be invert the video signal in the Apple before it is combined with the sync signal, but this would require motherboard modifications that are too complex to get into here. A final solution is to build a black box that strips the sync signal off of the video signal, invert the remaining video signal (keeping the amplitude the same), then re-combining the inverted signal with the sync signal. We will leave that as an exercise for the reader.

In regard to the lack of earth grounding on Apple disk drives: Sather said there is no reason that the

drive chassis should have to be grounded, as evidenced by the 2 million plus Apple II owners who have survived the design without problems. But the grounding may solve a problem not necessarily connected to the Apple itself, such as a radio-frequency signal from an outside source corrupting signals as the disk head attempts to transfer data. Grounding the Apple drives may cure the problem by reducing the strength of the interference as seen by the read/write head. Sather said that if you have problems with the hard drive, it certainly would make sense to try the same solution, but suggested that you double check with your hard drive manufacturer's technicians first to see if they have any thoughts on the matter.

In regard to leaving your computer on 24 hours a day: Dennis and I had both read somewhere that that's what Woz does, and if it's good enough for him it suits us, too. If you leave your computer on all the time it will stay at a constant (if somewhat warm) temperature rather than going through daily warm-up cool-down cycles. Leaving it on avoids a power surge when you turn the machine on, leaves your machine always ready to use, and, if you are a big RAMdisk user, avoids the daily delay while you load the RAMdisk. Sather disputed most of that. His point was that modern digital equipment is very reliable and is built to withstand temperature cycling. Leaving the system on 24 hours a day increases the probability to 100 per cent that if any surges or spikes occur they will happen while the system is on. There is also the possibility that constantly warm components may age faster. In summary, we judge this to be one of those rare and wonderful situations where you're not damned if you do and you're not damned if you don't.

In regard to the world-wide compatibility of RGB interfaces and monitors: Sather said that there is an outside chance that you could use the U.S. RGB interface and monitor with a European Apple IIe motherboard. It depends on whether the 60 cycle U.S. equipment can sync to the 50 cycle video signal of the European Apple. The only way to know for sure would be to try the set-up and see if the video is stable. Somewhere we probably have a subscriber who has tried to cross-pollinate the video in such a manner, so I expect to know more about this soon.

In regard to gadgets that use the Apple II cassette port: such items are rare, but they do exist. The Cauzin Softstrip reader is an example of a device that will hook to the IIe cassette connector. Dennis also remembers a letter from a subscriber who was working on a network that used the cassette connections, but the details escape us.

In regard to the Dark Star Snapshot/**Shuttle** combination: I've seen it. It works. I like it. Dark Star's address is 78 Robin Hood Way, Greenford, Middlesex UB6 7QW, England 01-900-0104, Source Mail BCJ 456.

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**

Love's labor's lost?

When you are forced to work on a IIgs in native mode for three months, with all the ponderous overhead of the Apple Programmer's Workshop and the toolset intricacies, you begin to see that whatever the IIgs might be, it isn't a proper successor to the Apple II. Since I discovered the Apple II in mid-1978, *nothing* has turned my head; I've been faithful (except for some C-64 micro "affairs") ever since. Waiting (as we all have been) for what Apple Computer Co would do with the line. Cheering for the 65816 and snickering at all the Mac-nuts out there. "Just wait," we said, "the new II will knock you on your keister!"

Gotta tell ya. The emperor has no clothes. This is the end of the line for the II.

Or is it?

Seems I've fallen in love again. This time with another "II." Yeah, the Mac II. Can't afford one. Can't even afford a book about one. But check those specs!

The easiest way I know for old Apple II nuts to hear what I have to say to shut your eyes and pretend that the Mac II has a souped up 65816 screaming along at 10 MHz. When we finish the journey, it's fair to open our eyes and admit it doesn't.

* open architecture (96-pin Nubus)
* graphics-based operating system
* graphics that are completely extendible with any application software changes.
* gobs and gobs more!

Ten years ago, the Apple II blew the doors off *all* the other designs of that time. *Because it was extendible!* Because Woz *foresaw* the natural design/learning/cost reduction curves coming. Do you still think the IIgs is a fitting successor? *I* don't.

Brooke Boering
Vagabondo Enterprises
Aptos, Calif.

In its own class, the IIgs blows off my doors for pretty much the same reasons the Apple II did. The IIgs is the first Apple II since Woz's original that was designed to take advantage of future developments. Future developments such as 1-megabit and 4-megabit memory chips. Future developments such as optical disks and CD-ROMs. Future developments that haven't been designed yet but that will plug into one of those eight slots.

Beyond that, however, the old Integer Basic demonstration program APPLEVISION runs on the IIgs. Almost every trick we've learned over the past 10 years still works on the IIgs. The IIgs is a tool that real people already know how use to make a difference in their lives. What makes the IIgs magnificent (and a fitting successor in the Apple II line) is that it encompasses both the past and the future. We don't have to give up the power of the old while we learn how to harness the new.

*Your real complaint is with the IIgs programmer's workshop (the set of system software used to write "native mode" or "16-bit" programs for the IIgs) rather than with the IIgs itself. The programmer's workshop is typical of the system software we've always gotten from Apple—fairly powerful but slow and none too friendly. Think of the difference between Apple's assemblers and third-party assemblers. Think of the difference between Applesoft and Applesoft with **GPLE** or any of the other good line editors. Think of the difference between FILER and CONVERT and **Copy II Plus.***

I think that to be fair we really have to give both Apple and third-party systems software developers more time to show us what they can do. When the Macintosh had been out for as long as the IIgs has been there wasn't even an assembler available for it—you had to have a Lisa to program the thing. The IIgs is far beyond that, but still not near where it'll be in a year or two. The ProDOS 16 it has today is really just ProDOS 8 wrapped in a protein coat that looks like ProDOS 16. The engineers who write languages and the engineers who write toolbox routines are still trying to figure out where today's bugs are. Things and people need time to develop.

*I do agree that **super-high-speed** super-high-resolution graphics don't seem to be a strong point of the IIgs. For people willing **to pay** for super-high-speed and resolution graphics, the Mac II is a superior machine. But you can buy two IIgs's for the price of similarly equipped Mac II and still have more than $1,000 left. ($2,164 for a IIgs with one 3.5 drive, color monitor, and 1 Meg of RAM; $5,396 for the Mac II with the same equipment.) The Mac II color monitor alone costs as much as the IIgs system unit ($999). You are comparing two machines that are in entirely different price-performance categories.*

*The IIgs, like its predecessors, is a **personal** computer. The Mac II, like the Lisa (much like the Lisa, in fact) and its other predecessors, is an **institutional** computer. The revolutionary thing about Wozniak's Apple II was that it took the power of computers out of the hands of institutions and gave it to individuals. To Girl Scout badge moms and Junior Softball coaches and sixth graders entering science fairs. To people who can't walk or who can't see. It liberated scientists from institutional programmers, it liberated executives from "no-can-do" data processing departments. The IIgs is part of that tradition. Programs as diverse as AppleWorks and **Fire Organ** run on it right now. Within eighteen months, non-professionals will be using new languages that take full advantage of the machine's powers. Within a few years, revolutionary, yet undreamed of software will make this machine a legend.*