$1.00

Washington Apple Pi

PO Box 34511 Washington, DC 20034

volume 1 September 1979 Number 8

Officers

President	John Moon	(202) 332-9102
Vice President	Bernard Urban	(301) 229-3458
Treasurer	Robert Peck	(301) 770-1954
Secretary	Genevie Urban	(301) 229-3458
Newsletter Staff:		
Editor	Bernard Urban	(301) 229-3458
Associate Editor	Mark Crosby	(202) 488-1979
Program Librarian	David Morganstein	(301) 474-5768
Members-at-Large	Sue Eickmeyer	(301) 490-7627
	Sandy Greenfarb	(301) 674-5982
	Mark Crosby	(202) 488-1979

Contents

PRESIDENT'S MESSAGE

Dear APPLE Enthusiast:

As is obvious, GWU has some interest in APPLEs - they own a dozen or so of them with more on order! They use them in a number of courses, including introductory programming (such as CSCI052) as well as some of the more advanced courses. This semester I'm taking a couple of courses (and maybe someday I'll get my M.S.) that I especially arranged with the professors beforehand in order to use my APPLE for the homework and project assignments. One course is Interactive Graphics, the other is on Digital Programming Systems.

I'm hoping that as a side benefit I can use the homework problems and projects as subjects for articles in this newsletter (after all, I have to write them up for class anyway). For example, in the Interactive Graphics course, one of the key topics that will be covered is a Standard Graphics system known as the Core Graphics System. This system includes routines for windowing, scaling, transformations, viewports and all manner of other such graphics-oriented things. I hope to end up with an implementation of the system on the APPLE as well as some articles describing what all those funny words mean.

In the Programming Systems course, the two homework assignments will be to write an Assembler and a Relocatable Loader. Not a trivial assignment! But the present assemblers that I have access to on the APPLE assemble a single source file in memory and have no capability to link more than one program together - something very useful if you would like to mix and match your subroutines.

On the negative side, maybe all this is going to keep me so busy that I won't have time to figure out which end is up; well, at least it's on the APPLE so I can do it at home! (Saves energy, is tax deductable, etc. etc.)

Bernie suggested a rather wild - largescale, but fascinating idea/project for the club - to set up and run a Personal Computing Conference here in Washington. I'm staggered by the thought of it; but like I said, fascinated... Anybody interested give it some thought and see me or Bernie at the next meeting - which, by the way, has been changed to Saturday, September 29 due to a holiday conflict.

Hopefully I will have contacted Pete Kendrichs at the Source so that they will arrange a demonstration at the next meeting. If not, then I will try for October. So - at this stage, you'll just have to come to the meeting in order to find our if we've set it up or not.

See you there...

John L. Moon

MINUTES

APPLE Pi - The meeting was held in Tompkins Hall, GWU on August 25, 1979. John Moon asked if the club would like to accept an offer from The Source for a demo for the club. A resolution was passed by the club to have the demo arranged at the earliest convenient club meeting. Sandy Greenfarb moved that the next meeting date be set as September 29, 1979. This passed - the next meeting is on the 29th. Sandy then gave in impromptu discussion on the Integer BASIC Workshop that he has modified and is putting into the club library. Hersch Pilloff said that the Paper Tiger demo was still awaiting a machine. Several general discussions were held on various software items and a request was made for someone to review some of the available word processors for the APPLE. A motion was put forth to make the club policy to keep the membership-mailing list confidential. After some discussion, the motion was amended to require individual permission to release the individual's name. The motion passed as amemended.

John L. Moon

NOVAPPLE - The meeting was opened at 7:30 p.m. on August 23 by the Secretary. Several announcements were made. First, the next meeting of NOVAPPLE will be September 12, 1979 at Computerland of Tysons Corner when a demonstration will be given on the "Source". This is a time-sharing commercial service open to APPLE owners for a fee. The meeting will start at 7:30 p.m. There is a new piece of software out from Dan McCreary, PO Box 16435-X, San Diego, California 92116. It is known as Apple 80. The ads say it converts an Apple into an 8080 simulator, and also cautions that it is slower than either system since it is an interpreter program. No one had tried it yet. The cost is $20 plus $1.50 for shipping and handling.

One member brought in a copy of the disk program "Dr. Memory" for anyone to review before they purchase one.

A "goof" on my part occured in last months minutes. We will meet at Computerland of Tysons Corner on the 4th Thursday and Computers Plus of Franconia on the 2nd Wednesday. Please mark your calendars. The meeting nights are subject to change if the stores request it but for now the meeting dates appear firm.

The program was presented by Mr. Ken Woodward. He began an eight week course in assembly language. The course outline is shown below:

CONTENTS OF ASSEMBLY LANGUAGE LECTURE:

1. Introduction to Number Systems
2. Introduction to Data Codes
3. Introduction to 6502 Machine Language
4. Monitor Usage
5. Arithmetic on the 6502 Processor
6. Moving Data
7. Basic Input and Output
8. Looping Techniques
9. Bit Operations
10. Use of Psuedo-Opcodes
11. Converting BASIC programs to ASM
12. Introduction to Sweet 16
13. Stack Processing
14. Specialized Output Routines
15. High Res Graphics/Lo Res Graphics
16. Coding for Speed
17. Coding for Efficiency
18. Floating Point Routines
19. Peripheral Programming/Hardware Programming
20. Using the Disk II Assembly Programs

continued

Mr. Woodward's first session went into binary, decimal and hex numbering systems. He passed out and explained conversion tables which allow rapid conversion from one to another. He also described the microprocessor's structure. (A review of similar information is in RAINBOW, Vol 1 Issue 7, dated August, 1979.) A demonstration was performed using an Apple to display how one could work with the monitor language. You can make changes, display, give instructions, and enter data into the Apple. Mr. Woodward will pick up his lessons on September 27, 1979. He would like everyone to bring their copy of the Red Reference Manual. He will use it to provide a basis for his discussion. If you never have understood assembly language before, this is the series for you. It is well prepared and simple enough for even a beginner.

The meeting was adjourned at 9:15 p.m.

Respectfully submitted,

Gerald R. Eskelund

EDITORIAL

We've come a long way from Volume 1 Number 1 and I hope you are as pleased as I am with the changes. Do I hear three cheers for Mark Crosby's efforts? The newsletter has grown from a timid trial balloon to the present high-quality product of which I think all the contributors can be justifiably proud. However, I am concerned that some of our plans have gone astray and I would like to remind you of one of them.

I don't believe that the greater Washington area can support more than one high-quality newsletter devoted to APPLE owners and users. Human nature being what it is, only a few individuals seem inspired to come forward with articles and items for inclusion in the newsletter. To set up several newsletters within this area would dilute their efforts to the point where each newsletter would not reach what I call the "critical mass" needed to become self-supporting for the benefit of the APPLE user community. There are considerable dollar and time costs associated with getting out an issue. Economics of scale can only be realized when the contributors are drawn from an organization representing approximately 200 individuals. Incorporation (like Call-A.P.P.L.E.) as a not-for-profit organization makes us eligible for considerably lower postal rates, mass purchase discounts, etc.

I urge all recipients whether members of Washington Apple Pi or NOVAPPLE, whether paid or not, to think again about the merits or drawbacks of forming one cohesive organization covering the greater Washington area for the primary purpose of publishing a regular, monthly newsletter that is of high calibre and of genuine use to all APPLE users - neophytes to masters, young and old. Remember, each chapter, branch or whatever they may be called can maintain its own identity, geographic location and schedule.

Other items which are in danger of falling through the cracks:

- o Club position on the ethics of exchanging proprietary software.
- o Establishing a library of written materials for the benefit of all members.
- o Scheduling in-depth courses on the workings of the APPLE at all levels, e.g., fundamentals of programming, PASCAL emulation of Z-80's.

peace,

Bernie Urban

NIBBLES

Beginning September 15, 1979, Apple Computer, Inc., will offer a low-cost extended warranty for its personal and small business computer customers.

For $195, the one-year extended warranty features "Same-day Turnaround" for carry-in repairs at Apple's authorized Level I service centers. The extension covers all Apple systems and products, as well as additions to the base system made during the warranty period.

Will Houde, Apple's director of service operations, said that the new warranty program will emphasize local dealer support to Apple customers.

The extended warranty may be purchased at Apple Level I service centers during the normal 90-day parts and labor warranty period. It may be renewed in annual increments.π

Recognizing the need for expanded educational opportunities, Apple Computer, Inc. announced the formation of the Apple Education Foundation. Initially funded by Apple Computer, the nonprofit foundation will offer support and resources to organizations and individuals who are pioneering learning methods through the use of microcomputers. The foundation will distribute hardware equipment for both developmental and demonstration projects involved in producing instructional computing materials. In addition, a few funding grants will be available for educational enrichment projects.

Final grant proposals and authorizations for funding disbursements will be reviewed by a board of directors, backed by an advisory council composed of leaders in the field of computer-based education. The advisors will provide guidance, and will review grant applications and submit them for final approval by the board of directors.

The foundation's primary goal is to place hardware into the hands of people who will further those educational methods which take best advantage of the personal microcomputer's capabilities.

The foundation will also sponsor the Education Program Information Center (EPIC). EPIC will support microcomputer users in developing new instructional programs and in obtaining available information on educational materials. The center will publish information packages containing design and development guides, editorial and marketing guidelines, software techniques and authorizing tools. Authors are encouraged to submit their work to the center for review and feedback on the most effective uses and placements of their materials.

Further assisting microcomputer users, EPIC's Users Guide will give overviews of state-of-the-art computing, plus critical reviews of educational programs available for popular small computers.

Both the Apple Education Foundation and EPIC may be contacted at: Apple Education Foundation, 20605 Lazaneo Drive, Cupertino, CA 95014. π

continued

The effort to initiate a group purchase of the IDS 440 printer is well underway and a demonstration of this unit interfaced with an Apple is planned for the Sep 29 meeting. The 440 retails for $995 and the optional graphics 2K buffer is $199. We presently have 5 printers on order (all with high-resolution graphics option) and these will be shipped on a first ordered basis. An additional order for 5 or more printers prior to October 15 will qualify all purchasers for a 12% discount, otherwise 10%. An additional 2% discount is available for those who wish to pay at the time they place their order. Members can place their orders at the September meeting or can call me at 292-3100 after October 2.

Hersch Pilloff π

MicroNET tm is a computer time sharing and software distribution service for home and small business applications. The service costs only $5 per connect hour plus a one-time application fee of $9, part of which is refunded to you in the form of one free hour of connect time. All billing is through Visa or Master Charge accounts. On-line file storage (up to 64K bytes) is also included in the basic connect time rate. Files must be accessed at least once every seven days. Contact Personal Computing Division, CompuServe Inc., 5000 Arlington Centre Blvd., Columbus, OH 43220 (614) 457-8600. π

If you have been wondering where to get that temperature transducer so you can control your house heating or for experimentation, there is available an inexpensive two-terminal IC temperature transducer from Tri-Tek, Inc. This little device, which comes in a TO-52 metal can (about 1/8 inch diameter) produces an output current proportional to absolute temperature. It can be used with +4 to +30V supplies and is excellent for remote applications due to its high impedance. Because this produces variable current output, voltage variations are nulled. Although Tri-Tek requires a minimum of $20 on charges, this transducer is only a pleasant $3.49 each and requires only a trimmer resistor for operation.

AD590J $3.49. Include $.80 for Specs and Application sheets.

TRI-TEK, Incl, 7808 N. 27th Avenue, Phoenix, AZ 85021 (602) 995-9352. π

Remember Jade Computer Products? Well, they now have released their 1979 Software Catalog which is jammed with the most popular and interesting Apple and other software for microcomputers. The catalog is organized by the type of software, e.g., High Level Languages, Games and Simulations, Educational Software, etc. They are also soliciting authors' programs for distribution. Contact Jade Computer Products, 4901 W. Rosecrans, Hawthorne, CA 90250 (213) 679-3313. π

Located in Baltimore, the Muse Company has begun operation of its personal computer phone service. It will provide a bulletin board, software demonstrations and on-line ordering. Dial (301) 661-8962/3. They are using D.C.Hayes MICROMODEMS which are compatible with the Bell System model 103 low-speed modem and are normally operating at 300 baud, full duplex. They are compiling a directory of modem equipped microcomputer owners. If you would like to be listed, contact MUSE MICRO-PHONE, 7112 Darlington Dr., Baltimore, MD 21234 (301) 661-8547. π

EVENT QUEUE

π There will be a Personal Computing Convention in Philadelphia on October 5,6,7 at the Civic Center. See this months Byte magazine for details. $10 at the door will get you in for all three days. There are supposed to be exhibits as well as tutorial and lecture sessions. π

π Washington Apple Pi will meet Saturday, September 29 at George Washington University corner of 23rd and H Streets NW in Tompkins Hall School of Engineering Room 206 at 9:30 a.m.

π NOVAPPLE will meet September 27 at Computerland Tysons Corner and October 10 at Computers Plus Franconia. Both meetings are at 7:30 p.m.

MODEMania

MORE BULLETIN BOARDS, ETC. REPRINTED COURTESY OF AMRAD NEWSLETTER SEPTEMBER 1979.

```
STATE CITY  SPONSOR  TYPE  PHONE NUMBER

CA FRESNO..................209-638-6392
CA LAWNDALE COMPUTER........... ... ....
...COMPONENTS INC ABBS......213-370-3160
CA LOS ANGELES................. ... ....
...SAN FERNANDO CBBS........213-843-5390
CA WESTMINSTER COMPUTER........ ... ....
...COMPONENTS INC ABBS......714-898-1984
CA SAN DIEGO COMPUTER.......... ... ....
...MERCHANTS ABBS...........714-582-9557
CA HAWTHORNE ABBS...........213-675-8803
CA HUNTINGTON BEACH KORS-...... ... ....
...MEYER ELECTRONICS ABBS...714-964-4346
CA MARINA DEL REY ABBS......213-821-7369
CA SANTEE PEOPLES' MSG......... ... ....
...SYSTEM ABBS..............714-449-5689
CA SIGNAL HILL PERIPHER-....... ... ....
...ALS UNLTD INC ABBS.......213-424-3506
CA SAN FRANCISCO ABBS.......415-668-4246
CA SAN DIEGO BILL'S ABBS....714-449-5689
CA PASEDENA CBBS............213-795-3788
CA SAN DIEGO COMPUTER SOC...714-697-2176
CA SANTA CLARA CBBS.........408-246-2805
CA SAN DIEGO STAN SKOG-........ ... ....
...LUND INFOBIT CBBS........714-565-0961
CA CANOGA PARK SAN FER-........ ... ....
...NANDO VALLEY ABBS........213-340-0135
DC WASHINGTON JWACS............ ... ....
...110 BAUD.................202-635-5710
...300 BAUD.................202-635-5730
DC (ALSO SEE VA)............... ... ....
FL DESTIN FT WALTON............ ... ....
...BEACH ABBS...............904-243-1257
FL MIAMI ABBS...............305-821 7401
GA ATLANTA NORTHSTAR........404-934-1520
GA ATLANTA COMP SOC CBBS....404-394-4220
GA ATLANTA..................404-458-4886
GA ATLANTA..................404-325-0526
IL CHICAGO PERS COMP........... ... ....
...OF CHICAGO ABBS..........312-337-6631
IL CHICAGO CBBS.............312-528-7141
IL CHICAGO FORUM 80.........312-925-0259
IL JOLIET WAYNE JUPITER........ ... ....
...110 BAUD ONLY............815-727-7069
KS WICHITA FORUM 80.........316-746-2078
MA BOSTON...................617-963-8310
MA MAYNARD NECS CBBS........617-963-8310
MD PARKVILLE MUSE CO........301-661-8962
MD PARKVILLE MUSE CO........301-661-8963
MO KANSAS CITY..............816-737-1031
MO KANSAS CITY FORUM 80.....816-861-7040
NJ BOUND BROOK S JERSEY........ ... ....
...ELECTRONIC MAIL SYS......201-457-0893
NY LONG ISLAND ABBS.........212-448-6576
OK AKRON DIGITAL GROUP......216-745-7855
OR BEAVERTON CBBS NW........503-646-5510
SC COLUMBIA UNIV OF COL.....803-771-0922
TX DALLAS CBBS..............214-641-8759
TX DALLAS FORUM 80..........214-288-4859
TX DALLAS UNIV OF TX........214-634-0842
TX DALLAS UNIV OF TX........214-634-0878
TX FT WORTH FORUM 80........817-923-0009
TX HOUSTON ABBS.............713-977-7019
TX SAN ANTONIO ABBS.........512-657-0779
VA ALEXANDRIA POTOMAC.......... ... ....
...MICRO MAGIC INC..........703-750-0930
VA FALLS CHURCH VIRGINIA....... ... ....
...BUSINESS SYSTEMS ABBS....703-533-8591
VA MCLEAN PAUL RINALDO......703-893-W4RI
...W4RI ABBS................703-893-9474
VA VIENNA AMRAD WD4IWG......703-281-2125
```

Internal Structure of Integer BASIC

by Sandy Greenfarb

The purpose of this article is the consolidation of information on the internal structure of integer BASIC. Any similarity between this and the reference material is intentional, and reflects the outstanding job of the originating authors. The material presented is advanced and is not intended for beginners. Despite this fact, pains have been taken to carefully define any terms that readers might be experiencing for the first time. Also, some introductory material has been added to make this article a complete entity, requiring no additional reference.

Within the Integer BASIC internal structure is contained a myriad of strange animals, special-purpose bytes, characters, tokens, relative addresses, and absolute addresses:

BYTE - the smallest addressable unit on the APPLE II. Each byte can represent 256 different values. On a 32K machine, there are 32,768 addressable locations. A byte is composed of eight bits, each capable of a "zero" or "one" state. To paraphrase, "a byte is a byte is a byte". These 256 values may be considered in several ways, all a reflection of the same value. Sometimes the values may be considered as representing Ø thru 255 (absolute value), other times -128 thru +127 (signed value). Later on will be expressed terms such as positive token and negative ASCII. These will be referring to the state of the left-most or high-order bit. In algebraic considerations, when this bit is a zero, the byte is considered positive, and when the bit is one, the byte is negative.

SPECIAL PURPOSE BYTE - is just what it says. For whatever systematic reason, a byte in a certain position or place has a special defined set of special meanings for its possible values.

CHARACTERS - Integer BASIC uses the standard ASCII character set. Suffice it to say that this is a standard way of representing the various alphabetic, numeric, special (punctuation), and control characters that are available for use. These are listed in the ASCII Character table. At this stage, some readers have opened their Applesoft manuals and compared the ASCII table in this article with that in the manual and are noting the differences in the values. Applesoft uses positive ASCII (high bit=Ø) and Integer BASIC uses negative ASCII (high bit=1). Add decimal 128 or hexadecimal $80 to the Applesoft values to realize they are the same. (Note that as a shorthand, the dollar sign "$" is used as a prefix to indicate a hexadecimal number).

TOKEN - As a way of reducing the size of programs, Integer BASIC internally reduces all command language to single characters. For example, GOSUB becomes $5C. These characters are referred to as tokens, in effect a "tokenized" representation for particular meanings. The Token table contains all tokens used with Integer BASIC.

ADDRESSING - "Where do the Browns live? Eight houses up the street at 6502 Apple Lane." Relative addressing is relative to the current location...eight houses up the street (from here). Absolute is a self-contained entity...6502 Apple Lane. Integer BASIC uses both. Note that since one byte can only describe 256 unique values, it takes two bytes to fully describe an APPLE II address.

WORD - A pair of bytes with a specific meaning; most often used are address words and pointers, a form of address. Because of hardware reasons, these words are formed with low byte first and then high byte. The actual value of these "words" is calculated by multiplying the second byte by 256 and adding the first byte to that result. This is not really so strange as it sounds. Think of your APPLE II as a city. This city has 256 blocks named Zero street thru 255th street. Each street has 256 houses numbered zero thru 255. Now, is there anything wrong with saying 32 28th Street? That's the same format as in the APPLE II - low then high portion.

POINTER - is a word that contains the address of another location. A pointer "points to" or designates the location which address it contains.

****************************END OF INTRODUCTORY INFORMATION****************************

Figure 1 illustrates the four significant pointers for an Integer BASIC program in memory. HIMEM is normally the top of memory available for programming. The pointer word for HIMEM ($4C and $4D) contains the location of the first byte immediately following the last byte of the last line of the program. If DOS is resident, it contains the address of the first byte reserved by DOS. HIMEM may be changed by the user with the HIMEM: command. The general purpose of the HIMEM pointer is to indicate the end of the area occupied by a program. PP or Program Pointer ($CA and $CB) denotes the first byte of a program. After a Control-B or NEW or DOS INT command, PP will be equal to HIMEM. This is not to say that there is no program in memory, but to say that Integer BASIC has commanded its pointers to ignore what is there as not significant. Such programs are sometimes recoverable, but are not discussed within the scope of this article.

As program lines are entered (by keying in or EXECing a text file or by LOADing), PP decreases to allow for the growth of the program. The lower limit is PV (next paragraph) at which time any attempt to add more program will cause MEM FULL ERROR.

LOMEM pointer ($4A and $4B) is the address assigned for the start of variables. This address is normally $800 (2048) unless changed by a LOMEM: command. PV, the Variable Pointer for the end of variables ($CC and $CD) is initially equal to LOMEM, if no variables are actively assigned as in the case of a NEW, LOMEM, CLR, or RUN command. Similar to HIMEM, PV contains the address of the location immediately following the last location allocated to variables. While PP changes while a program is being entered, PV does not change until a program is running. Each time Integer BASIC encounters a symbolic variable (during the running of a program), it searches the symbol table. If the referenced symbol is not found, it is created and added to the variables and PV increases to account for its required space. While the variables are being built during the running of a program, an attempt to increase PV greater than PP will also cause MEM FULL ERROR. Note that PV and the variables remain intact on a soft entry to a program (CON, GOTO line number, or machine language trickery).

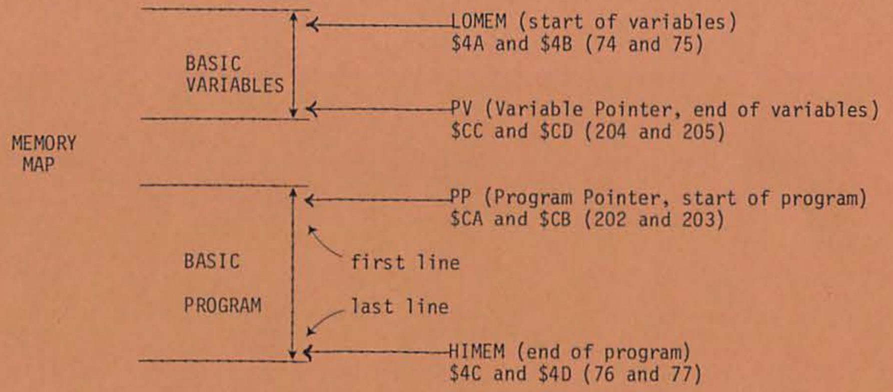
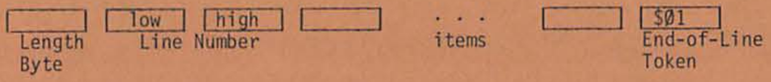

LOMEM (start of variables)
$4A and $4B (74 and 75)
BASIC
VARIABLES
PV (Variable Pointer, end of variables)
$CC and $CD (204 and 205)
MEMORY
MAP
PP (Program Pointer, start of program)
$CA and $CB (202 and 203)
BASIC
first line
PROGRAM
last line
HIMEM (end of program)
$4C and $4D (76 and 77)

figure 1

Lines of a BASIC program are not stored as they were originally entered (in ASCII) on the APPLE II due to a pretranslation stage. Internally each line begins with a length byte which may serve as a link to the next line (relative addressing). The length byte is immediately followed by a byte pair line number stored in binary, low order byte first. The pretranslator only accepts line numbers from Ø to 32767, however there are ways to enter and sometimes legitimate uses for line numbers up to 65535. The line number is followed by items of various types, the final of which is an end-of-line token (#Ø1). Refer to figure 2.

figure 2 - LINE REPRESENTATION



Single bytes of value less than $80 (positive ASCII) are tokens generated by the translator. Each token stands for a fixed unit of text as required by the syntax of BASIC. Some stand for keywords such as PRINT or THEN while others stand for punctuation or operators such as "," or "+".

Integer constants are stored as three consecutive bytes. The first contains an arbitrary ASCII digit ($BØ-$B9) signifying that the next two contain a binary constant stored low order byte first. (This provides a means of distinguishing from a symbolic variable name which by definition begins with an alphabetic letter.) The line number is not itself preceded by $BØ-$B9 as its position in the line has already defined its meaning. All constants are in this form including line number references such as in the statement GOTO 500. Although one or both bytes of a constant may be positive (less than $80) they are not tokens. A constant is always followed by a token.

Variable names are stored as consecutive ASCII characters with the high order bit set (negative ASCII). The first character is between $C1 and $DA (ASCII A-Z), distinguishing names from constants. All names are terminated (followed) by a positive token. It should be noted that the $ in string names is represented by the token $40 rather than the ASCII $A4.

String constants are opened with $28, the token for left quote, and closed with $29, the token for right quote. Between is normally negative ASCII. REM statements begin with the REM token $5D followed by ASCII text followed by the end-of-line token $Ø1.

figure 3 - ITEMS

CONSTANT (1500)	$B1 (BO-B9)	$DC low	$Ø5 high	some positive token	
NAME (ABC)	$C1	$C2	$C3	some positive token	
STRING NAME (BØ9$)	$C2	$BØ	$B9	$4Ø	some positive token

-continued-

figure 3 continued

STRING CONSTANT (A-1)	$28 left quote	$C1	$AD	$B1	$29 right quote
REM	$5D REM token	ASCII	ASCII	ASCII	$Ø1 end-of-line token

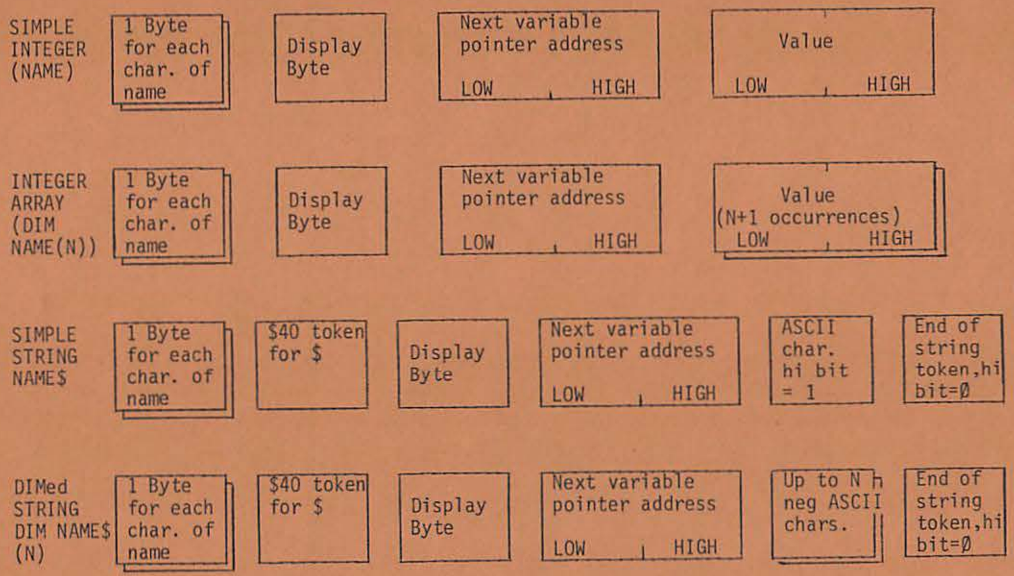
Whether in immediate or RUN mode, when BASIC recognizes a variable name, it searches the variable area to determine if the variable has already been defined. The search starts at LOMEM and ends when the variable is identified (in the area) or PV is reached. If PV is reached without a match, the variable is added and PV is increased appropriately. This search logic should make it apparent why "frequently used" variables should be the first encountered (initialized) in a program. There are four types of variables in Integer BASIC: simple and DIMensioned integer variables and simple and DIMensioned strings. Each has its own unique format, however, all four formats are very similar.

All four formats begin with the variable name represented in negative ASCII. As was true in the program area, in the variable area as well, the $ of a string name is represented by a $40 token. Next comes the display byte. This byte has two functions. First, by nature of its only possible values, zero or one, it delimits or denotes an end to the symbolic name. Second, it indicates whether or not the variable should be displayed ($00=no display, $01=display). (See pages 23 and 25 of the "Red Book", DSP and NO DSP commands.) The next two bytes in a variable definition are the pointer to the start (byte) of the next variable name. This is an absolute address. At this point, the four types of variables differ. A simple integer value has only two more bytes representing the value of the integer (LOW/HIGH). A DIMensioned integer has two bytes plus two more bytes for the size of the DIM statement. DIM A(5) would have two bytes reserved for the value of A(0) and two more bytes each for the respective values of A(1) thru A(5). Note that there are no special tokens to indicate the end of the DIM. As the next variable will begin in the first byte after the current one, and as this address of the next variable is defined (in the preceding two bytes to the variable values), Integer BASIC has sufficient facility for determining the size of integer values.

On the other hand, string variables might not necessarily occupy their full alloted space, and their format is a little different. Following the next variable pointer, a simple (non-DIMed) string uses two more bytes. The first is for the negative ASCII character. (remember that a non-DIMed string can only be one or no characters in length.) The second byte is a positive value (less than $80) denoting the end of the string. As a special case, if the string is null, both bytes will be equal to $ØØ. The variable value portion of a DIMed string is "N" bytes (where N is the length described in the DIM statement) plus one for the end-of-string token. As above, if the string is null, the first two value bytes = $ØØ. Strings may be shorter than their defined length. BASIC reminds itself of the actual string length by making the first non-used byte a positive token. For example with the following short program: 10 DIM A$(5) : A$ = "ABC" : END would produce in the variable area in hex "C1 C2 C3 1E FF FF". Note the use of $1E to indicate the premature end of the string. The variable area formats are described in figure 4. For what it's worth, the following is the length of the area required for each type of variable:

INTEGER NAME = number	Five bytes + number of characters in name.
DIM INTEGER DIM NAME(N)	Five bytes + number of characters in name + 2*N.
SIMPLE STRING NAME$="X"	Five bytes + number of characters in name$.
DIM STRING DIM NAME$(N)	Four bytes + number of characters in name$ + N.

figure 4 - VARIABLE FORMATS


SIMPLE INTEGER (NAME)
1 Byte for each char. of name
Display Byte
Next variable pointer address
LOW
HIGH
Value
LOW
HIGH
INTEGER ARRAY (DIM NAME(N))
1 Byte for each char. of name
Display Byte
Next variable pointer address
LOW
HIGH
Value (N+1 occurrences)
LOW
HIGH
SIMPLE STRING NAME$
1 Byte for each char. of name
$40 token for $
Display Byte
Next variable pointer address
LOW
HIGH
ASCII char. hi bit = 1
End of string token,hi bit=Ø
DIMed STRING DIM NAME$ (N)
1 Byte for each char. of name
$40 token for $
Display Byte
Next variable pointer address
LOW
HIGH
Up to N neg ASCII chars.
End of string token,hi bit=Ø

Note: For variable strings, remember the null string is represented by two bytes of $00 immediately following the Next variable pointer address. Also remember that the "in use" length of a DIMed string may be less than its defined length and that the length can be identified by being followed by a positive token.

FINAL NOTES: The remaining material is in no logical order, but expresses some additional materials and experiences that are related to the article without having a logical place to fit without detracting from the presented materials. With a knowledge of the internal structure, one can now figure how to add illegal statements such as HIMEM:, DEL, LIST line, etc. APPLE wisely made these statements illegal. Any "Trickery" that is added to a program should be carefully tested to ensure it accomplishes its desired purpose for all possible situations. In general, it should not be used, but this is not meant to deny those few situations where it is exactly what is needed. All that is recommended is that when deviating from the "standard" Integer BASIC, be ULTRA-careful.

LOMEM may be changed within a program by entering an illegal LOMEM command in a program or by POKEing the pointers. Remember that LOMEM causes PV to equal LOMEM, effectively deleting all variables.

STRINGS: By knowing where strings are stored in memory, it is possible to simulate the APPLESOFT CHR$ function or even simulate string arrays. To quote most every college professor, "The exercise is left to the reader."

HIMEM: When possible, set or reset HIMEM between programs, that is if it must be changed. Lowering the value of HIMEM will automatically move the program downward in memory to correspond to the new values of HIMEM and PP. Raising the value of HIMEM will not! The program works normally until it reaches a branch instruction (GOTO or GOSUB), then it tries to find a line number which isn't where it should be. At this point the program generally "hangs up".

Why do illegal HIMEM statements only work part of the time? This has a complicated but beneficial answer. As stated above, when HIMEM is lowered, the program is lowered in memory to correspond. That is the first fact. The second half of the answer is the knowledge of how BASIC executes its lines. Once a program is running, Integer BASIC has no need to refer to the PP (Program Pointer) unless it reaches a branch instruction. Instructions continue to be executed in sequence WITHOUT REGARD TO POINTERS until a branch is reached. When a HIMEM lowers a program in memory, it may cause the program to overlay a portion of itself. Should it cause the area of memory which is currently executing to change, it may (and generally will) cause the area previously occupied by the next instructions to become "garbage" which will confuse BASIC and normally "hang up" the machine. For these reasons, illegal HIMEM statements should be in the latter portion of the program which will be left intact (even though no longer pointed to) if the lowering of HIMEM is significant enough. This same "sequential" execution is what also allows unhitching machine language prefix programs and creating APPENDing routines.

SAMPLE APPEND ROUTINE:
```
10000 POKE Ø,PEEK(76):POKE 1,PEEK(77);REM SAVE ORIGINAL HIMEM
10010 POKE 76,PEEK(202):POKE77,PEEK(203):REM LOWER HIMEM TO
      CURRENT PP, LEAVING OLD PROGRAM INTACT
10020 PRINT "D^CLOAD NEW-PROGRAM": REM FIRST CHARACTER OF PRINT
      STATEMENT IS CONTROL-D
10030 POKE 76,PEEK(Ø):POKE 77,PEEK(1): REM RESTORE HIMEM POINTER.
      NEW-PROGRAM IS NOW APPENDED TO BEGINNING OF OLD PROGRAM.
```

The warning with this type of routine is to ensure that the line numbers of the appended program are less than the original program line numbers. When BASIC executes a branch, it starts at PP (Program Pointer) and searches upward until it finds the desired line number. If there is more than one with the same number, the second could never be the object of a branch. If BASIC finds a line number greater than the object before it finds the object, the program will halt with a BAD BRANCH ERR. ∏

TABLE OF ASCII CHARACTER VALUES

ASCII	APPLE	NUMBER		ASCII	APPLE	NUMBER		ASCII	APPLE	NUMBER	
CHAR	KEYBD	DEC	HEX	CHAR	KEYBD	DEC	HEX	CHAR	KEYBD	DEC	HEX
NUL	Sp^{C}	128	$80	+	sa.	171	$AB	V	sa.	214	$D6
SOH	A^{C}	129	$81	,		172	$AC	W		215	$D7
STX	B^{C}	130	$82	-		173	$AD	X		216	$D8
ETX	C^{C}	131	$83	.		174	$AE	Y		217	$D9
EOT	D^{C}	132	$84	/		175	$AF	Z	↓	218	$DA
ENQ	E^{C}	133	$85	Ø		176	$B0	[	--	219	$DB
ACK	F^{C}	134	$86	1		177	$B1	\	--	220	$DC
BEL	G^{C}	135	$87	2		178	$B2	]	${}^{S}M$	221	$DD
BS	←	136	$88	3		179	$B3	^	--	222	$DE
HT	I^{C}	137	$89	4		180	$B4		--	223	$DF
LF	J^{C}	138	$8A	5		181	$B5	⊤	--	224	$E0
VT	K^{C}	139	$8B	6		182	$B6	a	*sa.	225	$E1
FF	L^{C}	140	$8C	7		183	$B7	b		226	$E2
CR	M^{C}	141	$8D	8		184	$B8	c		227	$E3
SO	N^{C}	142	$8E	9		185	$B9	d		228	$E4
SI	O^{C}	143	$8F	:		186	$BA	e		229	$E5
DLE	P^{C}	144	$90	;		187	$BB	f		230	$E6
DC1	Q^{C}	145	$91	<		188	$BC	g		231	$E7
DC2	R^{C}	146	$92	=		189	$BD	h		232	$E8
DC3	S^{C}	147	$93	>		190	$BE	i		233	$E9
DC4	T^{C}	148	$94	?		191	$BF	j		234	$EA
NAK	U^{C}	149	$95	@		192	$C0	k		235	$EB
SYN	V^{C}	150	$96	A		193	$C1	l		236	$EC
ETB	W^{C}	151	$97	B		194	$C2	m		237	$ED
CAN	X^{C}	152	$98	C		195	$C3	n		238	$EE
EM	Y^{C}	153	$99	D		196	$C4	o		239	$EF
SUB	Z^{C}	154	$9A	E		197	$C5	p		240	$F0
ESC	sa.	155	$9B	F		198	$C6	q		241	$F1
FS	--	156	$9C	G		199	$C7	r		242	$F2
GS	${}^{S}M^{C}$	157	$9D	H		200	$C8	s		243	$F3
RS	${}^{S}N^{C}$	158	$9E	I		201	$C9	t		244	$F4
US	--	159	$9F	J		202	$CA	u		245	$F5
SP	sa.	160	$A0	K		203	$CB	v		246	$F6
!		161	$A1	L		204	$CC	w		247	$F7
"		162	$A2	M		205	$CD	x		248	$F8
#		163	$A3	N		206	$CE	y		249	$F9
$		164	$A4	O		207	$CF	z	↓	250	$FA
%		165	$A5	P		208	$D0	{	--	251	$FB
&		166	$A6	Q		209	$D1	:	--	252	$FC
'		167	$A7	R		210	$D2	}	--	253	$FD
(		168	$A8	S		211	$D3	"	--	254	$FE
)		169	$A9	T		212	$D4	DEL/ RUBOUT	--	255	$FF
*	↓	170	$AA	U	↓	213	$D5				

LF=Line Feed; CR=Carriage Return; SP=Space; ESC=Escape; sa.=keyboard same as ASCII.
s-prefix=shift; c-suffix=control. (*Upper case only)

INTEGER BASIC TOKEN TABLE (with comments by original author-Bruce Tognazzini)

NUMBER		TOKEN	COMMENTS (Minor additional comments by S.G.)
DEC	HEX		
0	$0	HIMEM:	Token irrelevent - used internally as begin-of-line
1	$1		End-of-line token - each line ends with $01
2	$2		Used internally in delete line processing
3	$3	:	Colon for statement separation
4	$4	LOAD	Tape command
5	$5	SAVE	Tape command
6	$6	CON	
7	$7	RUN	RUN n, where n is a line number
8	$8	RUN	RUN from first line of program
9	$9	DEL	
10	$A	,	Comma used with DEL (Del 0,10)
11	$B	NEW	Program self-destruct unless you know what you're doing
12	$C	CLR	Clears variables, resets PV to LOMEM value
13	$D	AUTO	
14	$E	,	Comma used with AUTO (AUTO 10,20)
15	$F	MAN	
16	$10	HIMEM:	The real thing, note colon is already in command
17	$11	LOMEM:	
			THE FOLLOWING ARE NUMERIC OPERATORS:
18	$12	+	
19	$13	-	The associated parenthese are 56(left) and 114 (right)
20	$14	*	example: A = 14 * (27+15)
21	$15	÷	
			THE FOLLOWING ARE NUMERIC VARIABLE LOGIC OPERATORS:
22	$16	=	example: IF X = 13 THEN END
23	$17	#	
24	$18	>=	
25	$19	>	
26	$1A	<=	
27	$1B	<>	
28	$1C	<	
29	$1D	AND	
30	$1E	OR	
31	$1F	MOD	
32	$20	^	
33	$21	+	unused
34	$22	(	used in string DIMs: DIM A$(n)
35	$23	,	comma used in A$(3,3)
36	$24	THEN	Followed by a line number: IF X=3 THEN 10
37	$25	THEN	Followed by a statement: IF X=3 THEN A$="CAT"
38	$26	,	used with string inputs: INPUT "WHO",W$
39	$27	,	used with numeric inputs: INPUT "QUANTITY",Q
40	$28	"	Beginning or left quote
41	$29	"	Ending or right quote
42	$2A	(	substring left parenthesis: PRINT A$(12,14) used with 114 as right parenthesis (see also 66)
43	$2B	!	unused
44	$2C	!	unused
45	$2D	(	variable array left parenthesis: X(12) used with 114 as right paren.

NUMBER		TOKEN	COMMENTS
DEC	HEX		
46	$2E	PEEK	uses 63 and 114 for parentheses
47	$2F	RND	" "
48	$30	SGN	" "
49	$31	ABS	" "
50	$32	PDL	" "
51	$33	RNDX	unused
52	$34	(	used in variable DIMS: DIM A(10)
53	$35	+	unary signum: A=+5
54	$36	-	unary signum: B=-5
55	$37	NOT	numeric: IF NOT A THEN B=3
56	$38	(	used with 114 in logic statements and numeric operations: IF C AND (A=14 OR B=12) THEN X=(27+3)/13
57	$39	=	string logical operator: IF A$="CAT" THEN ...
58	$3A	#	string logical operator
59	$3B	LEN(	uses 114 as right parenthesis
60	$3C	ASC(	" "
61	$3D	SCRN(	" "
62	$3E	,	comma used with SCRN: PRINT SCRN(X,Y)
63	$3F	(	used with 114 after PEEK, RND, SGN, ABS, and PDL
64	$40	$	string
65	$41	$	unused
66	$42	(	special case string array right parenthesis, used when string array is the destination of the data. In the example, A$(1)=B$(1), the A$ left parenthesis will be 66 and B$'s will be 42. Used with 114 as right parenthesis.
67	$43	,	Next variable in DIM statement is string: DIM ANYTYPE,STRING NAME
68	$44	,	Next variable in DIM statement is integer: DIM ANYTYPE,INT NAME
69	$45	;	String prints: PRINT ANYTYPE;STRING VAR NAME
70	$46	;	numeric prints: PRINT ANYTYPE;NUMERIC VAR NAME
71	$47	;	end of print statement PRINT A;
72	$48	,	string prints: PRINT ANYTYPE, STRING VAR NAME
73	$49	,	numeric print: PRINT ANYTYPE, NUMERIC VAR NAME
74	$4A	,	end of print statement: PRINT A$,
75	$4B	TEXT	
76	$4C	GR	
77	$4D	CALL	
78	$4E	DIM	string var. Parentheses 34 and 114. If comma is used, it is 67 or 68, depending on type of next variable.
79	$4F	DIM	numeric var. Parentheses 52 and 114. Comma same as with 78.
80	$50	TAB	
81	$51	END	
82	$52	INPUT	String with no prompt: INPUT A$
83	$53	INPUT	String or numeric with prompt: INPUT "WHO",A$ uses comma 38 INPUT "NUM",A uses comma 39
84	$54	INPUT	numeric with no prompt: INPUT A

THE FOLLOWING ARE FOR FOR/NEXT LOOPS:

85	$55	FOR	
86	$56	=	
87	$57	TO	
88	$58	STEP	
89	$59	NEXT	
90	$5A	,	NEXT I,J

NUMBER		TOKEN	COMMENTS
DEC	HEX		
91	$5B	RETURN	
92	$5C	GOSUB	
93	$5D	REM	
94	$5E	LET	
95	$5F	GOTO	
96	$60	IF	
97	$61	PRINT	string variable or literal: PRINT A$:PRINT"HELLO"
98	$62	PRINT	numeric value: PRINT 123: PRINT A: PRINT ASC(A$)
99	$63	PRINT	dummy PRINT: PRINT:PRINT
100	$64	POKE	
101	$65	,	comma used with POKE
102	$66	COLOR=	
103	$67	PLOT	
104	$68	,	comma used with PLOT: PLOT X,Y
105	$69	HLIN	
106	$6A	,	comma used with HLIN
107	$6B	AT	AT used with HLIN
108	$6C	VLIN	
109	$6D	,	comma used with VLIN
110	$6E	AT	AT used with VLIN
111	$6F	VTAB	
112	$70	=	string -- non-conditional: A$="HELLO"
113	$71	=	numeric -- non-conditional: A=14
114	$72	)	the only right parenthesis token - won most popular token award at Atlantic City
115	$73	)	unused
116	$74	LIST	List a range of numbers or specific number: LIST 10:LIST 5,30
117	$75	,	comma used with list
118	$76	LIST	LIST entire program
119	$77	POP	
120	$78	NODSP	string variable
121	$79	NODSP	numeric variable
122	$7A	NOTRACE	
123	$7B	DSP	string variable
124	$7C	DSP	numeric variable
125	$7D	TRACE	
126	$7E	PR#	
127	$7F	IN#	

ADDITIONAL READING

1. INTEGER BASIC SUBROUTINE PACKAGE, by Bruce Tognazzini, APPLE Software Bank, Contributed Programs Volumes 3-5, pages 69-87. Free, see your dealer.

2 INTEGER BASIC INTERNALS, by APPLE Computer Engineering Staff, AppleSauce, June 1979, Vol 1, No. 4, pages 4.20-4.22. $1.50, for subscriptions $10/yr or single issue, write APPLESAUCE, 12804 Magnolia, Chino, CA 91710.

3 CALL-A.P.P.L.E., various issues. Write CALL-A.P.P.L.E., 8710 Salty Drive N.W., Olympia, WA 98502 for details. Though not specifically used in this article, the author gives full credit for the knowledge and experience gained from reading the issues of CALL-A.P.P.L.E. This is a non-profit organization dedicated to the sharing of knowledge of the APPLE II and APPLE II programming.

References: APPLE II BASIC STRUCTURE, by Steve Wozniak, Dr. Dobbs Journal of Computer Calisthenics and Orthodontia, Issue 23.
APPLE II Reference Manual, January 1978.
APPLE Software Bank, Contributed Programs Volumes 3-5.

Calendars by John L. Moon

This article describes a Perpetual Calendar program. With this program you can print calendars for any month of any year (since the start of modern date keeping), or by using one of the subroutines within the program you can identify the day of the week of any arbitrary day.

See the attached listing of the program to follow this discussion. The heart of my Perpetual Calendar program is the subroutine from line 100 to 190. Its inputs are the variables Y, M, D for Year, Month, and Day. It returns the variable N with a value from 0 to 6 for Saturday, Sunday, ... Friday.

The routine from 1010 to 1095 prints out a formatted calendar for a single month. Its inputs are the variables Y1 and M1 for Year and Month. It prints out the month name and the selected year, and then repeatedly calls the day of week routine for as many days as are in a month to fill in the body of the calendar.

The main loop of the program is from 2000 to 3000. Here is where the instructions are printed out and the user is asked for the Year and Month desired. If the Month is entered as a zero, then a loop is used to call the routine at 1010 from Month 1 to 12 to make the calendar for a year.

The algorithm is derived from Zeller's congruence. An explanation can be found in the September 1979 issue of Byte on page 126. π

```
10  DIM M$(40): M$="JANFEBMARAPRMAYJUNJUL
    AUGSEPOCTNOVDEC"
20  DIM MD(14):MD(1)=31: MD(2)=28: MD(3)=31:
    MD(4)=30: MD(5)=31: MD(6)=30: MD(7)=31:
    MD(8)=31
30  MD(9)=30: MD(10)=31: MD(11)=30: MD(12)=31:
    MD(13)=31: MD(14)=28
40  GOTO 1000
100 IF M>2 THEN 130
110 M=M+12
120 Y=Y-1
130 N=D+2*M+((M+1)/2)
140 N=N+Y+(Y/4)-Y/100+Y/400+2
150 IF MD(M)<31 THEN N=N+1
170 IF M=8 OR M=10 OR M=12 THEN N =N+1
180 N=N MOD 7
190 RETURN
1000 GOTO 2000
1010 M=M1: Y=Y1
1012 DAYS=MD(M)
1013 IF(Y MOD 4 =0 OR Y MOD 400 =0) AND M=2
     THEN DAYS=29
1020 PRINT: PRINT "      ";
1021 PRINT "          ";M$((M-1)*3+1, (M-1)*3+3);
     "  ";Y
1022 PRINT
1030 PRINT " SUN MON TUE WED THU FRI SAT"
1035 PRINT
1040 FOR D=1 TO DAYS
1050 M=M1: Y=Y1: GOSUB 100
1060 IF N=0  THEN N=7
1070 IF D>9 THEN 1072: TAB 4*N: PRINT D;:
     GOTO 1080
1072 TAB 4*N-1: PRINT D ;
1080 IF N=7 THEN PRINT
1090 NEXT D
1095 RETURN
2000 CALL -936: VTAB 4
2010 PRINT "PERPETUAL CALENDAR PROGRAM":
     PRINT: PRINT "ENTER YEAR, MONTH.  IF
     MONTH IS ZERO"
2020 PRINT "THEN THE WHOLE YEAR WILL BE
     PRINTED"
2025 PRINT "YEAR AND MONTH = 0 ENDS PRO
     GRAM"
2030 INPUT "YEAR, MONTH", Y3, M1
2035 IF Y3 =0 THEN 3000
2040 IF M1 =0 THEN 2050:Y1=Y3: GOSUB 1010:
     GOTO 2010
2050 Y1=Y3: FOR M1=1 TO 12: GOSUB 1010:
     INPUT "HIT RETURN FOR NEXT MONTH",
     A$: NEXT M1: GOTO 2010
3000 END
```

Applesoft Surprise by Jim Kelly

Last month I decided to take the plunge and get an APPLESOFT firmware card. When I inspected my purchase at the store, the salesman pointed out that I had a "full board." I didn't appreciate the significance of that comment at the time. When I got home and tried out my new addition, I discovered to my delight that my firmware card contained the Auto Start ROM! Evidently this is being included free of charge in the newest version of the firmware card.

The design seems to be similar to that suggested by Darrell Aldrich in the July-August 1979 Call - APPLE, except that the old monitor remains with the Integer BASIC mode. The new version also contains a new manual, The APPLESOFT Tutorial, very similar in style to The BASIC Programming Manual written for Integer BASIC. You should be able to tell if you are getting the new version of the firmware by looking at the back of the package - the new card comes with two manuals, the APPLESOFT Reference Manual and the new tutorial, clearly visible from the rear.

There is virtually no documentation on the Auto Start ROM in this new offering. The new tutorial manual assumes that you have Auto Start (it was probably written for the APPLE II Plus system) and has an appendix on the old monitor. But this manual is written at an elementary level and does not recognize the existence of Integer BASIC. The APPLESOFT Reference Manual makes no mention of Auto Start.

There seems also the be one minor operating problem. Switching from Integer BASIC to APPLESOFT sometimes bombs. What happens is that the APPLESOFT prompt symbol appears but every attempt to hit "Return" puts you in the monitor. The only sure solution I've found is to turn the APPLE off and then on again. Switching from APPLESOFT to Integer BASIC appears to be no problem. I don't have a disk, so I don't know if this problem will affect the selection of APPLESOFT from the DOS.

Well, enough of looking a gift horse in the mouth. It is certainly a pleasant surprise to find this little freebie in the box. After all, the Auto Start ROM alone lists for $65.00. With that savings I suppose I can buy the documentation. π

ATTENTION MEMBERS! ATTENTION MEMBERS! ATTENTION MEMBERS! ATTENTION MEMBERS!

Please check your mailing label now. If it has a "P" in the upper right-hand corner, you are a paid member. "C" means complimentary (to our computer stores locally) and (whoops!) "U" means you have not paid your membership dues.

The deadline to send in your dues is October 1, 1979. If you don't, we will remove your name from the mailing list and you will not receive the newsletter.

If you want to be sure to get every issue, fill out the form on the right and send it in with your dues as quickly as possible.

We want to serve our membership as efficiently as possible which means we have to weed out the "dead wood" people who don't care to pay for the excellent work of our club. In that way, we can better serve our members' needs.

Thank you.

Washington Apple Pi Membership Application

NOTE: Club policy prohibits revealing members' names and addresses. Additionally, the information requested below is for planning purposes only and will not be released to anyone, including other members.

NAME____

ADDRESS____

CITY, STATE, ZIP__________________________________

TELEPHONE NUMBERS: HOME (   )______________ WORK (   )______________

PLEASE LIST HARDWARE YOU OWN:______________________________

____

____

OCCUPATION____

I WOULD LIKE TO WRITE ARTICLES FOR THE NEWSLETTER (Y/N)__________

I WOULD LIKE TO ASSIST ON A COMMITTEE (SPECIFY AREAS OF INTEREST IF YES) Y/N____

____

PLEASE ENCLOSE PAYMENT WITH THIS APPLICATION IN THE AMOUNT OF $6 FOR 6 MONTHS

MONTH JOINED____________PAID (Y/N)__________

MAKE CHECK OR MONEY ORDER PAYABLE TO: WASHINGTON APPLE PI

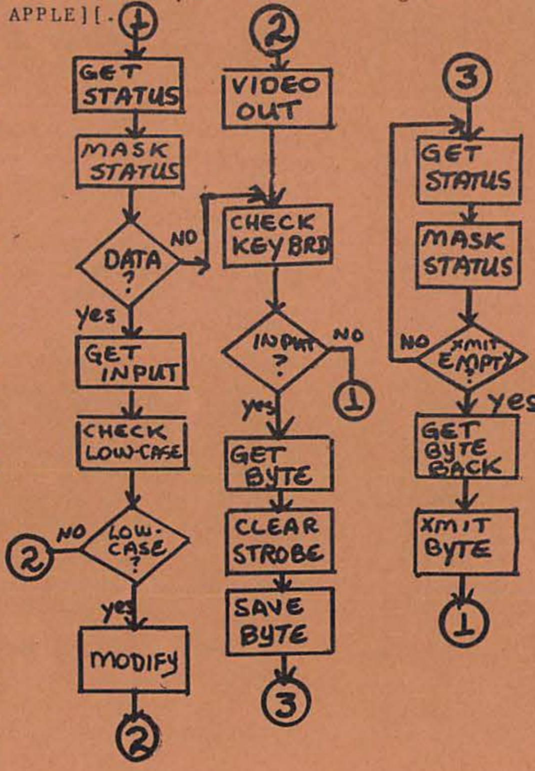
SEND TO: WASHINGTON APPLE PI
PO BOX 34511
WASHINGTON, DC 20034

Hard & Soft Facts on the Serial I/O*

by Susan Eickmeyer

The APPLE][has a lot to offer in its own right, but with a printer attached, or on a time sharing system, the potential of its use increases significantly. Most timesharing systems, and a number of printers (among a lot of other pieces of hardware, use a serial interface, or serial I/O. This is usually just a printed circuit card which sits in one of the slots in the *APPLE][motherboard, and often some associated software. One such serial board, which is available through ELECTRONIC SYSTEMS is fairly inexpensive, uncomplicated and works satisfactorily for many applications. This serial board, like all others, has the 'job' of sending out information one bit at a time. On this board, the bits of information are sent out as voltages, specifically voltages which conform to a standard called RS-232-c. The intent of this article is not to examine that standard, though, so we will leave any further discussion of it to a later article. What I want to delve into here is a discussion of how the APPLE][and the serial board interact, and look at an assembly language program which handles that information. The article assumes you know a little, but not very much about assembly language. The convention of using a " $ " to signify hexadecimal notation will be used in this article. Before going any further, I will present the program I use with the serial I/O board. It is presented below just as it will list out on the APPLE][.

```
0300-  AD B1 C0    LDA $C0B1
0303-  29 80       AND #$80
0305-  F0 0E       BEQ $0315
0307-  AD B0 C0    LDA $C0B0
030A-  09 80       ORA #$80
030C-  C9 E0       CMP #$E0
030E-  30 02       BMI $0312
0310-  29 DF       AND #$DF
0312-  20 F0 FD    JSR $FDF0
0315-  2C 00 C0    BIT $C000
0318-  10 E6       BPL $0300
031A-  AD 00 C0    LDA $C000
031D-  2C 10 C0    BIT $C010
0320-  8D 33 03    STA $0333
0323-  AD B1 C0    LDA $C0B1
0326-  29 01       AND #$01
0328-  F0 F9       BEQ $0323
032A-  AD 33 03    LDA $0333
032D-  8D B2 C0    STA $C0B2
0330-  4C 00 03    JMP $0300
```

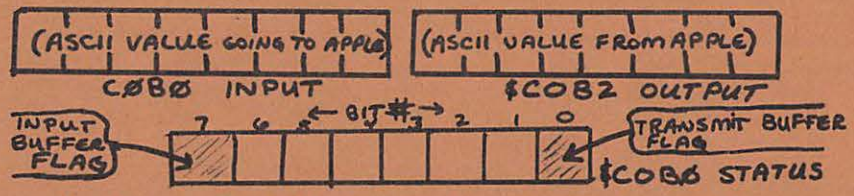


We'll use the program listing and the accompanying flowchart in our discussion of the serial I/O interface, so keep them handy. First let's discuss how the APPLE][communicates with hardware in general. Simply put, the APPLE][sees all hardware devices as memory addresses. These addresses will vary depending on which port or slot the board is in, how the board is connected up, etc. Usually there will be a data sheet with the hardware or board that will supply this information. Our board is in slot #3, and according to the information on documentation that came with it, there are only three addresses with which we need to be concerned. These are:

$C0B0--INPUT
$C0B1--STATUS (bit 7 : input ready / bit 0:transmit empty)
$C0B2--TRANSMIT BUFFER

The input address will always contain the ASCII value of the character in the input buffer, and the transmit buffer address is where we will 'store' any characters we want to send out. STATUS has two functions; 1-to flag whether or not a character has come into the INPUT buffer, and 2-to flag whether or not the Transmit Buffer is ready to accept another character for output. Since the STATUS address is only handling two 'yes' or 'no' questions, it can perform its 'job' by just setting or not setting (setting a bit means making it a one) two of the eight bits in the byte. The program must then specifically test for those bits, and make its decision based on them. We will see how this is done.

The diagrams below will illustrate this setup.



Now we are ready to look at the program in detail. To begin, we GET the STATUS byte. All we have to do to do this is load the accumulator (LDA) with the contents at $C0B1 , which is the address which the documentation told us was the STATUS. Since we first want to find out if the board has information ready for us, we need to check the Input ready bit, bit seven (the high order or leftmost bit) of the STATUS byte. To do this we use an operation known as "Masking", which is done with the 6502 'AND' instruction. When an AND is performed on two bytes of data, corresponding bits in each are analyzed, with a result of 1 if and only if both the corresponding bits are also 1's. For any other combination of bits, the result will be a 0. A couple of examples follow.

```
     10111011          10001001       01110001
AND  00010010      AND 10000000   AND 10000000
     00010010          10000000       00000000
```

fig.1

fig.2

Masking essentially boils down to this: We mask with the number whose binary equivalent has a '1' in and only in the position of the bit in which we are interested. In fig.2 we used the binary 10000000 to mask (AND) . This is the same as $80, as in the program listing. Notice that $80 has a '1' in the 7th bit position, which is the bit in which we are interested. Compare the two examples in figure 2 and notice that when the 7th bit in the number to which we are ANDing $80 is set, we get a non-zero result. If the 7th bit in the other number is not set, we get a zero result. Thus, when we MASK a bit, we get zero if the bit is not set, and non-zero if it is. In our program we use the Mask to determine if the 7th bit is set. This tells us if there is input from the external device. If the AND returns a 0, we know that there is no input, and as the flowchart shows, will go on to check the keyboard. If the result is non-zero, it indicates that there is something in the input buffer. For now let's say that the result was non-zero, so there is data waiting for us. We get the data by simply loading the accumulator with the contents of the INPUT buffer address, or LDA$C0B0. We now have a form of the ASCII value from the external device in our A-register (Accumulator). On ASCII code, the 7th bit may or may not be set, depending on the external device. Since APPLE][requires that 7th bit to be set to avoid getting certain 'garbage' characters, we must be sure that if it was not already set, that it is set by the program. Again, we must manipulate a certain bit, this time leaving all the rest unaffected by our operation. It is possible to do this with the ORA (OR the Accumulator) instruction. The ORA works this way; If either of the corresponding bits between the accumulator and the number with which it is ORA'd are a 1, a 1 is returned as the result. Only if both the bits are 0's will a 0 be the result. Again, examples:

```
 acc.10111011        acc.10001001      acc.01110001
 ORA 00010010        ORA 10000000      ORA 10000000
 acc 10111011        acc 10001001      acc 11110001
    fig3                         fig 4
```

Figure 4 shows an ORA with $80 (binary 10000000). When we ORA we use a number with a 1 in the binary position which we want to set, irregardless of previous condition in the accumulator. We used $80 whose binary has a 1 only in the 7th bit. With the ORA instruction, the reult of the operation is left in the accumulator, with the appropriate bit set. Notice how all the other bits in the accumulator stayed the same. Our accumulator now has the ASCII character code with the high bit set, like the APPLE][wants.

We are still not quite ready to let the APPLE][output its character, however, since we must first check to see if the character we received was lower case or not. The standard APPLE][cannot handle lowercase without software/hardware modifications. For our program we will do it the easiest way and merely convert the lower case letters to upper case. Lower case ASCII characters have a higher value than upper case letters, so we will compare our ASCII value in the accumulator with the smallest lower case value for an ASCII character with the high bit set, $E0. Here we use the CMP instruction. This instruction does a subtraction, but does not store the result in the accumulator, as a regular SBC (subtract) instruction would. The CMP instruction does set flags in our 6502's status register, though, and we can find out if the result was positive, zero or negative. Our use of the CMP will return a negative result only if we have an upper case character. A positive or zero result indicates we have a lower case character, and must modify it so it is handled as an upper case letter by the APPLE][. To modify it we are going to use the AND instruction again, only this time we are going to turn our procedure around and use a zero to change the bit in which we are interested. The difference between upper and lower case ASCII values lies in the 5th bit. It is set in lower case, it is not set in upper case. We need to 'unset' the fifth bit in the lower case letter, and we will end up with an upper case. To do this we AND with a number which has 1's in all bit positions except for the one we want to 'unset'. Since we want to make the 5th bit a zero, we need to use the number $DF, which has a binary value 11011111 (the 5th bit is a zero). Below, we use the letter 'A' as an example of the process.

```
Upper case A ; $C1  binary 11000001         lower 11100001
Lower case a ; $E1  binary 11100001       AND $DF 11011111
                              /                   11000001
                          5th bit         (=upper 11000001)
Modification ; $DF  binary 11011111
```

This modification works for all lowercase letters.

We are finally ready to let the APPLE][have the character and output it to the screen. The APPLE monitor will do the work for us here, all we have to do is jump to subroutine for outputting a character to screen that exists in the monitor at $FDF0. (JSR $FDF0). After the character has been output to the screen, the program will return to this main program for the next step.

So far we have looked at how the APPLE][gets a letter from the serial I/O board, modifies it as necessary, and outputs it to the screen. The other half of the operation involves the APPLE][giving the serial board a character to transmit to the external device.

As the flowchart shows, we must first check the keyboard and determine whether or not a key has been pressed, that is, whether or not there is input from the keyboard. We will use the BIT instruction to check for input. The BIT instruction is similar to the AND instruction, in that it also logically AND's the bits of the A-register with the bits of another specified byte. The primary difference between an AND and a BIT instruction is that the BIT instruction does not store the result of the operation in the A-Register, while an AND instruction does. With a BIT instruction three things occur which can be tested to determine the result of the BIT operation: 1) The zero flag in the 6502 will be set or not set depending on whether or not the comparison of bits succeeds (results in a non-zero number). 2) The 6th bit is transferred into the 6502 overflow flag. 3) The 7th bit of the memory data is transferred to the 'Negative' flag in the 6502. It is the 3rd item, that involving the negative flag that we will need in this step of the program. In the APPLE][, The keyboard is a kind of peripheral device which communicates with the APPLE][, itself, through memory locations. Two addresses are involved in our program, $C000 and $C010. When a key is pressed, the ASCII value of the key can be found at $C000. The important point here is that the APPLE][always sets the high (or 7th) bit in its ASCII representation of a character. When the keyboard strobe is cleared, the high bit is set back to 0. To summarize: if a key has been pressed since the last time the strobe was cleared, the 7th bit will be set. If no key has been pressed since the last time the strobe was cleared, the 7th bit will be 0. Now, when we BIT the keyboard, that 7th bit is transferred into the N-flag (negative flag). If the 7th bit was a one, the N-flag will be set as a result of the operation. If the 7th bit had not been set, the N-flag would not be set as a result of the operation. Hence, if a key was pressed, the 7th bit was set, and the N-flag will be set as a result of the BIT operation. If the N-flag is not set as a result of the BIT operation, then we know that no key was pressed, and we can go back to the start and see if the serial I/O card has any input for the APPLE][. This is exactly what we do in the BPL instruction. This says Branch to the specified address on PLus . APPLE][determines that a number is plus if the N-flag is not set.

Let's assume that the branch failed, that is the N-flag was set (indicating a key had been pressed since the last instruction) so we 'fell through' to the next instruction. Here we simply load the accumulator with the contents of the keyboard address and get the ASCII value for the character. Next we must clear the keyboard strobe, since, if we do not, the 7th bit will stay set, and it will look to the APPLE][like the same key is being pressed repeatedly. We use the BIT instruction again, then see a piece of hardware "magic" occur. All we are doing with the BIT instruction here is addressing a location. The hardware on the keyboard can detect the fact that it has been addressed, and will as a result, clear the keyboard strobe. Actually we could have used any instruction involving that address and had the same result.

Our accumulator still contains the ASCII value from the keyboard, but we need to use our accumulator for something else at the moment, so our next step is to save its contents in a temporary location so we can get them back later. In our program we will store the ASCII value in the accumulator in the location which immediately follows the last byte of the program itself, in $0333. We STore the contents of the Accumulator at $0333 with the instruction STA $0333 .

In the next segment of the program, we will need to test our serial I/O STATUS byte again. Again, we use the LDA $C0B1 instruction to do so. This time, however, we want to know, not whether the board has information for the APPLE][, but whether the board is ready to receive some information from the APPLE][. What we need to find out is whether or not the transmit buffer on the board is empty, and has room to receive a character . We check this, because if input from the keyboard is coming to the board too quickly, the board may not have time to get one character out before the APPLE][tries to shove another one into it. We can see if the transmit buffer on the board is empty by testing bit 0 in the STATUS byte (this information was supplied with the board). Again we will use the AND instruction to MASK for the bit in which we are interested, so here we will MASK with $01, or 00000001 binary. This allows us to look at the bit 0 of the STATUS byte, which will be set only of the transmit buffer is empty. If we get a zero as a result of our AND operation, we will branch back and test the status byte again, so we use the Branch if EQual instruction , BEQ $0323. If a non-zero result is returned, we know that the transmit buffer is ready to receive our ASCII character, which is still sitting in its temporary location in $0333. We get our character back with an LDA $0333 instruction, then pass it on to the board. The documentation supplied with the serial board says that the address for the TRANSMIT BUFFER is $C0B2. To pass the ASCII character to the serial I/O board, we need only to store the contents of the A-register in location $C0B2, and the board will take it from there.

At this point we have travelled through our entire program. We have checked for and handled then output any character sent through the serial I/O board to the APPLE][from an external device; and we have allowed the APPLE][to send out a character to the external device via the board. The only thing left for us to do is to go back to the beginning of the program with a JuMP (JMP) instruction, and start all over for the next character to input or output.

Hopefully the length of this article has not scared too many of you away. Its intent was to explain in detail just how an interfacing program for a piece of hardware worked. Many peripheral devices use many of the same routines as were found in this program. By understanding the principles involved, you may well be able to make modifications and improvements in your own device's software which will make it better adapted for your applications.

- -

*APPLE][is a registered trademark of the APPLE Computer Co. --Cupertino, CA.

**Article based on the APPLE][serial I/O interface, available through Electronic Systems, P.O Box 21638, San Jose, CA. 95151. Program is based on a software listing included in board documentation. Available as board, kit or assembled ($15.00, $42.00 and $62.00 respectively)

π

Classified ads accepted from members 50 words or less at no charge provided the material is obviously non-commercial. Submit your classified at least 30 days in advance attention CLASSIFIED ADS, PO Box 34511, Washington, DC 20034.

YOUR AD HERE !

RATES—	$20	full
	$10	half
	$ 7	quarter
	$ 5	eighth

(line copy only - no half-tones or colors)

Auto–List and Count Filemaker **by Howie Mitchell**

The following is a little program that works well and pleases me. This program will make a file which can be used to add an auto-lister to any BASIC program. It shows some influence from Bruce Tognazinni ("Infinite Number of Monkey"). The last line in the program (line 30026) is from the June 1979 CONTACT 5. Use of the step (lines 30002 and 30004) greatly speeds up the listing, and line 30006 gives something to watch while the program looks for the next line to list. Line 30005 slows down the listing near the end, so it doesn't overshoot. π

```
10     CALL -936: VTAB 2: PRINT "*** AUTO -
       LIST AND COUNT FILEMAKER ***": PRINT
20     PRINT "  THIS PROGRAM WILL MAKE A
       ";: CALL -384: PRINT " FILE ";: CALL
       -380: PRINT " WHICH"
30     PRINT " CAN BE USED TO ";: CALL -384:
       PRINT "ADD";: CALL -380: PRINT " AN
       AUTO-LISTER TO ANY BASIC PROGRAM."
40     PRINT: PRINT "  THERE IS ONE RESTRIC
       TION:": PRINT
50     PRINT "  THIS PROGRAM WILL WIPE OUT
       SOME PARTS OF YOUR PROGRAM, IF THEY
       LIE BETWEEN  LINES 30000 TO 30027.":
       PRINT
60     PRINT "  IN A MOMENT, A FILE WILL BE
       MADE,     CALLED: 'AUTO LIST & COUNT
        FILE'."
70     PRINT: PRINT "   TO USE THE FILE:":
       PRINT: PRINT " 1. LOAD ANY INT. BASIC
        PROGRAM."
80     PRINT " 2. TYPE 'EXEC AUTO-LIST &
       COUNT FILE'."
90     PRINT " 3. TYPE 'RUN 30000' TO RUN AUTO-
       LISTER."
95     IF REP THEN POKE 34, PEEK(37): IF REP
       THEN 30025
100    INPUT " (PRESS 'RETURN' TO CONTINUE.)
        ", HOLD$
110    REP = 1: CALL -936: GOTO 60
29999  END
30000  A=PEEK(224): B=256*PEEK(225): FIRST= A+
       B: DONE = ASC("#"): IF DONE# ASC("%")
       THEN 30011
30001  CALL -936: VTAB 2: INPUT "LIST FROM
       WHAT, TO WHAT ", START, FINISH: PRINT
30002  INPUT " APPROX. HOW FAR APART ARE
       YOUR PROGRAM LINES ", S
30003  CALL - 936: VTAB 3
30004  FOR X = START TO FINISH STEP S: GOSUB
       30009
30005  IF (FINISH-X)>2*S THEN 30006: S=1: FOR
       X=X+1 TO FINISH: GOSUB 30009
30006  Z=PEEK(37): VTAB 1: TAB 20: PRINT "LIST
       THRU: "; X+S-1: VTAB Z+1
30007  IF PEEK (37)>18 THEN GOSUB 30010
30008  NEXT X: PRINT: PRINT "END OF LISTING.
       ": END
30009 PRINT X, X+S-1: RETURN: PRINT "@@": REM
       "DOUBLE AT" USED AS MEMORY REFER
       ENCE FOR CHANGING PRINT TO LIST.
30010  INPUT "@", HOLD$: CALL -936: VTAB 3:
       RETURN
30011  LAST=PEEK(224) + 256*PEEK(225)
30012  FOR X = FIRST TO LAST
30013  IF PEEK(X) = ASC("@") AND PEEK(X+1)=
       ASC("@") THEN GOSUB 30016
30014  IF PEEK(X) = ASC("#") THEN POKE X, ASC
       ("%"): REM INDICATE "DONE IS DONE" IN
       LINE #30000, BY REPLACING ASC("#") WITH
       ASC("%")!
30015  NEXT X: GOTO 30000
30016  POKE X-15, 116: POKE X-13, 117: RETURN:
       REM CHANGE "PRINT X, X+S-1" TO "LIST
        X, X+S-1 IN LINE # 30009!
30017  REM
30018  REM**********************************
30019  REM     PROGRAM BY:
30020  REM     HOWIE MITCHELL
30021  REM     7823 SW. 55th PLACE
30022  REM     GAINESVILLE, FLA., 36201
30023  REM     AUGUST, 1979
30024  REM**********************************
30025  DIM N$(30): N$ = "AUTO-LIST & COUNT
       FILE": D$ = "": REM  D$ = "ctrl. D"
30026  PRINT D$;"OPEN ";N$: POKE 33, 33: PRINT
       D$;"WRITE ";N$: LIST 29999, 30024: PRINT
       D$;"CLOSE": TEXT: END
```

SOFTWARE REVIEW by Mark Crosby

3-D software for the APPLE has arrived - again! SubLogic of Savoy, Illinois has released an assembly language version of their 3D to 2D transformation matrix/converter for the APPLE II.

Aside from requiring minimum memory for efficient and easy-to-understand operation, the programs supplied along with the very good documentation and technical manual are enough to get you up and enthused quickly.

There are 3 main programs and one demo all supplied on one good-quality cassette and each is repeated to assure at least one good copy (I had no trouble with any). The first program is the assembly language portion which fits from $800 to $2FFF (a total of about 10K). The other two programs are an Integer BASIC and an Applesoft version of their "Development" program which is used to develop a three-dimensional scene.

After loading the assembly language, LOMEM is set to 16384 and then either BASIC program is loaded and RUN. The development program POKE's code into memory which are used by the drawing algorhythm to identify points and lines in 3D space. It also contains many utility codes, e.g., a screen erase feature, a continue line code, an "eye" from which the viewer "sees" the 3D scene, etc. All of this code and corresponding memory locations are printed on the screen so that you can save the "scene" or, more specifically the coded instructions on disk or cassette afterwards. Naturally, they can be loaded back in again as necessary. More than one scene can be fit too with "partitions" between them.

Multi-line scenes are drawn very quickly which means you can get into animation too. A selective partial screen erase feature helps in this regard - immensely. Additionally, if you have enough memory and the proper hardware (Applesoft Firmware Card) you can use both Hi-Res pages for "ping-ponging" back and forth to smooth animation (you can erase and re-draw one screen while displaying another, then vice-versa).

In short, this is no run-of-the-mill 3D program rather it is a professionally designed and complete development package that should be valuable to anyone who needs 3D displays. I recommend it for the moderate to advanced programmer as it often requires intimate knowledge of the internal memory map (included in the documentation though) and machine language saves on tapes or disk and the possibility of switching from Monitor to BASIC, etc. Once past those "little" problems, even your kids would have a ball in short order. There is nothing more fascinating than watching a 3D cube rotate before your very eyes! π

A2-3D1 $45 SubLogic, Box V, Savoy, IL 61874 (217) 359-8482