Open-Apple™

October 1987
Vol. 3, No. 9

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

Releasing the power to everyone.

Control-I(nterface) S(tandards)

Computers work best in an environment of standards and compatibility. If all computers, interface cards, and printers were exactly the same, no one would ever have a problem with mysterious line-breaks in the middle of a master's thesis.

On the other hand, if everyone always adhered to standards, and the standard for Apple II printers was based on the technology available the day the Apple II was introduced, we would still be using 72-column, upper-case-only teletype printers with our Apples. Printing graphics would be a dream, printing in various character widths or proportional type impossible, and no one would even consider using a computer to print a master's thesis.

Progress dooms us to incompatibilities and to knowing far more about how computers talk to printers than we really want to know. However, if every computer and every printer spoke a different language, every human at every keyboard would need to be an expert translator to get them to talk to each other. To avoid this situation we have standards. The more detailed they are, the better. The more discussed they are before they are finalized, the better. The more we follow them, the better.

Better standards are needed in many, many areas of the kingdom. This month we're going to examine just one tiny little area in detail to see what's involved. This area is one we've discussed here several times in the past—serial and parallel interface "firmware." This is the stuff your software talks to when you tell it to print something on your printer or send something through your modem. It's built into your serial or parallel interface "cards," if you have them, or into your serial "ports" if you're using a IIc or a IIgs.

There are a number of standards already in effect for serial and parallel interface firmware. Some details of these standards have even been published clearly and frequently enough for anyone to follow. Other details become obvious as one works with interfaces. Other details are neither obvious nor published (or have been published murkily), and it is in the vicinity of these details that most incompatibilities arise.

At the most elementary level, the standards that must be followed by all devices in Apple II slots are these. Each slot (except "slot 0") has 256 bytes of memory space that can be used for firmware. "Firmware" consists of machine language instructions for the Apple's microprocessor that tell it what to do to get the card to work. It's the same thing as "software," except that it's been permanently written into chips that are built into your interface card or computer.

These 256 bytes, known as the "slot ROM space," appear in the address range $Cs00 to $CsFF, where "s" is the slot number of the card. For example, the 256 bytes for slot 1 are at $C100 to $C1FF. (The dollar sign indicates a hexadecimal number. Hexadecimal numbers have 16 possible digits that go from 0 to 9, followed by A through F. $0000 is the lowest possible four-digit hexadecimal number, $FFFF is the highest. I'm using them here because they're easier to remember in this context than their decimal equivalents.)

In addition to the 256 exclusive-use bytes devoted to each slot, there are 2,048 additional bytes, in the address range from $C800 to $CFFF, that all the slots can share. This is known as the "expansion ROM space." The usage rules for this area are that whenever address $CFFF is used, all cards are to turn off their $C800 memory. Whenever an address within a card's 256-byte space is used, that card alone is to turn its $C800 memory on. Thus, a card that wants to use its own $C800 memory merely needs to tickle $CFFF and JMP to the $C800 space. The JMP instruction itself, which will be in the 256-byte space, will turn the card's $C800 space back on after the $CFFF turned it, and everyone else's, off.

In addition to space for firmware, each slot also has the exclusive use of 16 bytes for accessing hardware "registers" and "softswitches" and 8 bytes of RAM memory for remembering things.

The 16 hardware bytes, known as "I/O space," appear at $C080+s0 to $C080+sF, where "s," again, is the slot number. Thus, the I/O space for slot 1 is at $C090-$C09F.

The 8 bytes of RAM, which reside in an area known as the "screenholes" (because this RAM is inside the range of memory used by the Apple display screen), are at the following addresses—$478+s, $4F8+s, $578+s, $5F8+s, $678+s, $6F8+s, $778+s, and $7F8+s.

These standards have existed since the first Apple II. They are documented in detail in each of Apple's technical reference manuals. Even so, one detail of even these elementary standards was broken by Apple's own Super Serial Card. It is an unintentional bug, but it demonstrates how difficult it is for developers to follow standards in every detail, even when they want to. More on this later.

One of the first interface cards designed for the Apple II was the Parallel Printer Interface Card. The firmware for this card, which was written by Steve Wozniak in 1977, used only the 256-byte slot ROM space.

The card was designed to be used with Integer Basic, from the Monitor, or with assembly language. To turn the card on from Integer Basic, you entered the PR#s command (where "s," again, indicates which slot the card is in). This command changes a location known as the "output vector" so that when anything is PRINTed, control of the computer will pass to the firmware on the card rather than to the built-in "video" firmware.

After a PR#s, the Apple's microprocessor executes the first byte of firmware on the card (at $Cs00) the next time something is printed. Wozniak called this the "default entry," because starting at this address caused the card to reset itself to its "default" settings (it "initialized" itself). While setting the defaults, the card also made an additional change to the output vector so that succeeding calls to the card would go to byte $Cs02. Wozniak called this the "normal entry." Later, this scheme of passing control to firmware became known as the "BASIC Firmware Protocol."

A character sent to the card was held until the printer was ready to accept it. Then it was sent to the printer. After that it was sent to the video firmware. The card automatically added a "linefeed" (a control character that tells a printer to advance the paper one line) after each carriage return. Unlike typewriters,

"GARY AND SOME OF HIS FRIENDS WANTED TO BOB FOR APPLES THIS YEAR. I GUESS IT CAN'T HURT AS LONG AS THEY'RE NOT PLUGGED IN."

many printers then (and now) used "carriage return" as a signal to move to the left margin *without* advancing any paper. Thus, Wozniak's card added linefeeds so that what was being printed wouldn't end up all on one line.

However, some printers then (and now) automatically advanced the paper one line when they saw the carriage return signal. These printers would print double-spaced with the parallel card (advancing one line because of the carriage return and another line because of Wozniak's linefeed) unless there was some way to tell the card ***not*** to send linefeeds.

To accomplish this, Wozniak decided to make his card watch for an "escape character" in the incoming character stream. If the card saw that character, it would know that what followed was a command it should execute, rather than letters it was supposed to pass on to the printer. The escape character the card was taught to watch for was control-I (interface?). And a control-I followed by a K told the card to stop sending (kill?) linefeeds.

Problem. Control-I is used by some printers as a "tab" command. If the interface card thinks each control-I is meant for it (and consequently refuses to send it on to the printer), how do you get printer tabs to work? Wozniak solved this problem by allowing you to change the control-I to any other ASCII control-character by sending the new escape character right after the old one. "Control-I control-A," for example, changed the escape character to control-A. "Control-A control-I," or *reinitializing the card* (by using PR#s or another means to restart at the $Cs00 entry point), changed it back again.

Problem. Lines sent to the printer broke at column 40, just as they did on the screen, even though printers could handle wider lines. In order to get longer lines to the printer, it was necessary to turn the video (also known as "screen echo") off. The command Wozniak decided to use for this was "control-I nN," where the small n was a decimal number that told the card where you *would* like an automatic carriage return, since column 40 isn't it. Another command, "control-I I," or *reinitializing the card with PR#s*, put things back like they were—video on with line breaks at column 40.

Because it was first, Wozniak's Parallel Printer Interface Card set a lot of standards that have never been officially written down anywhere. But most interface cards for the Apple II have honored the "traditions" of Wozniak. Examples of these traditions include using control-I as the escape character, having the BASIC entry point of the card change the output vector to a "normal entry point," and using K, nN, and I as the kill linefeed, turn off video and set line width, and turn on video and reset line width commands.

The next major standard-setting card to appear was Apple's Super Serial Card. The firmware for this card was finished in January 1981. There were some other serial cards from Apple before the Super Serial Card and some short-lived interface-card standards (Pascal 1.0), but nothing memorable.

What made the Super Serial Card a big deal was that it was the first card to use what Apple calls the "Pascal 1.1 Firmware Protocol." The name is unfortunate. I've gotten on my soapbox before about how important it is for engineers to work as hard on the names they give things as on other aspects of a device's design (August 1987, page 2.49-50). The only link between Pascal and this interface is that Pascal was the first software to use it. However, because of the name, most programmers think of it as something that's useful only in a Pascal environment.

Although the two firmware interfaces used on the Apple II are known as "the BASIC interface" and "the Pascal interface," and although Applesoft uses the BASIC interface and Apple Pascal uses the Pascal interface, programs written in assembly language can use either one. With the addition of a small assembly language "driver," an Applesoft program can use the Pascal interface quite easily, and vice-versa. There is no unbreakable bond between the two interfaces and the two languages. (I've heard rumors that some Pascal firmware uses some of the same "zero-page" locations as Applesoft, which would make the firmware incompatible with Applesoft programs. I haven't yet found any, however, so I'll appeal to readers who have encountered this to let me know the details.)

In order to make it as clear as possible that *any software* can use either interface, here in ***Open-Apple*** I will refer to the "BASIC Interface" as the "Basic Interface" and the "Pascal 1.1 Interface" as the "Advanced Interface."

The manual that came with the Super Serial Card documented the Advanced Firmware Protocol in great detail. (The Super Serial Card is still in production but its manual has been sanitized and no longer contains any of this information. Nowadays you can find most of the advanced interface documentation in the IIe and IIc technical reference manuals and the IIgs firmware manual, but none of these includes all of the details found in the 1981 Super Serial Card manual.)

The Advanced Firmware Protocol provides a set of alternate "entry points" to the interface card firmware. These entry points provide finer control over the card than the basic ($Cs00) entry point. There are three separate entry points for initializing the firmware, for writing to the device, and for reading what the device has sent to the Apple. In addition, a fourth entry point for determining the "status" of the device (has it sent us a character? is it ready for us to send it another character?), a fifth entry point for controlling the firmware, and a sixth entry point for determining whether the firmware's card was the source of an interrupt were provided by the standard. We'll see how to use these entry points in a moment.

The advanced standard also provides for "device identification." Four bytes are used to identify what a card is used for and whether it supports the advanced standard. These bytes are:

Advanced Firmware Device Identification Protocol

hexadecimal address	hexadecimal value	decimal address	decimal value
$Cs05	$38	49157 + (slot*256)	56
$Cs07	$18	49159 + (slot*256)	24
$Cs0B	$01	49163 + (slot*256)	1
$Cs0C	$ci	49164 + (slot*256)	(device signature byte)

The fourth of these locations is called the "device signature byte." While I've purposefully limited this article to serial and parallel I/O firmware, the Advanced Firmware Protocol is available for most Apple II character-oriented devices. These include the 80-column screen firmware, the keyboard, and the AppleTalk network, as well as serial and parallel cards. Some third-party clocks, speech and sound cards, and slot-resident modems may also support this protocol. (The Apple IIgs clock and its sound chip, however, are accessed through IIgs Toolbox routines rather than through slot-based firmware protocols. This is because there just aren't enough slots available—each device requires a separate slot under the basic and advanced firmware protocols.)

Incidentally, the other kind of device you'll find on an Apple II is called a block-oriented device. An example is a disk drive. On the Apple II, block-oriented devices use a different firmware protocol, known as "Smartport," which was described in ***Open-Apple*** in January 1987, pages 2.89-93.

The first hex digit of the Advanced Firmware Protocol's device signature byte identifies a device's class. Defined signatures are:

$00	reserved	$60	clock
$10	printer	$70	mass storage
$20	mouse or joystick	$80	80-column card
$30	serial or parallel I/O	$90	network or bus interface
$40	modem	$A0	other (none of the above)
$50	sound/speech	$B0-$F0	reserved

The second hex digit is a "unique identifier for the card, assigned by Apple Technical Support," according to the original Super Serial Card manual (page 51). According to the technical reference manual for the IIe, it is a "unique identifier for the card, used by some manufacturers for their cards" (page 144). According to the IIc technical manual, it is "an identifier (not necessarily unique)" (page 67).

The Super Serial Card's definition for this byte was a nice try, but obviously the standard didn't allow for enough possible combinations—there have been far more than 16 manufacturers of "serial or parallel I/O" cards during the history of the Apple II. The IIc manual's comment reflects the truth of the situation today.

According to the Super Serial Card manual, byte $CsFF is to hold the "firmware revision-level" (page 57). On the Super Serial Card, this byte holds an $08. Apple has never revised the Super Serial Card firmware, so this is the only value you'll find here on a true Super Serial Card.

However, Apple *has* produced several Super Serial Card clones. One was built into the original IIc, one into the 3.5 IIc (the IIc that supports 3.5 inch disk drives), and one into the IIgs. (Actually, there are *two* Super Serial Card clones built into each IIc and IIgs, one connected to port 1 and one connected to port 2.)

According to the Advanced Firmware Device Identification Protocol, each of these is a Super Serial Card, although, as we shall see later, there are many subtle (and not so subtle) differences between the four versions. One of the subtle differences is that Apple's IIc programmers forgot about the firmware revision-level byte at $CsFF—both versions of the IIc have a $00 in that byte. On the IIgs, on the other hand, a new IIgs protocol makes version numbers two bytes long (but there's a bit of confusion among Apple's engineers about how to implement them—see "Machine code version numbers" in this month's letters section). The original IIgs ROMs have $00 $10 at $C1FE-FF

and at $C2FE-FF. This should be taken to mean version 1.0. The new IIgs ROMs have $10 $10 (version 1.1) at both locations. (A few bugs were fixed in the serial firmware in the newer IIgs ROMs. The bugs had to do with status calls, Applesoft tabbing, and buffering problems when baud, data format, or parity were changed with buffering enabled.)

To my knowledge, Apple has never defined a way to tell its four (five if you count both IIgs versions) "Super Serial Cards" apart. For most programmers this is only part of the problem, as many other companies have developed serial and parallel I/O devices that are subtly (or grossly) different from the Super Serial Card "standard." Until I hear of something better, I suggest you use the following "signatures" for identifying serial and parallel I/O cards that follow the advanced protocol:

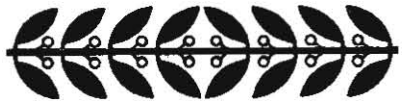
Signatures for identifying specific serial/parallel firmware

A) $Cs05=$38; $Cs07=$18; $Cs0B=$01; $Cs0C=$3x (x can be anything)

B)

	$Cs0D-$Cs13	$CsFE-FF
Super Serial Card	8E 94 97 9A 85 27 B6	C5 0B
IIc (original, port 1)	E4 EE F6 FB 0A A2 C1	00 00
IIc (original, port 2)	11 13 15 17 B0 01 B0	00 00
IIc (3.5 ROMs, port 1)	9E AB B4 BB 0A A2 C1	00 00
IIc (" , port 2)	11 13 15 17 B0 BB B0	00 00
IIgs (1.0 ROM, ports 1&2)	45 46 47 48 00 14 00	00 10
IIgs (1.1 ROM, ports 1&2)	45 46 47 48 00 14 00	10 10

Obviously, it doesn't take all eight bytes to tell Apple's versions of the Super Serial Card apart, but using all eight should allow us to tell *any* serial or parallel I/O card from any other. (Send us the values you find in these bytes on your own third-party I/O cards and we'll document them in a future issue. To avoid transcription errors, turn on your printer, enter the Monitor (CALL -151), and type Cs00.Cs13—replacing the "s" with the slot number of your card—and then CsFE.CsFF. Then type in everything you know about the type of card, manufacturer, and version number and send us the piece of paper.)



Once you've confirmed that a card follows the advanced protocol, it's easy to use the advanced interface from assembly language. The area at $Cs0D-$Cs13 that we just used for a firmware "signature" is actually a table of entry point addresses. For example, the value at $Cs0D tells you the entry point for the initialization routine. Looking at our table of signatures, you'll see that the value in $Cs0D on the Super Serial Card is $8E. Consequently, the place to go to initialize that card, using its advanced firmware, is $Cs8E. On the IIgs, on the other hand, the initialization routine starts at $Cs45, as can be seen in the signature (entry point table) for that machine.

When calling the advanced firmware, you pass information to it (such as a character you want sent to your printer) in the microprocessor's registers. It passes information back (such as a character that came in over your modem) the same way. The following table shows which offset to use in the table of entry point addresses and what is passed in the registers for each advanced firmware function:

Advanced Firmware Entry Point Protocol

cmd	offset	A reg	X reg	Y reg	carry
Init	$Cs0D				
	on entry	---	$Cs	$s0	---
	on exit	---	error code	---	---
Read	$Cs0E				
	on entry	---	$Cs	$s0	---
	on exit	char read	error code	---	---
Write	$Cs0F				
	on entry	char to write	$Cs	$s0	---
	on exit	---	error code	---	---
Status	$Cs10				
	ready for output?				
	on entry	$00	$Cs	$s0	---
	on exit	---	error code	---	0=N 1=Y
	has a character been received?				
	on entry	$01	$Cs	$s0	
	on exit	---	error code	---	0=N 1=Y
More?	$Cs11				
	if value at $Cs11=0, then following commands are also supported:				
Control	$Cs12				
	Mouse				
	on entry	mode cmd	$Cs	$s0	---
	on exit	---	error code	---	1=error
	IIgs ''extended interface''				
	cmd list adr	low byte	hi byte	bank	---
	on exit	---	---	---	---
Intrupt	$Cs13				
	Mouse				
	on entry	---	---	---	---
	on exit	---	---	---	0=Y 1=N

If you are writing firmware that includes the advanced interface, you should attempt to bring the A and Y registers back unchanged. If you are writing software that uses the advanced interface, you should assume that those registers come back looking ransacked. This maximizes the compatibility of your firmware with other people's programs and vice-versa.

If you are writing firmware that includes the advanced interface, you should be very conservative in your use of zero-page memory. The advanced interface on the Super Serial Card uses only byte $26 (for remembering $s0), byte $27 (for remembering the input/output character), and bytes $2A, $2B, and $35 (for temporary manipulations). None of these locations conflict with Applesoft per se, although $26 and $27 are also used by Monitor routines that display low-resolution graphics. The basic interface on the Super Serial Card uses several additional zero-page locations to exchange information with the Monitor—byte $24 (CH), $28-$29 (BASL), $36-$37 (output hook), $38-$39 (input hook), and $4E-$4F (random number seed).

Earlier in this article I mentioned that the Super Serial Card fails to follow one of the details of the protocol for slot-based devices. That detail is that the advanced interface does not tickle $CFFF before jumping to routines in the expansion ROM space at $C800. This went unnoticed for years because the Super Serial Card was able to overpower other cards that still had their expansion ROMs turned on. With the appearance of the 3.5 disk controller, however, the bug was revealed. Because of this, it is necessary to tickle $CFFF yourself just before you use the advanced firmware on the Super Serial Card (and it doesn't hurt to tickle it before using other cards as well).

Another subtle problem arises because the original protocol for slot-based devices has been slightly enhanced in the last few years, but news of this enhancement never made it into the manuals or into the firmware that had already been written. This enhancement is that any card that turns on its expansion ROM space should store its slot number (in the form $Cs) in the screenhole byte at $7F8 (known as MSLOT) *before* turning on the expansion ROM. This is so that the operating system can know who owns the expansion ROM space when an interrupt occurs. (As a result of an interrupt, the operating system may have to use another card's expansion ROM; it must then turn the executing program's ROM back on before returning from the interrupt or the system will have a stroke and die.)

The Super Serial Card does use MSLOT, but not until after it is executing in the expansion ROM, so there is always a brief interval when an interrupt could cause a fatal hemorrhage.

As a result, programmers should be careful to take the following steps:

```
software authors:
   just before calling the advanced interface, do something like this:
         LDA ????    character out or status request number
         LDX slot.number in the form $Cs
         LDY slot.number in the form $s0
         STX $7F8    (MSLOT)
         STX $CFFF
         JMP (entry.point)

firmware authors:
   if you use the expansion ROM, execute the following instructions
      in the slot ROM space before jumping to $C800 or beyond:
         STX $7F8    (MSLOT)
         STX $CFFF
```

I'll include a longer assembly language example of how to use the advanced firmware interface next month (or so).

In addition to problems with $CFFF and MSLOT, several details of the Advanced Firmware Protocol were neglected in the Super Serial Card manual. As a consequence, those details are not handled consistently by Apple II firmware and their value has been lost.

One missing detail was a list of possible error codes that could be returned in the X register. I scanned pages and pages of Apple documentation and found only one reference to this value. That was in the description of the Apple video firmware in the IIc technical reference manual (page 110). It says that if the value in the A register is not $00 or $01 for a Status call, the interface "returns with a 3 in the X register (IOResult=ILLEGAL OPERATION); otherwise it returns with a 0 in the X register (IOResult=GOOD)."

The term IOResult smelled of Pascal, so I asked Dennis to see if he could find anything pertinent to the X register in Apple's Pascal documentation. He succeeded. "All drivers must pass back a completion code in the X register corresponding to the table on page 280 of the 1.1 *Apple II Apple Pascal Operating System Reference Manual*," according to a little ditty called the "ATTACH-BIOS document for Apple II Pascal 1.1," dated January 12, 1980. The table of completion results lists 18 codes, all of which are disk errors except for $00, no error; $03, illegal operation; and $40, device error.

However, after studying the actual source code of the Super Serial Card firmware (it's in the original manual), I discovered that it can return 16 additional X register error codes (between $20 and $2F) not mentioned in either the Pascal or Super Serial Card manuals. The Read call and the has-a-character-been-received Status call come back with a byte that looks like this:

```
Meaning of Super Serial Card X Register Errors, by bit

0 0 e 0 c o f p

    e is an overall error indicator; if e=1 at least one other bit also = 1
      if c=1 the carrier was lost during the last receive operation
        if o=1 (overrun) the Apple isn't collecting data fast enough
          if f=1 (framing) not enough stop bits were received
            if p=1 a parity error occurred
```

On the Super Serial Card, the Init call, the Write call, and the are-you-ready-to-send Status call, on the other hand, always come back with a zero, or "no error," in the X register. The ILLEGAL OPERATION error talked about in the IIc manual isn't supported (if you give the Super Serial Card an *odd* number in the A register on a Status call, it assumes you sent a one; if you give it an *even* number, it assumes you sent a zero).

Because of poor initial documentation of this aspect of the advanced firmware protocol, all of its value has been lost. I doubt there are half-a-dozen commercial programs that even look at the X register after making an advanced interface call. Even Apple's own Super Serial Card clones on the IIc and the IIgs don't follow the X register aspect of the original Super Serial Card. They support only the error on Status calls. All other functions return a zero in the X register. Interestingly, on bad status calls the IIc and IIgs serial ports return the $40 DEVICE ERROR code, rather than the $03 ILLEGAL OPERATION code that the video firmware returns for this mistake.

Apple's mouse card, which *appears* to support the advanced firmware interface, returns the $03 ILLEGAL OPERATION code in the X register for *any* Init, Read, Write, or Status call. These calls do nothing to the mouse, which is actually controlled by a number of other firmware entry points.

Even in the face of all these differences, I recommend that programmers using the advanced interface check the X register for errors, particularly on Init calls. If a non-zero value is returned by an Init call you should treat the interface as unusable.

Other missing details in the Super Serial Card documentation had to do with the Control and Interrupt functions of the advanced interface. The original Super Serial Card Manual only said that if byte $Cs11 was zero, two optional entry points followed — the first for a "control routine" and the second for an "interrupt handling routine." Because of the lack of documentation, there is no uniformity whatsoever in how devices use these entry points.

There are three devices I know of that use at least one of these functions. One is the Appletalk firmware built into the Apple IIgs. It has an offset for the Control function, but it has a zero in the offset for the interrupt function (which has to mean there's nothing there — if you called that address you'd go to $Cs00, the basic entry point). Apple's IIgs firmware documentation doesn't even smile in Appletalk's direction, so I have have no idea what the Appletalk Control function does or how it works.

The second device that uses the control entry point is Apple's mouse. The mouse is controlled by nine routines that can be found by looking in an offset table just like the one that the advanced firmware interface has. The first two bytes of this table purposefully overlap the Control and Interrupt offsets of the advanced firmware interface. The Control offset points to a routine called Setmouse, which turns the mouse hardware on and off and tells it what kinds of interrupts it should generate. (This offset is not to be confused with Initmouse, which sets defaults and synchronizes mouse interrupts with the video's vertical blanking interval, and which, for reasons unknown, does not use the Init entry point.)

The Interrupt offset points to a routine called Servemouse, which puts a zero in the microprocessor's "carry" bit if the mouse has generated an interrupt signal. An interrupt signal is kind of like a scream in the night. It tells the microprocessor to immediately stop what it's doing and go to the aid of a device. But the only way the operating system can figure out where the scream came from is to knock on the door of every device connected to the computer until it finds one whimpering for attention. The Servemouse routine provides an easy way for interrupt firmware to figure out if it's the mouse that's squeaking.

The third device that uses the control entry point is the serial firmware on the Apple IIgs. Like Appletalk, this firmware has $00 in the interrupt entry point.

The control entry point in the IIgs serial firmware uses a completely new scheme for passing data. Rather than using the registers for data, as in all other advanced firmware calls, the IIgs serial firmware uses the registers to pass a *pointer* to the data. The low byte of this pointer goes in the A register, the "high" byte in the X register, and the bank number in the Y register. (Although the data for the command can be placed anywhere in IIgs memory, the microprocessor should be in 6502 mode when you actually pass control to the entry point. This is always the case when using either the basic or the advanced firmware interface.)

Your pointer is aimed at a "command list." The shortest command list currently in use has four bytes, the longest ten. Each command list begins with a one-byte parameter count, a one-byte command, and a two-byte space for an error code to be returned. Bytes after those four are used by the firmware to pass information to you, or by you to pass information to the firmware.

Two of the supported commands are for "mode" control, nine are for buffer control (the IIgs serial ports can buffer both incoming and outgoing data and can do printer buffering), and seven are for hardware control. It's best to avoid the hardware control commands unless absolutely necessary, since they will change on any future Apple II that uses a serial interface chip different from the one in the IIgs.

One of the mode control commands, called GetModeBits, is for inquiring about how a port is currently set up and the other, called SetModeBits, is for making changes. Both have a parameter count of 3. The command code for GetModeBits is 0 and for SetModeBits 1. Each uses an additional four bytes for the "mode bit image." Thus, the command tables for these commands look like this:

GetModsBits		SetModeBits
$03	parameter count	$03
$00	command	$01
$00 $00	result code	$00
$00 $00 $00 $00	mode bit image	$00 $00 $00 $00

The possible result codes currently supported, which you'll find in the first of the result code bytes, are $00, no error; $01, bad call; and $02, bad parameter count.

The mode bit image controls such firmware functions as whether to add a linefeed after carriage returns, whether to echo characters to the screen, and other functions that can also be controlled with escape-character commands.

Next month we'll look at the IIgs mode-bit image, and at the subtle differences between escape-character commands on Apple's various Super Serial Cards, in detail.

Ask (or tell) Uncle DOS

Apple's John Sculley is interviewed in the September ***Playboy*** *(page 51). Sculley's comments are extremely interesting, particularly his remarks about Apple as a "third-wave" company. He says he regards himself as the company's "Chief Listener." Apple is unlike the typical American company, where decisions are imposed from the top, Sculley says, and unlike the traditional Japanese company, where ideas must move up a hierarchy in a consensus-building process. At Apple, according to Sculley, "ideas can occur anywhere in the organization....top management is not predetermining company strategies." Add this interview to your reading list.*

It's pencil time, again. According to a letter Apple sent to educators, IIgs units that have serial numbers beginning with the three digits 705 through 724 need new ROMs, but not new video chips (September, front page, second paragraph). In addition, we have now heard reports of some copy-protected software not working with the new IIgs ROMs because of illegal monkeying around by the copy protection scheme. When buying software for the IIgs, ***insist*** *on un-protected products.*

The lady who owns the phone number we gave at the end of last month's ***Printrix*** *review "is pretty upset and claims she is receiving hundreds of calls," according to the people at Data Transforms, whose phone number ends with 1501, not 2502 (page 3.61).*

The first letter published below points out an error in the next to the last paragraph of my answer to "IIgs programming subtleties," in last month's issue, page 3.61-62.

Back in our August issue, at the very bottom of the very last page, we printed a patch to increase the size of the AppleWorks catalog buffer (page 2.56). It really does allow you to catalog subdirectories with more than 85 files in them. However, one of our testing laboratories has discovered that it also prevents you from loading any of them (ouch!). Put a big X over that patch and write "doesn't work" next to it. For a patch that does work, send us three good reasons to have more than 85 files in a single subdirectory (also include the box top from our September issue and a stamped, self-addressed shipping pallet).

Finally, go way back to our November 1985 Special Printer Issue and turn to page 1.83. Replace the second through the last paragraphs of the section called "The data format," with the following (you'll have to write reat small):

When characters are sent serially, the signal on the wire is kept ON (or 1, or MARK) when no characters are being sent. A sudden transition to OFF (or 0, or SPACE) is called a ***start bit*** *and indicates to the receiving device that a character is coming. After the start bit come five to eight* ***data bits****, represented by MARK or SPACE voltage levels.*

After the data bits, sometimes, is a ***parity*** *bit. The parity bit is an optional, extra bit. Parity bits, if used, can be odd, even, MARK, or SPACE. MARK parity means the parity bit is always 1. SPACE parity means it is always 0. Odd and even parity mean the transmitting device sets the parity bit to 1 or 0 in such a way that the total number of 1 bits in the data, including the parity bit itself, will be either odd or even. Parity can be used for error checking, but rarely is. In most Apple II applications, parity is set to "none," which means no parity bit is sent.*

After the parity bit, the signal on the wire goes back to ON in preparation for the start of the next character. This part of the signal is said to consist of one or more ***stop bits****.*

When no characters are being transmitted, the serial line is kept ON, or full of stop bits. Under the RS-232 standard, there is no requirement that characters must appear at equal intervals. While the start, data, and parity bits all are represented by a certain voltage level for a certain time period (the faster the baud rate, the shorter the time period), stop "bits" consist of a certain voltage level for an uncertain time period. For example, if your computer is sending out characters at a rate slightly slower than the baud capacity of the interface, extra stop "bits"—even fractional "bits"—can appear between characters. This ability to support a variable time interval between characters earns the RS-232 interface the adjective ***asynchronous****.*

The term ***data format*** *defines how many data bits each character will have, whether the character will include a parity bit (and if so, what* ***kind*** *of parity), and the* ***minimum*** *number of stop bits between each character. This minimum number is typically either 1, 1-1/2, or 2. For signalling purposes, one stop bit is enough—when extra stop bits (or parts of stop bits) are inserted it's either to give the receiving device more time to process each character coming in or it's an attempt to solve timing problems in the sending device.*

For example, the default format of the Apple IIc's printer port is 8 data bits, no parity bits, and 2 stop bits (or "8N2"). Every other Apple serial device defaults to 8N1. A printer expecting just one stop bit can receive data from a device sending two stop bits with no difficulty—it just thinks the engine in the sending device isn't operating on all eight cylinders. The additional stop bit from the sender slows things down slightly. This is apparently exactly what was expected from the 2 stop bits on the IIc's port, because the serial ports on early models of the IIc marched to the beat of a hasty drummer (see June 1985, page 1.47).

There are some nasty remarks about printer documentation in the original that you might like to keep, but the rest is a reflection of my confusion. Boy it feels good to finally understand how serial communication works! My thanks go to chapter 11, "The Serial Interface Ports," of Gary Little's ***Inside the Apple IIc.***

TLShutDown does too

On page 3.62 of your September 1987 issue you say that the IIgs toolbox command TLShutDown does nothing at present. Not true! Under ProDOS 16, TLShutDown calls the BootInit functions of all installed tools and unloads RAM-based tools (disconnects them from the tool table and leaves their memory purgable).

Incidentally, regarding the $400-byte stack/direct page block you mentioned in your response to the same letter, there is no reason that a direct "page" must be limited to 256 bytes on the IIgs. While some addressing modes allow access to only the first 256 bytes of a direct page, indexing from the direct page with a 16-bit X or Y register allows you to access all of bank 0. The actual allocation of the $400 bytes is up to the application; if no direct page is needed, the whole $400 can be used for a stack; if a different stack area is being used, the whole $400 can be used for direct-page storage.

David A. Lyons
North Liberty, Iowa

Machine code version numbers

Are you sure that Line 500 of your Smartport reader program in the January 1987 issue is correct (page 2.92)? The line reads 500 : : VERS = PEEK(807) + PEEK(808)*256.

When I run the program, I get version 4096 for my UniDisk 3.5. This seems high for the version number of a year-old device. Even if the PEEKs are reversed, a result of 16 would still be a little high.

Robert J. Schack
New York, N.Y.

I noticed the weird version number results when I wrote the Smartport article, but didn't know what to do about them. Apple's Smartport documentation said only that those two bytes were the version number, but gave no information about how to interpret them.

Recently I ran across the following paragraph on page 28 of Michael Fischer's ***Apple IIgs Technical Reference*** *from Osborn McGraw-Hill. It refers to toolbox version numbers, not firmware versions numbers, but is still enlightening:*

"Version numbers for each tool consist of a word value. Bits 0-7 of the word contain the minor revision number, beginning with zero. Bits 8-14 contain the major revision number, beginning with one. Bit 15 is set if the version is a prototype and is clear if the version is a released version. Thus $90 is version 1.0 prototype, $12 is version 1.2 official release, and so on."

Notice that Fischer's description and his examples, which he no doubt got from some Apple documentation somewhere, don't match. According to his description, "version 1.0 prototype" should be $8100. "Version 1.2 official release" should be $0102.

If we throw away the low byte of the Smartport version number and decipher the high byte as in Fischer's ***examples****, we get the results we expect, version 1.0. Thus, to get January's Smartport program to report the UniDisk version number "correctly," we should change lines 500 and 600 to:*

```
500 : : VERS$ = STR$(INT(PEEK(808)/16) + "." +
      STR$(PEEK(808) - VAL(VERS$)*16)

600 change VERS to VERS$
```

But I suspect Fischer's ***description*** *is really the way version numbers are supposed to be used on the IIgs (otherwise, why use two bytes and then throw one away?), in which case the "correct" corrections to the Smartport program would be:*

```
500 : : VERS$ = STR$(PEEK(808)) + "." +
      STR$(PEEK(807))

600 change VERS to VERS$
```

This, of course, returns a version number of 16.0 for the UniDisk, which reflects a certain amount of confusion at Apple about version number protocol.

What PR# and IN# do

Please explain why PR#3 and IN#3 turn on my 80-column card at different times when I replace one with the other in my STARTUP program. In immediate mode both PR#3 and IN#3 turn on the 80-column card immediately.

Dave Uherka
Grand Forks, ND

While common knowledge is that PR#3 "turns on 80-columns," in fact, things are more complex than that. What PR#3 and IN#3 really do is "redirect" output or input to slot 3. The "80-column firmware" lives in slot 3. However, you won't see an 80-column display until you either PRINT something (after PR#) or ask for some INPUT (after IN#). In immediate mode, Applesoft immediately prints something (a)) and thereafter asks you to input another command, right after you type in either PR#3 or IN#3. Thus, either PR#3 or IN#3 appears to take effect immediately. To get the same effect in your programs, you must put a PRINT statement right after your PRINT CHR$(4);"PR#3" or an INPUT statement right after your PRINT CHR$(4);"IN#3".

For a much more detailed description of how this works, see Chapter 12 of the ***DOStalk Scrapbook****, "How the System Operates," pages 81-91.*

Disassembly lines

Does your remarkable ability (to me) to crack object code without any available source code stem from long experience or could you illustrate how you do it?

Are the colons you sometimes use after a line number in Applesoft program listings for indentation purposes?

I'm not sure I understand the equivalence of port 1 on the IIgs and the Super Serial Card. Can a "Super Serial Card" choice be used for configuring older programs? If it can, I presume Slot 1 should not be used. Similarly, if slot 3 is the eighty-column pathway, can it be used for a MIDI card?

Herbert M. Olnick
Mineral Bluff, Ga.

It takes some experience to get good at disassembling machine language instructions, but there are a few tricks of the trade I can describe here. First, you need a disassembler. This a piece of software that scans the values in a series of bytes and tells you what they mean in assembly language. There is a rudimentary disassembler built into every Apple II—the Monitor's L(ist) command. More advanced disassemblers, which are available from most of the companies that sell assemblers, will send the disassembly to a file (rather than simply to the screen or a printer), will create lists of the addresses referenced, and will even plug in the names of built-in Apple II subroutines, softswitches, and zero-page locations where it appears they are being used.

In most disassemblies you aren't concerned about how the whole program works but just want to find a specific area that's troubling you to fix it. You find the area by scanning through the program looking for embedded text messages, for accesses to certain softswitches, zero-page locations, or Monitor subroutines. You can also figure out a lot by looking for ProDOS MLI calls (JSR $BF00) and, on the IIgs, toolbox calls (JSL $E10000).

If this doesn't work, or if you really want to disassemble the whole program, your next step is to load the disassembly into a good word processor and start chinking away. Whenever you find something you can identify—be it a zero-page location, a subroutine, a message, or whatever—use the search and replace capabilities of the word processor to change all references to that address to a name that makes sense. As you proceed, the search and replace procedure keeps adding clues to parts of the program that originally made no sense at all. Gradually you figure everything out (usually finding lots of little bugs and undocumented features of the program along the way). The experience is very much like working on a giant jigsaw puzzle.

The biggest disassembly I ever did was the portion of DOS 3.3 from $9D00 to $B800 (the command interpreter and the file manager). This is about 7,000 bytes of code. I had a copy of Beneath Apple DOS at my side that I used as a hint book. It took six weeks of daily work. When I was finished I knew enough about DOS, its features, and its bugs to write ***ProntoDOS, Softalk's*** *"DOStalk" column, and the* ***DOStalk Scrapbook.***

I think you'll find that many of the best assembly language programmers spend a lot of time disassembling other people's work, just as the best writers spend a lot of time reading other people's work. Of course, you don't really need to disassemble anything to "read" other people's work. Lots of assembly language source code is around, particularly in Apple's technical manuals. Apple has a tradition of publishing the source code to at least the Monitor in all of its computer-specific technical references.

*As you do more and more of this, you'll find that different programmers have very different styles. Some are ultra-organized. They have a place for everything, everything in its place, and don't give a beep how long it takes to execute. The DOS 3.3 file manager is a good example of this kind of programming. Others write code that is extremely fast, efficient, sensible, and elegant. Look at anything by Steve Wozniak (the original Apple II Monitor, DOS 3.3's RWTS routines) or Paul Lutus (****Apple Writer****) for examples. Others write code that is impossibly complicated, confused, crude, and full of bugs (the unknown author of DOS 3.3's APPEND patch, for example).*

If any of this interests you, I suggest you get a copy of Don Lancaster's ***Enhancing Your Apple II, Vol 1*** *($15.50, Synergetics, Box 809, Thatcher AZ 85552 602-428-4073). Chapter 3 of that book is called "Tearing into Machine-Language Code" and is worth the price of admission. Lancaster demonstrates what he calls the "tearing method" of disassembly. This method is paper-based and gives you a better view of a program's "big picture" than my word-processor method. On the other hand, I end up with completely commented source code in a file that I can modify and run through an assembler without any additional typing.*

I use extra colons in Applesoft listings simply to make it clear what lines are inside loops. I'm surprised that none of our readers has ever complained that ***Open-Apple's*** *program listings don't look like the ones Applesoft itself puts on your screen. Back in the early days of the Apple II this was a big issue in the letters-to-the-editor pages of Apple II magazines. I guess it means you've all figured out that the program listings in* ***Open-Apple*** *are designed more for easy reading and understanding than for easy typing or speedy execution.*

The "firmware" in port 1 of the IIgs is functionally equivalent to a Super Serial Card but the "hardware" is not. Thus, whether a "Super Serial Card" configuration will work for any particular program depends on whether the program accesses only the firmware or whether it also tries to access the hardware. Unfortunately, few programs document this kind of stuff; your only recourse is to try it and see what happens. If a program won't work through the built-in port, you can have a Super Serial Card in slot 1, change the slot 1 configuration via the control panel from "printer port" to "your card," press open-apple/control/reset, and use the Super Serial Card. Using this technique, you can switch back and forth as often as like without turning the computer off (turn it off to insert the card in the first place, however, and to change cable connections).

Slot 3 can be used in this same way, however, whenever you set slot 3 to "your card" you lose the ability to use Apple's 80-column firmware, which will prevent many programs from working correctly. But, for example, if your music software doesn't use Apple's 80-column firmware, a MIDI card might be a good choice for that slot. You would have to enter the control panel, change the slot 3 assignment, and reboot before and after running your music software, but you wouldn't have to give up some other slot.

Apple makes family grumpy

We just moved into a new house that has the TV antenna directly above my IIe. When the Apple is on, it causes video interference on channels 4 and 5.

Everything is grounded...I checked that on the suggestion of our local dealer's service person. No one locally can come up with a solution. As a result, my family gets grumpy if I use the computer when they're watching NBC or CBS. Telling them PBS is good for them hasn't helped. Do you have any hints?

Fred Olin
San Antonio, Texas

I haven't any hints. Somebody somewhere has probably figured this out, however. Let's hope they write us.

Logo IIgs

In response to "Go, Logo, Go," in your September issue, there is one version of Logo for the Apple that has already been altered to take advantage of the IIgs environment. That is *Logo II*, an upgrade of *Apple Logo II*, available from Logo Computer Systems Inc, 121 Mount Vernon St, Boston, MA 02108 800-321-5646 617-742-2990. Several purchase options are available, including single copies, lab packs, and site licenses.

Your readers who are interested in Logo might like to join the Logo Exchange. We're a group of educators interested in the use of Logo. Membership includes a subscription to our magazine. Queries can be sent to the Logo Exchange, ICCE, University of Oregon, 1787 Agate St, Eugene OR 97403 503-686-4414. A year's subscription is $24.95 for U.S. ICCE members, $5 additional for non-members, and $5 additional for international subscriptions.

Tom Lough
Charlottesville, Va.

Hard feelings

The IIgs chip upgrade you mention in your September issue is interesting. Apple Canada prides itself on being a separate entity from its mother company and seems also to pride itself on always being a month or two behind new developments "south of the border." In this age of lightning-fast communications, this attitude causes a great deal of hard feelings. As of

September 1, Apple Canada had no chip upgrade policy. I'd be curious to know if a similar situation exists in other countries.

Lorne Walton
Maple Ridge, BC

LaserWriter possibilities

I just purchased a LaserWriter Plus to go along with my IIgs. Despite what Apple says, you can get *Apple Writer 2.1* to print to the LaserWriter just fine. You need Don Lancaster's patch to *Apple Writer* that allows the program to use the new serial ports (*see "The great Tinaja Quest" in May 1987, page 3.31*). Then connect your IIgs to the LaserWriter via port 1 with a IIgs Adapter Cable (#A9M0333, $29.95 retail) and a DB25-male to DB25-male straight-through cable (approximately $20 retail).

You need the second cable because the adapter cable is less than a foot long. Don't use a null-modem cable; a straight-through cable works just fine. Make sure the LaserWriter is off and set the back panel switch to "SPECIAL." This tells the LaserWriter to emulate a Diablo 630 printer. Turn your LaserWriter on, let it warm up, and run *Apple Writer* as usual. The Control Panel slot assignment for slot 1 should be directed to the printer port. With this setup I've been able to print anything *Apple Writer* can do, including this letter.

Under this configuration, your LaserWriter is a very expensive Diablo printer; you can't reach the many special fonts, etc. To do PostScript processing, you can use the same cable and set the LaserWriter back panel switch to "9600". Then you need software to send the necessary PostScript commands. You can also connect the IIgs (or a IIe) to the LaserWriter via a Super Serial Card and the appropriate cable. Or you can connect the IIgs via AppleTalk.

I can't get *Apple Writer* to work with the LaserWriter via AppleTalk; why, I don't know. Is there any source that disassembles and describes the AppleTalk interface in the IIgs? AppleWorks does work with a LaserWriter via AppleTalk as advertised, in which case your LaserWriter is a very expensive ImageWriter.

Other programs will print to a LaserWriter connected to port 1 (or 2) as long as they can print to a Diablo-compatible printer and access the serial port or card. Some old DOS 3.3 programs, like *MultiPlan*, work just fine—I suspect *MultiPlan* accesses the printer slot or port without calling on any special firmware tricks.

A quick note concerning the *VIP* spreadsheet for the IIgs. I'm a spreadsheet junkie (I own *VisiCalc, Multiplan, Practicalc, Supercalc 3a, AppleWorks* and now *VIP*—my favorite is still, believe it or not, *Multiplan*, although *SuperCalc* has the best business-type graphics I've ever seen on an Apple). *VIP* is extensive and could give *Supercalc* and AppleWorks a run except for two problems: first, the output to printers is extremely limited. You can only choose to print to an ImageWriter or Epson printer, and then only through slots 1 or 2 (no LaserWriter, AppleTalk, pen-plotter or other printer support—not even the trusty Apple DMP!). Second, if you try to print to a non-recognized device, or if you try something else the program doesn't recognize, the whole system crashes, with *total* loss of data. Talk about being careful!

In November 1986 (page 2.79) you had a letter asking how to convert files from *Multiplan* into something that could be used by *SuperCalc*. I have written a program that will convert a Microsoft SYLK file into the more common DIF file, which *SuperCalc* can read. SYLK files and DIF files can only be used to transfer data, not formulas. I'd be happy to let anyone who sends me a disk and return postage have a copy of the program.

Finally, how do you get an AppleSoft program to run under ProDOS 16? How much memory can I access—am I still limited to 48K minus ProDOS?

Steven R. White, M.D.
411 N. Kensington Ave.
Lagrange Park, IL 60525

Anyone using a LaserWriter, and especially those using one with an Apple II, should read Don Lancaster's column in ***Computer Shopper*** *religiously ($21/yr, P.O. Box F, Titusville, FL 32781). Don has reprints of past columns for sale, as well as all kinds of* ***Apple Writer*** *goodies for making the LaserWriter do things that Macintosh programs find impossible. And he gives out free samples. (Don Lancaster, Synergetics, Box 809, Thatcher, AZ 85552 602-428-4073). You might also like to look into the National Postscript Bulletin Board at 409-244-4704 (300/1200 8N1).*

Documentation on AppleTalk as it exists on the IIgs is non-existent as far as we know. The beta draft of the ***Apple IIgs Firmware Reference Manual*** *documents all the other stuff that appears to be connected to IIgs slots, but doesn't breathe a word about AppleTalk.*

Applesoft runs under ProDOS 8 (with BASIC .SYSTEM), never ProDOS 16. AppleSoft is structurally limited to, and ideally suited for, a 48K program/variable space. Apple has no plans to modify it for the larger memory available on the IIgs or for ProDOS 16. I think they've made the right decision. By refusing to make any modification to Applesoft, Apple retains complete compatibility between older programs and new machines. Applesoft remains a universal language. Why can't we agree that any language that takes advantage of the eight-megabyte memory space available on the IIgs should be written from scratch and not be based on a 10-year-old language that uses line numbers and two-character variable names?

ImageWriter II cleaning

How do I remove and clean my ImageWriter II printhead?

Donald Bock
Hudson, Fla.

Hmmm... the ***ImageWriter II Owner's Manual*** *doesn't say in its section on maintenance (pages 63-66).*

Page 28 of the ***Imagewriter User's Manual*** *(for the original Imagewriter) has the essential procedure, except that the mechanism that locks down the print head is different on the ImageWriter II.*

First, make sure the printer is turned off. Remove the ribbon. As you look at the II's print mechanism from the front, you'll see a white plastic latch along the right side of the printhead that clamps it down. Gently push this latch toward the power-button side of the printer to clear the printhead and jiggle the printhead straight upward. If you are right-handed, you may find it easier to do this while standing behind the machine. Be careful not to let the head hit anything as it comes loose. The only cleaning procedure mentioned is to "wipe" the type head gently with a soft brush to clean away loose debris left by the ribbon or paper. Then carefully replace the head, reversing the above procedure.

Print heads are delicate and are not cheap; if you have any doubts about safely removing and cleaning yours let a technician do it for you.

Super hi-res converter

I do a lot of work in graphics. Traditionally this has involved a lot of hi-res and double hi-res files on my IIc. I used *Dazzle Draw* and was happy. But not any more. Now I'm the proud owner of an Apple IIgs, and I find myself with bushels full of *Dazzle Draw* format double-high and *MousePaint* format standard-high files that I'd like to convert to Super-320 mode, so that I can modify and resave them as *Deluxe Paint* pictures. Do you know of any utility that can do this, or of a simple machine code modification I could make to my old files to update them? It seems a waste to abandon all that old work, and redrawing it would take years.

James Waschuk
Saskatoon, SK

According to the July 1987 issue of ***Call-A.P.P.L.E.*** *(page 57), such a program is available in DL4 (Data Library 4) of the download section of MAUG on Compuserve. The program's name is SHRConvert, and it converts a number of types of graphics formats to Super Hi-res format, including Apple II single and double hi-res graphics, and Macintosh and Atari ST graphics.*

AppleWorks as copy machine

How about a patch for AppleWorks that would allow more than nine copies of something to be printed?

David L. Smith
Middlesboro, Ky.

Beagle Bros/Software Touch AppleWorks guru Alan Bird provided us with the following:

```
Patch to change the maximum number of copies to 255
   (AppleWorks 2.0 only)

POKE 768,255

BSAVE SEG.M1,T$00,L1,A768,B36074      ; for WP
BSAVE SEG.M1,T$00,L1,A768,B9185       ; for DB
BSAVE SEG.M1,T$00,L1,A768,B65895      ; for SS
```

For other versions of AppleWorks, Alan suggested searching the SEG.M1 file for the byte sequence "A9 09 20 35 D0" and replacing the 09 with a larger value. Dennis looked at AppleWorks 1.2 and 1.3 and discovered the actual sequence you have to look for in SEG.M1 is "A9 09 20 32 D0". This sequence occurs four times; change the first three (as Alan did for 2.0):

```
Equivalent 8 parameters for AppleWorks 1.2 and 1.3:

    V 1.2     V 1.3

   B34049    B34253
    B9151     B9151
   B67095    B67332
```

After the patch, you can enter any value from 1 to 255 copies directly.

Wire loose inside FILER

I've run into the same problem so many times I don't understand why I've never seen it commented in ***Open-Apple***. I usually cannot copy a disk onto a new unused disk with FILER. Until something is written on the new disk I get a NO DEVICE CONNECTED error. If the disk is first formatted with FILER there is no problem, but then time is wasted re-formatting at the start of the copy.

Even a DOS 3.3-formatted disk is acceptable as a copy target, but not a clean one. Why is this? A related problem is that quite often AppleWorks 1.2 and 1.3

will refuse to format a clean disk, with the same error message. Are they too polite to defile a virgin, or what?

Phil Albro
Cary, N.C.

I have a dim recollection from the early days of ProDOS of this problem. The reason no one talks about it is that no one uses FILER anymore. ***Copy II Plus*** *($39.95, Central Point Software, 9700 SW Capitol Highway, Suite 100, Portland, OR 97219, 503/244-5782) is a reasonable alternative that does most of what FILER and CONVERT together can do, and more, and better.*

Another alternative is Glen Bredon's ***ProSEL*** *package, which has been recommended here many times in the past ($40 from Bredon at 521 State Road, Princeton, NJ 08540). It can't convert DOS 3.3 files to ProDOS, but it includes all other disk utilities most people can imagine, plus a few more you'd never think of. (Such as a scheduler that will run programs for you, unattended, at the times you select. Early purchasers of ProSEL may be missing the scheduler and a few other updates to the package. The latest ProSEL version is always $5 for previous purchasers, directly from Bredon.)*

For AppleWorks, try changing to a different disk drive for formatting. That seems to work sometimes. Otherwise, it's wise to keep a few pre-formatted disks around for emergencies.

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.

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**

Free help

Concerning August's letter about loading non-TXT files into the AppleWorks word processor module ("Another use for AppleWorks," page 3.55)—in a perverse twist back in July I used this capability to load *Free Writer*, Paul Lutus' public domain word processor, into AppleWorks. I used the OA-Delete function to wipe out everything but the help screens so I could print a hardcopy.

Clark Stiles
Grand Rapids, Mich.

Just don't use this technique to cheat on adventure games.

More on OctoRAM

I reviewed the MDIdeas OctoRAM board mentioned by Doug McClure (August, page 3.51). Look for my review in the fall issue of *Apple II Buyers Guide*. There are eight SIMM sockets on the board. You can put in either 256K or 1 megabyte SIMMs, but you can't mix them. Consequently, if you purchase a 1 meg board with four 256K SIMMs (the way MDIdeas ships it), you can upgrade the board to two megabytes. If you want to go higher, you have to get 1 meg SIMMs, and sell your old SIMMs or let them gather dust in your junk box. At the present the SIMMs are more expensive than an equivalent number of standard memory chips, making the OctoRAM somewhat more expensive to expand and operate. Nevertheless, it's the only IIgs memory board which can be expanded to 8 Mbytes on its main board.

The ESP ROM board is an extra cost option, but it really isn't ROM. It is a piggyback board with 64K of static RAM. The board includes a rechargeable battery that powers the RAM while the power is off. Extra RAM packs are available which expand the ESP board up from its standard 128K capacity up to 512K. You can set the IIgs's control panel to boot the ESP board on start up or you can access it like any other ProDOS drive. Under normal circumstances, the static RAM takes very little power and with the rechargeable battery pack will last several years before losing data. The ROMdisk is a good idea, and the static RAM with battery backup makes it fairly easy to use. The disadvantage to static RAM is that it's extremely expensive and sensitive to static electricity. There's one major feature on the ESP that I don't like—it attaches to the front of the OctoRAM board and blocks slots 6 and 7 in your IIgs, the most important slots.

Philip Chien
Earth News
Titusville, Fla.

A tip from subscriber Peter Baum—If you buy an OctoRAM and have friends with Macintoshes, pay attention to see if they upgrade their Macs to 1 meg SIMMs (which they must do to get more than 1 megabyte of RAM). If they do, you'll have found a cheap source of 256K SIMMs, eight of which (2 megabytes worth) will fit in the OctoRAM.

The Universal Apple

I have lived in Germany for over ten years and was a manager of a local computer store for several years. We sold Apple, IBM, Osborne, Tandy, Wang, Nixdorf, and a few other brands. In regard to the letter "International Answers" in your August issue (page 3.53), let me assure you that there is no problem using any Apple hardware in Europe and there is seldom a problem even with combinations of U.S. and European Apple hardware.

I have two Apple IIe computers, a German version I purchased less than one month after Apple introduced computers in Germany, and a standard American version. The German machine was purchased with German-version monitor, 80-column card, and Apple DMP printer. The U.S. version was purchased with a U.S. monitor, text card, and ImageWriter. The U.S. version has worked perfectly in Germany for over two years on a step-down transformer.

I have added numerous items to each computer as the years have gone by, including mice, new drives, an RGB monitor, numeric keypad, graphics tablet, daisy-wheel printers, and a long list of other items. They all work just fine no matter which version they are hooked into, with the single exception of the U.S. numeric keypad—it's cord is too short for the European Apple II, which has the keypad connector located a little further toward the front of the motherboard. And I am happy to say that I have not experienced any of the screen flicker or weaving, caused by using a 60 Hertz Apple in a 50 Hertz country, that you mentioned. I also know several other people using this type of setup and have heard no complaints.

I don't recommend this for IBM users though. The U.S. version IBM monitor, when used on 50 Hz with a transformer, would wave and weave so much that my customers would get seasick and turn green. That made it real easy to sell Apples—one look at those IBM monitors and anything else would look great.

Here are a few other necessary details: travellers will need a 300 watt transformer to correct the voltage to what is needed for their system (U.S. systems need 110-120 volts, European systems need 220 volts). A transformer of this size will power the computer, a monitor, and two printers with room to spare. Remember to bring a multi-outlet dropcord from your native country since most transformers only have one or two outlets and this makes it very difficult to plug in four or five items.

Cards that plug into slot 3 will not work in European Apples since the auxiliary slot is located in front of this slot and therefore any cards plugged into the auxiliary slot will cover slot 3. I have yet to find any software that will not run on either version computer. Graphics programs, utility programs, word processors, games, and even copy-protected stuff seem to run just fine (who cares).

Leave the war games at home though. West Germany and several other European countries have banned many of these games. Some of the programs include *F-15 Strike Eagle* and a few other air simulators (not SubLogic), several submarine simulators, Rambo programs, karate programs, and some other programs that in their view glorify war or violence. Theoretically, this applies to anyone bringing these games into the country also. I say theoretically since nobody has been arrested doing so yet.

And finally, getting your computer repaired is not a problem. The service I have gotten in Europe has been excellent. However, the U.S. warranty is not valid outside the U.S.

Dwight Stewart
Leimen, West Germany

One item you don't mention that can't be mixed and matched is the normal composite monitor. Composite signals use different standards in different countries, so computers and monitors should move around the world as matched sets. We have a letter from a subscriber in Kuwait who mixed the two and "smelled money burning."