

What's Where in the APPLE?

Prof Luebbert's "What's Where in the Apple" ATLAS of PEEKs, POKEs,
HEX LOCN (DEC LOCN) [NAME] \USE-TYPE\ DESCRIPTION

\$0000~\$FFFF (0~-1) \HB\ RAM APPLE II (\$0000~\$FFFF)
\$0000~\$BFFF (0~-16383) \HB\ RAM APPLE II (NOT
\$0000~\$00FF (0~255) \HB\ RAM APPLE II (NOT
\$0000~\$001F (0~31) \HB\ RAM APPLE II (NOT
\$0000~\$0002 (0~3) \HB\ RAM APPLE II (NOT
\$0000~\$0001 (0~1) \HB\ RAM APPLE II (NOT
\$0000 (0) \HB\ RAM APPLE II (NOT
\$0001~\$0000 (1~0) \HB\ RAM APPLE II (NOT
\$0002~\$0001 (2~1) \HB\ RAM APPLE II (NOT
\$0003~\$0002 (3~2) \HB\ RAM APPLE II (NOT
\$0004~\$0003 (4~3) \HB\ RAM APPLE II (NOT
\$0006~\$0005 (6~5) \HB\ RAM APPLE II (NOT
\$0008~\$0007 (8~7) \HB\ RAM APPLE II (NOT
\$000A~\$0009 (10~9) \HB\ RAM APPLE II (NOT
\$000A~\$000A (10~10) \HB\ RAM APPLE II (NOT
\$000A~\$000A (10~10) \HB\ RAM APPLE II (NOT
\$000C~\$000B (12~11) \HB\ RAM APPLE II (NOT
\$000D (13) \HB\ RAM APPLE II (NOT
\$000D~\$000D (13~13) \HB\ RAM APPLE II (NOT
\$000E~\$000E (14~14) \HB\ RAM APPLE II (NOT
\$000E (14) \HB\ RAM APPLE II (NOT
\$0010~\$0010 (16~16) \HB\ RAM APPLE II (NOT
\$0011 (17) \HB\ RAM APPLE II (NOT
\$0012~\$0013 (18~19) \HB\ RAM APPLE II (NOT
\$0014~\$0015 (20~21) \HB\ RAM APPLE II (NOT
\$0014 (20) [SUBFLG] \HB\ RAM APPLE II (NOT
\$0016~\$0017 (22~23) \HB\ RAM APPLE II (NOT
\$0016 (22) [(COMPTYP)] \HB\ RAM APPLE II (NOT
\$0018~\$0019 (24~25) [(R12)] \P2\ 'SWEET-16' REGISTER R12 (IN 16-BIT
\$001A~\$001B (26~27) [(R13)] \P2\ 'SWEET-16' REGISTER R13 (IN 16-BIT

AN ATLAS TO THE APPLE COMPUTER

By William F. Luebbert

What's Where in the Apple?

An Atlas to the Apple Computer

MICRO INK Books on the Apple Computer

Edited by Ford Cavallari

What's Where in the Apple — An Atlas to the Apple Computer
by William F. Luebbert

August 1981

MICRO on the Apple Series — Works by various authors
published in MICRO magazine, 1977-80

Volume 1

April 1981

Volume 2

Fall 1981

(Other volumes to follow)

What's Where in the Apple?

An Atlas to the Apple Computer

William F. Luebbert
Adjunct Professor of Engineering
Dartmouth College, Hanover, New Hampshire

MICRO INK, Inc.
34 Chelmsford Street
P.O. Box 6502
Chelmsford, Massachusetts 01824

Notice

Apple is a registered trademark of Apple Computer, Inc.
MICRO is a trademark of MICRO INK, Inc.

Every effort has been made to supply complete and accurate information. However, MICRO INK, Inc. assumes no responsibility for its use, nor for infringements of patents or other rights of third parties which would result.

Copyright© 1981 by MICRO INK, Inc.
P.O. Box 6502 (34 Chelmsford Street)
Chelmsford, Massachusetts 01824

All rights reserved. No part of this book may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without prior agreement and written permission of the publisher.

What's Where in the Apple? ISBN: 0-938222-07-4
Printed in the United States of America
Printing 10 9 8 7 6 5 4 3 2 1

Acknowledgements

The information in this book has been accumulated over several years from a wide diversity of sources, including a variety of publications from Apple Computer, Inc., articles from many Apple user group publications and from many magazines, as well as from personal investigations triggered by one or more of these sources.

Unfortunately no record was made in the computer database at the time of original entry of the original source of each datum. Nevertheless the following persons, either through their writings or through personal contact, come to mind as particularly significant sources of information to whom I wish to extend my special gratitude:

Darrell Aldrich
Rick Auricchio
Bob Bishop
C. Bongers
John Crossley
William Dougherty
Andrew Eliason
Val Golding

Andy Hertzfeld
Donald Hyde
Peter Lechner
Lee Meador
C.K. Mesztenyi
Mark Pump
Lee Reynolds
William Reynolds

Lou Rivas
David Roe
Mike Rowe
Loy Spurlock
Dick Sutor
Don Worth
Steve Wozniak

I know that the moment this book goes to the printer, the names of several others who have been inadvertently omitted but who fully deserve to be on this list of those deserving special acknowledgement, will rise up out of my memory to weigh upon my conscience. To such worthy but unrecognized toilers in the orchard I offer, in advance, my sincere apologies.

William F. Luebbert
Hanover, N.H.
July 1981

Special thanks go to the Kiewit Computation Center, Dartmouth College, Hanover, New Hampshire, for assistance in producing the Atlas and Gazetteer output.

Contents

Introduction	9
User's Manual	11
Overview of Apple Memory Organization	
Examples	
Using a Monitor Routine from Applesoft	
Creating a Machine Language Program from Applesoft	
Checking Software Locations from within a Program	
Useful Tables	
Zero Page Usage	
Apple ASCII Representations	
Apple Text Screen Organization	
Atlas Numerical Listing of Apple Locations	
Gazetteer Alphabetical Listing of Apple Locations	

Introduction

You can get more out of an Apple — or any other computer with limited resources — by familiarizing yourself with its hardware and software environment.

When you graduate from simple programs to more ambitious programs involving careful control of man-machine interaction, analog to digital or digital to analog conversion, extensive use of computer graphics, the control of external devices, database management, sorting, or word processing, this knowledge becomes more important. When you get into real time programming, adding your own specialized interfaces, performing activities which require the absolute maximum speed, the information in this atlas becomes critical.

Not every serious programmer needs to become a machine language level programmer. However, good programmers know that when the computer is running their programs there is a lot of machine language code in the machine providing an operating environment for their programs. This operating environment typically includes the system monitor, a BASIC interpreter, and a disk operating system (DOS) and/or extra ROM packages.

When you look at interesting programs described in magazines and user group newsletters, you find that these programs often contain PEEKs, POKEs, and CALLs. These commands are extensions of BASIC (or other higher level languages). They allow you to interface with the computer hardware, operating environment software, and other machine language programs or subprograms.

PEEKs, POKEs, and CALLs all refer to memory locations which are identifiable by what they contain or what they do. PEEK examines the contents of a specified memory location and allows you to use that content in a program. POKE changes the content of a designated memory location to some specified value. It can be used to change parameters of the operating environment or to set up or change pieces of program or data. A CALL transfers program control to a particular memory location back to the CALLing routine in the user's

program. Explanations and examples of how this can be done are given on pages 15 - 17.

Subroutines and other pieces of code from the Apple's firmware (i.e. its MONITOR and BASIC interpreter—Applesoft or Integer BASIC), and from its quasi-firmware (i.e. the DOS 3.2 or 3.3 disk operating system), can be accessed via CALLs to provide useful capabilities without writing any additional code. Some of the more powerful and deeply imbedded machine language routines will require the passing of parameters to and from them. This can usually be done by POKEs and PEEKs.

Usually the code you find built into the Apple system has been carefully written in machine language, optimized by good programmers, and takes less space or less computer time than the same function would require if programmed by the user.

Even in the most awkward cases, where deeply imbedded firmware requires the pre-setting of machine-level hardware registers, it is possible to perform the set-ups without doing any assembly or machine language coding by use of the PEEKs, POKEs, and CALLs to the register SAVE and RESTORE routines built into the system monitor. (There is another similar pair of SAVE/RESTORE routines also built into the Disk Operating System.)

Some users may find it more esthetically pleasing to perform the linkage directly by using machine language instructions such as LDA (LoaD Accumulator), LDX (LoaD X-register) or LDY (LoaD Y-register) to form a tiny machine language linkage program, load it into memory by means of POKEs or S.H. Lam's technique for dynamically entering and exiting the system monitor from a BASIC program. If this is your preference, you will find that it is neither necessary nor desirable to use an assembler for this process. It is easier to hand-code from the information in the Apple Reference Manual, perhaps using the disassembler in your Apple II or II+ (and/or the mini-assembler in the Apple II) to check your work.

Incidentally, there could hardly be an easier and less painful way to back gently into developing

expertise for doing machine language/assembly language programming than by starting out with imbedding just a few machine language instructions into a predominately BASIC program.

A programming manual intended for serious programmers should supply some sort of memory map and information about the most important and frequently used PEEKs, POKEs, and CALLs. A good memory map can show the user where to get

information from the computer, what potentially useful software is available but perhaps hidden away inside the computer, and where to find the "hooks" provided to perform a wide variety of functions by means of CALLs, POKEs and/or PEEKs. Once programmers begin using it as a source of information, they begin to wish for a more complete atlas which will let them find more and more information and guide them in their own explorations inside the computer and its software.

William F. Luebbert

User's Manual

The Apple II contains an address space of over 65,000 locations. Many areas of this space are shared by the user, the monitor, the DOS, and high-level languages. It is possible to write interesting and useful programs without regard to how the Apple memory space is used and managed. In fact, this is exactly how most BASIC and machine language programs are written. Yet a simple knowledge of how the Apple's memory is organized can simplify the task of writing most programs, and can help produce a more compact and efficient code.

Locked within the Apple are many permanently resident routines which can accomplish many common tasks. Most of these routines remain largely undocumented, and thus are difficult for the average Apple owner to use. While information is available here and there about some of these routines, there has been, up to now, no *one* reference source containing documentation on all these routines. The Atlas and Gazetteer which follow contain this information and more.

The *What's Where in the Apple* Atlas is the first complete memory map released for the Apple computer. Further, it is one of the most up-to-date references on the Apple monitor, the Integer and Applesoft BASIC interpreters, the Disk Operating Systems (3.1 - 3.3), and the hardware and I/O memory areas.

The numerically organized memory Atlas considers, in succession, each important memory location in the Apple. Starting with a detailed study of page zero, the Atlas documents each parameter location and vector, each software switch, all video buffer areas, and every data area, in addition to subroutine locations and entry points. The Gazetteer covers the same information, but is arranged in alphabetical order, and contains only those memory locations and areas with commonly used names (e.g. HIRESPG1). Armed with these two packages, any Apple user will be able to gain better performance from the Apple.

The memory maps presented here were not designed with only the machine language programmer in mind. On the contrary, most of the information is of even greater use to the BASIC programmer via PEEKs, POKEs, and CALLs. Thus, numerical memory locations are given in decimal as well as hexadecimal. Utilizing the "built-in" features of the Apple, both Applesoft and Integer BASIC can become much more flexible and powerful languages. Several examples of how the

memory map information can improve BASIC programs are included for reference.

The Atlas and Gazetteer will also aid in understanding previously written BASIC programs which use POKEs and CALLs. Using the numerical Atlas, you can easily check just which routine is being CALLED or which vectors or parameters are being POKEd.

One common complaint with a language interpreter such as Applesoft is that it tends to be very slow when compared to machine language. Directly accessing many of the subroutines embedded within the Applesoft or Integer BASIC interpreter or the monitor may significantly decrease execution time. An example of this would be POKEing information directly to the video screen instead of repeatedly using TABs, VTABs, and PRINTs from BASIC. Another example would be to directly access the Applesoft high-resolution plotting routines via POKEs and CALLs.

The Atlas also points out locations which are of general interest to all Apple users. Examples of such locations include SPDBYT, the address which controls the output speed, PADDL0, the hardware address which contains the paddle 0 position readout, MOTORON, the location which turns on the disk drive motor, or the Applesoft mystery parameter at \$D6 (see the Atlas to find out what this one does!). Various characteristics of the Apple, ranging from video output format to DOS command names, can be changed by simple POKEs to the right addresses.

The *What's Where in the Apple* Atlas and Gazetteer also should be of great use to any machine language programmer who sees the value of tightening up code by using existing Apple firmware subroutines. For instance, 16 and 32 bit divisions can become a problem in 6502 machine code. However, using the already existing routine for division in the Applesoft interpreter can simplify this problem tremendously. The only key necessary is the knowledge of how to use the routine. The Atlas can provide that knowledge.

Countless other uses exist for the Atlas and Gazetteer; you are limited only by your imagination. With enough digging through these pages, you should be able to find information which can help in almost any problem. The examples which follow will illustrate both the different levels of use (BASIC vs. machine language environment, for instance) and some specific application-oriented examples. While going through the examples, you should be able to think up more without too much difficulty. Then it will be time for you to leaf through the Atlas, and find out exactly *What's Where in the Apple*.

Overview of Apple Memory Organization

A few words about the organization of Apple memory may help you find your way around this memory map.

First, even if you don't know a hexadecimal number from a hole in the ground, it is usually more convenient to use the \$ plus 4-character hexadecimal abbreviation for memory addresses when you are looking for information. Its first two characters specify which of the 256 memory pages is being used and its last two characters specify which of the 256 locations within that page. Since the machine language instructions of the machine depend upon this structure, you will find that the utilization of memory and software tends to group related functions on related pages. Thus, instead of a group of oddly sized decimal numbers which seem initially to make no sense at all, you can deal with easy-to-remember 4-character blocks which almost immediately develop a clear and logical structure.

For example, as you go up the memory from page zero towards page 256 you'll find:

Page 0 (\$00xx): Used for frequently accessed parameters

Since the parameters most likely to be used time and time again in running programs are those in the system monitor, the BASIC interpreter and the Disk Operating System, this page is dominated by these uses. (Several important hardware instructions run much faster or only run when memory locations in page zero are used.)

Page 1 (\$01xx): Used for the System Stack

This is a special area used primarily for subroutine returns (both machine language and BASIC), interrupts, and parameters used in re-entrant coding. Only the most careful and experienced programmers should ever fool with this area.

Page 2 (\$02xx): Keyboard and General-Purpose Input Buffer

Characters inputted from the keyboard are stored here. Normally they go no further until an end-of-line carriage-return releases them for further processing.

Page 3 (\$03xx): Linkage Vector Page

Except during DOS booting most of this page is unused except for the extreme top which contains jump commands and linkage vectors to key loca-

tions in firmware (e.g. \$03D0 is the start of the familiar 3D0G linkage which you use to return from the system monitor level to BASIC). During normal operations after disk booting, the otherwise vacant lower sections of this page are a favorite location for short, user-created machine language programs.

Pages 4-7 (\$04xx-\$07xx): Text and Lo-Res Graphics Display Buffer

The 1024 locations on these 4 pages contain 960 locations which correspond one-to-one with the 960 (40 × 24) possible text character positions on the Apple's display screen. The space is organized into 8 macro-lines of 128 bytes, each of which contains 3 text lines (one on the top 1/3 of the screen, one in the same relative position in the middle 1/3, and the last in the same relative position in the bottom 1/3 of the screen). The remaining 8 bytes are not displayed but are reserved for use by the Apple's special peripheral slots — one location for each slot 0 through 7. These locations are the specific locations involved (s = 0 for slot 0; s = 1 for slot 1; ... s = 7 for slot 7): \$0478 + s, \$04F8 + s, \$0578 + s, \$05F8 + s, \$0678 + s, \$06F8 + s, \$0778 + s, and \$07F8 + s.

In text mode, each character is represented in memory by a single byte (8 bits) of memory. The character displayed is determined by Apple's own special adaptation of the ASCII (American Standard Code for Information Interchange). The actual on-screen display is by a 8 high by 7 wide (including blank margins) array of dots.

In low-resolution graphics mode each 8-bit byte is treated as two 4-bit nibbles. The $2^4 = 16$ possible values of each nibble becomes 16 different color combinations, and the output is displayed as two colored blocks, one over the other. The color is controlled by a single nibble. Since there are 24 rows of characters this means there are 48 possible rows (vertical positions) for low resolution color blocks.

Pages 8-11 (Pages \$08xx-\$0Bxx): Lo-Res and Graphics Secondary Video Display Buffer

This area is seldom used as an alternate text display area. Layout is the same as the primary page, but is seldom used because there is no easy way to print the text here. (It must be POKED in or moved from page 1.)

Pages 8 upward (\$08xx upward): Default Apple-soft BASIC Program and Data Space or Default Integer BASIC Data Space

Note: Unless an overt use of LOMEM by the user alters the situation, user BASIC programs or data start at \$0800 (unless RAM Applesoft is in use).

Set LOMEM to start at \$1200 if Text/Lo-Res graphics page 2 is used. Start after the RAM version of Applesoft if you're using Applesoft without either an Apple with a language card, an Apple II+, or an Apple II with an Applesoft card.

Warning: If RAM Applesoft is used it extends far enough upward in memory to interfere with the use of Hi-Res Graphics Video Display Page 1. If Integer BASIC is used data starts here and works its way upward in memory.

If Applesoft BASIC is used, this space is normally occupied by Applesoft programs and data, with program statements on the bottom, data above the program and linkages to strings and arrays above that.

Warning: Note that as the program size increases, the data is pushed upward. \$1FFF is not the top limit of the program. It can expand upward until it meets the string data which expands downward from HIMEM (usually the beginning of the DOS), but after \$1FFF this program-related material begins to intrude upon the high-resolution graphics display space making it unusable for graphics purposes.

Pages 32-63 (Pages \$20xx-\$3Fxx): High-Resolution Graphics Primary Video-Display (HGR pg1)

It is conventional to describe the high resolution graphics video-display area as a bit-mapped area 280 dots wide by 192 dots high in which each possible dot position represents one bit in these pages of memory.

Since there are $280 \times 192 = 53760$ dot positions we must somehow map the 53,760 dot positions into 53760 bits of the 8K (32 pages of 256-bytes of 8 bits each) = 65,536 bits on these memory pages.

At first the mapping seems absurdly scrambled. If you are perceptive, you may finally detect an assignment pattern which is closely related to the mapping pattern used by text/low-resolution graphics.

This area, though eight times the size of the text screen buffer area, is organized in a conceptually similar fashion. It contains 8 text macrolines each 128 'standardized' characters long which break into 3 screen lines (top $\frac{1}{3}$, middle $\frac{1}{3}$, bottom $\frac{1}{3}$) plus 8 character positions leftover for allocation to peripheral slots.

However, in high-resolution graphics a 'standardized' character position is not represented by a single ASCII character. Instead it is an array 7 dot positions wide by 8 dot positions high, i.e. 8 slices

each containing 7 dot positions stacked one over another.

Thus the 40 'standardized' character positions also represent $7 \times 40 = 280$ dot positions. Each 'slice' of 7 dot positions is associated with one byte of memory, one dot/no-dot position per bit, with the eighth bit (the most significant bit) being a 'color bit'.

Note: On a black-and-white monitor a change in the color bit causes any dot within that byte of memory to shift $\frac{1}{2}$ position left or $\frac{1}{2}$ position right. This creates 560 distinguishable dot-positions across the screen and makes black-and-white plotting possible at a horizontal resolution of 560 bits — providing you program for it and don't use Apple's line-drawing software.

On a color monitor, a dot moving across the 560 distinguishable positions will change color in cycles of 4 colors: violet, blue, green, red/orange. This means that there are only 140 possible bit-mapped green dot positions, so the maximum, resolution for plotting in green (or any other color than black-and-white on a black-and-white monitor) is 140 dot positions across the screen.

On a color monitor if two adjacent colored dots are turned on simultaneously they will merge into a single larger, white-ish dot. The plotting technique used by Apple software uses this technique for plotting the color 'white'. Since there are 280 possible positions for these double-width dots, Apple's standard plotting technique achieves a 280 dot resolution across the width of the screen.

The individual 'slices' which make up a 'standardized' high-resolution character space are located 4 memory pages apart. Thus for the character at the top left corner of the screen, the topmost slice is represented by the byte at location \$2000, the next slice by the byte at \$2400, the next at \$2800, etc.

Since there are 8 slices (8 bytes of memory) stacked one above another per displayed 'standardized' high resolution graphics character, there are $8 \times 24 = 192$ lines of dots possible on the high-resolution graphics screen, so the screen display checks as 192 dots high by 280 dots wide.

It is from this pattern that we derive the initialy scrambled order of memory positions for the left edges of the individual lines of screen display which starting at the top line, goes as follows: \$2000, \$2400, \$2800,..., \$3800, \$3C00, \$2808, \$2480, \$2C80,... \$2380, \$2780, \$2B80, \$2F80, \$3380, \$3780, \$3F80, \$2128, \$2528,..., \$24A8, \$27A8, \$2AA8, \$2EA8, \$2050, \$2450,..., \$24D0, \$28D0, \$2CD0, \$2FD0.

Pages 64-95 (Pages \$40xx-\$5Fxx): Hi-Resolution Graphics Secondary Display Page (HGR pg2)

The interior layout is the same as HGR pg1 but \$2000 higher.

Pages before 150 (before Page \$96xx): Applesoft Strings or Integer BASIC Program

Unless an overt setting of HIMEM is used to override it, Applesoft strings work downward from \$BFFF if DOS not used or from the beginning of DOS if DOS is used. In a default case (when DOS is using 3 buffers) \$9600 is the beginning of DOS so strings work downward from here.

Unless an overt setting of HIMEM is used to override it, Integer BASIC puts its program in this same area with the end of the program at \$95FF and the beginning of the program pushing downward as far as necessary.

Pages 150-191 (Pages \$96xx-\$BFxx): Disk Operating System

When the Disk Operating System is booted on a 48K Apple it occupies locations \$9600-\$BFFF in the default case. In an Apple with less memory, the start of the DOS moves down by the amount of the reduction of memory. E.g., in a 32K Apple, the DOS would start at \$5600.

Warning: Note the interference with Hi-Resolution graphics page 2.

If DOS Maxfiles are set to a value other than the default value of 3, buffers added to or deleted from DOS will alter this boundary point. With maxfiles=6, DOS extends downward to \$8F57; with maxfiles=1, DOS extends downward only to \$9AA6.

DOS buffers normally occupy \$9600-\$9D00; the main body of DOS routine from \$9D00-\$AAC9; the file manager or I/O section of the DOS from \$AAC9 to \$B600; and the RWTS (Read-Write Track-Sector) routines from \$B600-\$C000.

Pages 192-207 (\$C0xx-\$CFxx): Special Hardware I-O Area

This area is reserved for Input/Output and 'slot' (peripheral) operations. It divides naturally into four sub-areas:

- \$C000-\$C07F Built-In I/O Locations
- \$C080-\$C0FF Peripheral Card I/O Space
- \$C100-\$C7FF Peripheral Card ROM Space
- \$C800-\$CFFF Expansion ROM Space
(Allocated to Currently Active Peripheral Slot).

Page 192 (\$C0xx) is divided into two half pages. The \$C000-\$C07F half-page contains special data and flag inputs (such as the keyboard, cassette pushbutton and game-control/joystick). It also contains strobe functions which activate special I/O activities and program-controllable 'soft-switches' and 'toggle-switches' which control such alternatives as video display of text vs. display of graphics; Lo-Res vs. Hi-Res graphics, Primary vs. Secondary video display page being displayed, all full page graphic display or mixed text-graphics display.

The \$C080-\$C0FF half-page is divided into 8, 16-byte chunks, each of which is assigned to one of the 8 peripheral slots (0-7) for use as Input/Output space for that peripheral.

Pages 193-199 (\$C1-\$C7) are allocated one page to each peripheral slot (1-7, but not slot 0) for its exclusive use by its own on-board PROM (Programmable Read Only Memory).

Pages 202-207 (\$C8-\$CF) is a 2K (8 page) area reserved for use by memory (usually a ROM) on a peripheral card. Only that memory on the card whose slot is currently active has access to the central machine.

Please note that the peripheral cards also have assigned to them additional individual bytes of RAM memory from the 'breakage' at the end of each line of the video display buffer areas.

Pages 208-255 (\$D0xx-\$FFxx): Used for Monitor and Interpreter ROM

Note: When the language card is used, ROM may be replaced by RAM into which firmware may be read and then protected against accidental writing to make it a de-facto ROM equivalent after initial loading.

The topmost part of this, pages 248-255 (\$F8-\$FF), are assigned to the monitor, which may appear in either of two forms: the (old) monitor ROM or the (new) autostart monitor ROM. The major differences between them are that the autostart version has had the autostart features added and has had the mini-assembler and single-step trace capabilities removed to make space for the additions.

In the Apple II+, the remainder of this area, pages 208-247 (\$D0-F7), is occupied by the Applesoft BASIC interpreter.

In the Apple II, the Integer BASIC, rather than the Applesoft BASIC, is built, and it occupies a smaller area, pages 240-255 (\$E0-\$FF). The remaining space, pages 208-239, is available for other ROMs such as the Integer BASIC 'Programmer's Aid #1'.

↑
Visa-Versa!

Examples

Using a Monitor Routine from Applesoft

```

1  REM *****
2  REM *
3  REM *      CASE STUDY NO. 1      *
5  REM * WHAT'S WHERE IN THE APPLE *
6  REM * ----- -- -- -- -- *
7  REM *
8  REM *****
9  REM
10 HOME : VTAB 7: PRINT "ENTER DECIMAL NUMBER";: INPUT N
15 HOME : VTAB 7: PRINT " DEC= ";N
20 MSP = INT (N / 256): POKE 0,MSP: REM  MSP => LOCATION 0
30 LSP = N - 256 * INT (N / 256): POKE 1,LSP: REM  LSP => LOCATION 1
40 POKE 60,0: POKE 61,0: REM  0 => PARAMETER A1
50 POKE 62,1: POKE 63,0: REM  1 => PARAMETER A2
60 CALL - 589: REM  ROUTINE TO HEX PRINT MEMORY FROM A1 TO A2 (0-1)
70 POKE 1064,160: POKE 1065,200: POKE 1066,197: POKE 1067,216
80 POKE 1068,189: POKE 1069,160: REM  POKE TO SCREEN "HEX = "
90 VTAB 11: PRINT "PRESS ANY KEY TO CONTINUE";: GET R$: GOTO 10

```

Analysis:

The best way to start is to look up the memory locations involved in the Programmers' Atlas:

1. Locations 0 and 1 seem to be usable for several different purposes.
2. Locations 60 and 61 are often used together as a two-byte general-usage 'Parameter A1' for many monitor subroutines.
3. Locations 62 and 63 are often similarly used as two-byte general-usage 'Parameter A2' for many monitor subroutines.
4. Location - 589 contains a subroutine which outputs a block of memory in hex format using parameters A1 and A2 to specify the starting and ending addresses of the block.
5. Locations 1064 through 1069 are located in the middle of text page 1. Anything POKed to them should appear as text on the screen of the Apple.

Now down to detailed analysis of the program:

1. Line 10 clears the screen, tabs part way down, and asks for and accepts as input a decimal number. It then reclears the screen and prints out ' DEC= ' and the value of the number accepted.
2. Lines 20 and 30 do a computation we have seen before. They break the integer value of

the number into two byte-sized pieces and put the more significant part (MSP) into location 0 and the less significant part (LSP) into location 1.

3. Lines 40 and 50 respectively put the address 0 into parameter A1 and the address 1 into parameter A2.
4. Line 60 calls the hexadecimal print subroutine used by the monitor for printout of the contents of any desired portion of memory. Parameter A1 tells it to start at location 0 (where the MSP of the number is located) and to print the hexadecimal contents of memory locations through that specified by parameter A2, which turns out to be only through location 1 (where the LSP of the number is located). Thus, only two locations are printed: that containing the MSP and that containing the LSP. These two locations contain the number which was to be printed in hexadecimal form.
5. Unfortunately the subroutine at - 589 also prints the starting memory location for the group of values that it prints out. This is unnecessary and confusing in the context of this use. The problem is resolved by having line 70 overprint the location on the screen where this undesired '0000 -' is printed with the identification ' HEX= '. Thus the Apple screen now shows ' DEC= ' and the decimal value and immediately below it

' HEX = ' with the corresponding hexadecimal equivalent.

6. Line 80 freezes the output on the screen by stopping the program until the user presses some key, then goes back to the beginning of the program to repeat the process.

Several comments are in order. First, the choice of memory locations 0-1 was purely arbitrary. They weren't being used for anything else and were easy to remember and POKE.

Second, the POKEd output onto the screen in line 70 might better be replaced by a BASIC 'PRINT " DEC = "' with appropriate prepositioning by TAB commands. However, this made a nice illustration of the direct POKE output onto the screen, so why not use it?

Finally, the monitor subroutine at -589 was not designed to do exactly what was wanted. It did something more which had to be undone by the overprinting operation.

Creating a Machine Language Program from Applesoft

```

1  REM *****
2  REM *
3  REM *      CASE STUDY NO. 2      *
4  REM *  WHAT'S WHERE IN THE APPLE*
5  REM *  -----
6  REM *
7  REM *****
8  REM
10 POKE 768,216: POKE 769,160: POKE 770,0: POKE 771,76: POKE 772,44: POKE 773,254
20 POKE 60,BEG - INT (BEG / 256) * 256: POKE 61, INT (BEG / 256)
30 POKE 62,EN - INT (EN / 256) * 256: POKE 63, INT (EN / 256)
40 POKE 66,DEST - INT (DEST / 256) * 256: POKE 67, INT (DEST / 256)
50 CALL 768

```

Analysis:

In a quick overview we note the following:

1. The first line POKes information into memory locations 768-773.
2. The next 3 lines each do similar computations of the type we have seen before: breaking a number down into two bytes — a more significant part, the quotient of an integer division by 256, and less significant part, the remainder of integer division by 256. First the computation is done on the value of BEG (the BEGinning of the block to be moved); next it is done on the value of EN (the ENd of the block to be moved); and finally it is done on the value of DEST (the DESTination of the block to be moved).
3. The results of these computations are POKed into memory locations 60,61 - 62,63 and 66,67. Finally,
4. The last line CALLs (transfers control to) memory location -768, the first location into which something was POKed at the beginning of the program. Since the last line

transferred control to it we may suspect that what was POKed into location 768, and the locations following it, was a tiny piece of program.

Now let's begin our normal analysis using the Programmers' Atlas:

1. At memory location 768 in the Atlas, we find the indication that the block of memory starting at that location is often used as a convenient location for user-written programs. The suspicion is reinforced as a working hypothesis, but not fully confirmed.
2. Locations 60 and 61 are listed together as a pair of 8-bit bytes: A1L and A1H. The L denotes the Low (or Least Significant Byte — LSB) and the H denotes the High (or Most Significant Byte — MSB) of two bytes normally used together to form the two-byte (16-bit) parameter A1. The Programmer's Atlas describes A1 as follows: "Monitor general-usage subroutine parameter A. Many uses include source pointer for monitor move subroutine." [A 'pointer' is an address which 'points' to a given location in memory.]

3. Line 20 uses variable BEG (for BEGinning) to compute the address of the beginning of the block of memory to be transferred, and puts it into the same memory locations as those used for general-usage parameter A1.
4. Line 30 performs similar computations on EN (the ENd address of the block of memory to be moved and puts the results into the same location used by monitor general-usage subroutine parameter A2.
5. Line 40 does the same again with DEST (the DESTination address) and puts the results into monitor general-usage subroutine parameter A4.
6. Could the MOVE subroutine, which is present as part of the monitor any time the Apple is running, be at the heart of the 'FAST MOVE' capability? Again we have preliminary suspicions, but lack confirmation. However, there is only one more line to the program. It does not call location 65068 or anything readily associated with -468, or with the 'MOVE' routine, wherever that may be. Instead it calls location 768, the first of such locations into which we POKEd something. Too bad! You can't win them all! However, let's not be too discouraged.
7. A CALL is a subroutine-type transfer of control to a piece of machine language code, so let's see what happens if we interpret the POKEs in line 10 which begin with a POKE to location as machine language. This involves a conversion from decimal format to machine language format. Perhaps they will make sense as a program, even though it seems unlikely that a program so short could accomplish block move.
8. The POKEs in line 10 do indeed describe a machine language program, beginning at hex location \$300. For those interested in the mechanics of this program, enter the monitor (call -151) and disassemble starting at location \$300 (300L).

Checking Software Locations from within a Program

```

10 REM *****
20 REM *
30 REM * CASE STUDY NO. 3 *
40 REM * WHAT'S WHERE IN THE APPLE*
50 REM * ----- *
60 REM *
70 REM *****
80 REM
100 SPD = 256 - PEEK (241): REM PRINTING SPEED
110 LEFT = PEEK (32): REM SAVE LEFT MARGIN
120 RIGHT = PEEK (32) + PEEK (33): REM SAVE RIGHT MARGIN
130 TP = PEEK (34): REM SAVE TOP MARGIN
140 BOT = PEEK (35): REM SAVE BOTTOM MARGIN
150 :
160 TEXT : HOME : REM RESET SCREEN
170 REM PRINT OUT SCREEN AND STATUS
180 PRINT "SPEED = ";SPD
190 PRINT "LEFT = ";LEFT
200 PRINT "RIGHT = ";RIGHT
210 PRINT "TOP = ";TP
220 PRINT "BOTTOM = ";BOT
230 :
240 REM SET SCREEN BACK
250 POKE 241, - (SPD - 256): REM LOAD PRINTING SPEED
260 POKE 32,LEFT: REM LOAD LEFT MARGIN
270 POKE 33,RIGHT - LEFT: REM LOAD RIGHT MARGIN
280 POKE 34,TP: REM LOAD TOP MARGIN
290 POKE 35,BOT: REM LOAD BOTTOM MARGIN
300 :
310 END

```

Analysis: Left as an exercise for the reader!

Useful Tables

Zero Page Usage

Decimal Hex	0 \$0	1 \$1	2 \$2	3 \$3	4 \$4	5 \$5	6 \$6	7 \$7	8 \$8	9 \$9	10 \$A	11 \$B	12 \$C	13 \$D	14 \$E	15 \$F
0 \$00	AS	AS	AS	AS	AS	AS	S	S	S	S	AS	AS	AS	AS	AS	AS
16 \$10	AS	AS	AS	AS	AS	AS	AS	AS	AS	S	S	S	S	S	S	S
32 \$20	M	M	M	M	M	M	MD	MD	M	M	MD	MD	MD	MD	MD	MD
48 \$30	M	M	M	M	M	MD	MD	MD	MD	MD	M	M	M	M	MD	MD
64 \$40	MD	MD	MD	MD	MD	MD	MD	MD	MD	M	DI	DI	DI	DI	M	M
80 \$50	MA	MA	MA	MA	MA	MAI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
96 \$60	AI	AI	AI	AI	AI	AI	AI	DAI	DAI	DAI	AI	AI	AI	AI	AI	DAI
112 \$70	DAI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
128 \$80	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
144 \$90	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
160 \$A0	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	DAI
176 \$B0	DAI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI
192 \$C0	AI	AI	AI	AI	AI	AI	AI	AI	AI	AI	DAI	DAI	DAI	DAI	AI	AI
208 \$D0	AI	AI	AI	AI	AI	AI	I	I	DAI	AI	AI	AI	AI	AI	AI	AI
224 \$E0	A	A	A		A	A	A	A	A	A	A					
240 \$F0	A	A	A	A	A	A	A	A	A							

A = Used by Applesoft
D = Used by DOS

I = Used by Integer BASIC
M = Used by Monitor

S = Used by Sweet-16 Interpreter

Apple ASCII Representation

		Inverse				Flashing				Normal							
										(Control)				(Lower C)			
Decimal	Hex	000	016	032	048	064	080	096	112	128	144	160	176	192	208	224	240
		\$00	\$10	\$20	\$30	\$40	\$50	\$60	\$70	\$80	\$90	\$A0	\$B0	\$C0	\$D0	\$E0	\$F0
00	\$0	@	P		0	@	P		0	@	P		0	@	P		0
01	\$1	A	Q	!	1	A	Q	!	1	A	Q	!	1	A	Q	!	1
02	\$2	B	R	"	2	B	R	"	2	B	R	"	2	B	R	"	2
03	\$3	C	S	#	3	C	S	#	3	C	S	#	3	C	S	#	3
04	\$4	D	T	\$	4	D	T	\$	4	D	T	\$	4	D	T	\$	4
05	\$5	E	U	%	5	E	U	%	5	E	U	%	5	E	U	%	5
06	\$6	F	V	&	6	F	V	&	6	F	V	&	6	F	V	&	6
07	\$7	G	W	'	7	G	W	'	7	G	W	'	7	G	W	'	7
08	\$8	H	X	(8	H	X	(8	H	X	(8	H	X	(8
09	\$9	I	Y)	9	I	Y)	9	I	Y)	9	I	Y)	9
10	\$A	J	Z	*	:	J	Z	*	:	J	Z	*	:	J	Z	*	:
11	\$B	K	[+	;	K	[+	;	K	[+	;	K	[+	;
12	\$C	L		,		L		,		L		,		L		,	
13	\$D	M]	-	=	M]	-	=	M]	-	=	M]	-	=
14	\$E	N		.		N		.		N		.		N		.	
15	\$F	O	-	/	?	O	-	/	?	O	-	/	?	O	-	/	?

Apple Text Screen Organization

Memory Layout of Text "Super-Line"

... First 40 characters ... (Top 1/3 of screen)	... Second 40 characters ... (Middle 1/3 of screen)	... Third 40 characters ... (Bottom 1/3 of screen)	... 8 ... 'slot' bytes
--	--	---	------------------------------

Logical Organization of Text Display Area

..... Super-Line 0 (SL00) - 120 displayable characters + 8 non-displayable scratchpad bytes	.
..... Super-Line 1 (SL01)
..... Super-Line 2 (SL02)
..... Super-Line 3 (SL03)
..... Super-Line 4 (SL04)
..... Super-Line 5 (SL05)
..... Super-Line 6 (SL06)
..... Super-Line 7 (SL07)

Screen Display Layout of "Super-Line"

00	- - - - - First 40 characters - - - - -
01
02
03
04
05
06
07
08	- - - - - Second 40 characters - - - - -
09
10
11
12
13
14
15
16	- - - - - Third 40 characters - - - - -
17
18
19
20
21
22
23

