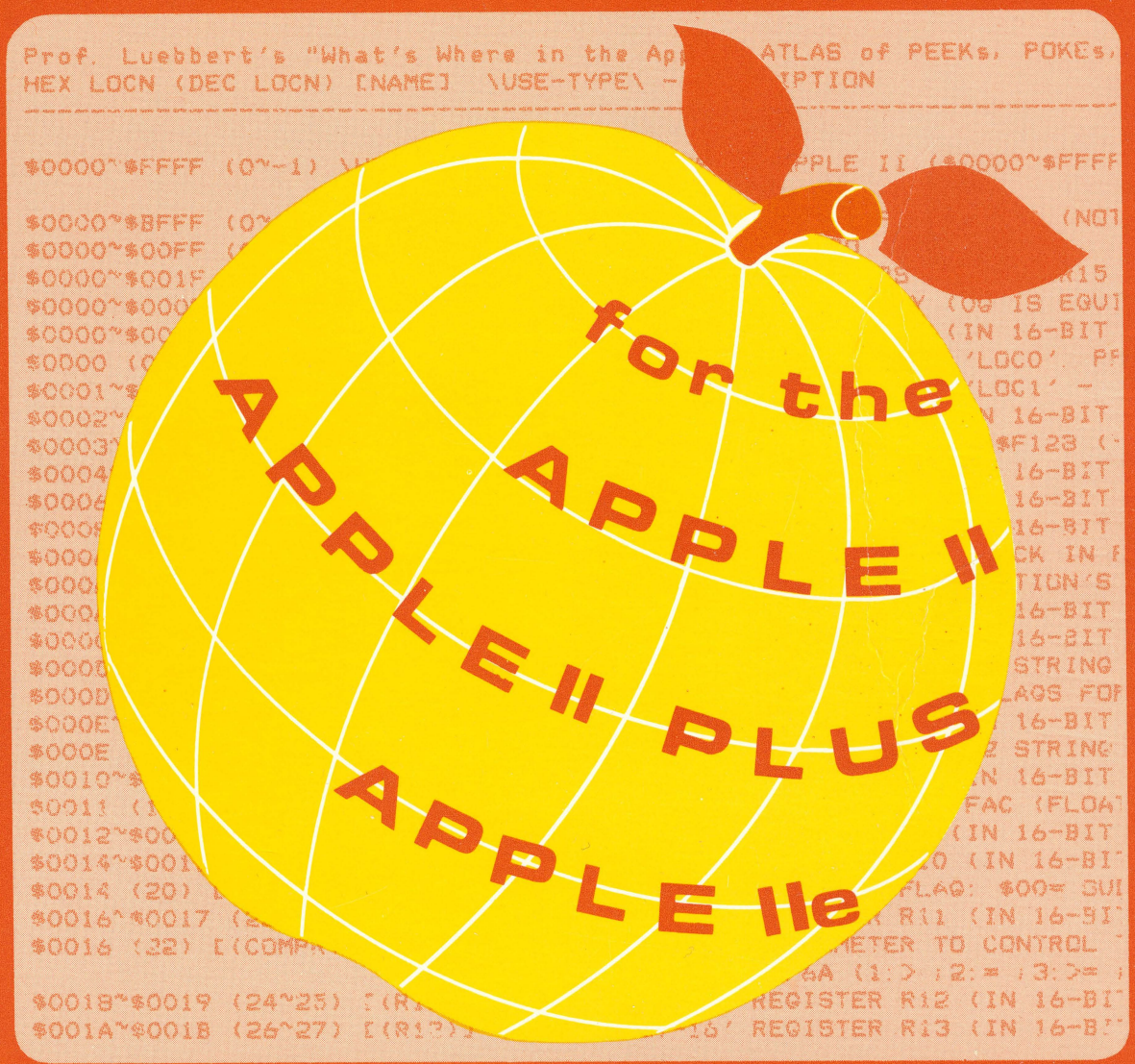
What's Where in the APPLE

A Complete Guide to the Apple Computer



Including:

the Atlas & the Gazetteer

by William F. Luebbert

What's Where in the APPLE

A Complete Guide to the Apple Computer

William F. Luebbert
President, Computer Literacy Institute
Adjunct Professor of Engineering
Dartmouth College, Hanover, New Hampshire

Apple //e Appendix
by
Phil Daley

MICRO INK
P.O. Box 6502
Chelmsford, Massachusetts 01824

Notice

What's Where in the Apple
A Complete Guide to the Apple Computer ISBN: 0938222-09-0
Printed in the United States of America
Printing 10 9 8 7 6 5 4

Contents

Author's 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 Suitor
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.

Publisher's Acknowledgements

A work of this complexity takes the dedicated labor of many individuals to bring it to completion. The following individuals each made significant contributions, above and beyond the call of duty.

Chief Editor: Marjorie Morse provided overall management of the project through its may iterations, in addition to performing a superb job of editing and rewriting.
Technical Editors: Ford Cavallari and Phil Daley gave guidance and worked diligently to produce technically accurate materials.
Layout and Production: Paula Kramer was instrumental in creating an easy-to-read product.
Typesetting: Emmalyn Bentley worked many hours on the detailed technical typesetting which resulted in the clean look of the book.

Robert M. Tripp
Chelmsford, MA

INTRODUCTION

You can get more out of your Apple — or any other computer with limited resources — by familiarizing yourself with its overall hardware and software environment. This book helps you get to know your Apple better. It provides you with information about hardware and software resources that are imbedded within the Apple at all times, but are usually hidden from users other than exceptionally well-informed and experienced system-level programmers.

What's Where in the Apple also introduces, explains, and demonstrates techniques for using this knowledge both in a BASIC language environment and in an assembly-language environment. Even more importantly, it introduces the concepts of programming in a Quasi-BASIC or system-specific BASIC environment. (Quasi-BASIC is BASIC augmented by non-BASIC Apple assembly-language firmware.) This environment requires little, if any, user-written machine code, but makes extensive use of machine code written and polished by the professional programmers who put the Apple II system together.

This work is organized in three parts:

1. Part I, the Programmers' Guide, is a comprehensive guidebook to the hardware and firmware organization and architecture of the Apple II system. It also discusses concepts and programming techniques you may find useful in exploiting the inner workings and hidden mechanisms of the Apple II system.

2. Part II, the Programmers' Atlas, is a detailed breakdown of the inner structure and organization of the Apple II system. Arranged in memory-address sequence, it contains specific information on each of over 2000 memory locations and blocks of memory locations inside the Apple II system. This information includes hexadecimal and decimal addresses, memory location name (if any), and a description of function(s) performed at each location or block of locations. Included are the major system-specific hardware locations in the Apple as well as the major subroutines, parameters, buffers, and code-entry points in the Apple system monitor, disk operating system, Applesoft and Integer BASIC interpreters.

3. Part III, the Programmers' Gazetteer, is a detailed breakdown of all named memory locations in the Apple II system, arranged in alphabetical order by memory location name. Like the main Atlas, it contains hexadecimal and decimal memory locations, memory location name, and the nature and description of the use of the location(s) by the Apple II system.

This information and these techniques should help you become a better-informed and more creative programmer.

It is amazing how much well-informed and creative programmers can get from a microcomputer when they use their knowledge of the inner workings and hidden mechanisms of the total system's hardware and software environment. It is disheartening to see how much time many inadequately-informed programmers waste because they lack this knowledge.

The information and techniques presented here should be of special value 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. When (and if) you get into real-time programming, adding your own specialized interfaces, performing activities which require the absolute maximum speed or absolute minimum memory utilization, the information here becomes critical if you don't want to waste time and effort wheel-spinning.

Some people may take a quick look at the Pro-

grammers' Atlas database listings in this book and decide that the Atlas and the system-specific programming techniques are tools exclusively for systems programmers and machine-language (or assembly-language) programmers. Because these techniques can be extremely useful to such programmers, this belief has just enough superficial truth in it to be a serious conceptual and practical error.

Though the Atlas does provide a great deal of information useful in machine-language programming, it still offers at least equal assistance to the BASIC programmer. BASIC programmers who are non-machine-language programmers may want to take full advantage of the capabilities of the computer by exercising direct control over the hardware and the machine-language firmware which is normally resident in the hardware.

Often you can exercise this control by using the information in higher-level language programs and by changing control parameters in programs that are executed by the system itself as part of its normal operations. Other times it may require the use of machine-language subprograms which are accessed by PEEKs, POKEs, and CALLs which are not themselves machine language but are written in a higher-level language, BASIC. Even then, the only machine-language programs used may be those already available in firmware.

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.

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 Apple 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.

Chapter I
There's More In Your Apple II System Than You May Think

1.1
The Apple System Environment — Hardware and Firmware

The Apple II system environment consists of the system hardware, plus a great deal of software provided by the manufacturer, which extends the system's capabilities. The most important parts of this software are often called 'firmware.' (*Note:* Firmware is software that is a permanent part of the computer operating environment. It is always available and may be considered an extension of the system hardware, available regardless of what kind of problem the computer is attacking and what language it is using. Software that is put into a ROM (Read-Only Memory) and is available without any special setup procedures is a prime example of firmware.)

The 'firmest' of the firmware in the Apple II is the system monitor. Without the monitor you could not load other programs into the system, nor make effective use of the keyboard for input, or the display screen for text or graphic output; the system would be functionally inoperable.

However, the monitor is not the only piece of firmware you'll normally find imbedded in the Apple every time you try to run a program. To use the BASIC language you need a BASIC interpreter, either Integer BASIC or Applesoft BASIC. To use the disk sub-system you need disk operating system firmware, usually known as DOS firmware.

These three major software/firmware packages each contain a gold-mine of carefully-written routines which can make it possible to write better, faster-running programs. However, most of these resources, and many of the hardware-specific characteristics of the Apple system remain hidden away and are not readily available to the typical Apple user. Oftentimes even finding out about their existence and capabilities becomes a task worthy of the talents of Sherlock Holmes. Sometimes only the manufacturer's 'systems programmers' learn about some of the features.

Even within a single module of firmware there can be a great diversity of routines useful to the average programmer. For example, in addition to the program-loading, input/output, and graphics functions alluded to earlier, the Apple II system monitor (old version) contained routines that performed the following functions: moved blocks of memory from place to place, verified that one block of memory contained the same contents as another, simulated the existence within the Apple of a 16-bit microprocessor (the 'Sweet-16'), single-stepped a machine-language program, assembled a machine-language program with mnemonic operations codes and any of several kinds of machine addressing into binary code which will run as a program on the computer, disassembled binary code back into mnemonic form, converted decimal inputs to hexadecimal display or binary internal formats or *vice versa*, etc.

Similarly, the Applesoft BASIC interpreter (or any other higher-level language interpreter) is usually also a gold-mine of useful software. For example, it contains software packages for implementing all the operations of floating-point arithmetic. (Floating-point arithmetic is used for numbers which contain decimal points or scientific power-of-ten notation.)

1.2
Making Hardware and Firmware Resources Accessible

The Programmers' Guide provides a framework for understanding both the overall organization and structure of the Apple system and those programming techniques which exploit that knowledge. The Atlas and Gazetteer provide supplementary detailed reference information you need in actual programming.

This detailed information is presented in 'Geographical' (Memory Map) order in the Programmers' Atlas (Part II) with an alphabetic-by-name Programmers' Gazetteer (Part III) to provide an alternate means of retrieval. An optional diskette version of the database together with a retrieval program provide machine-implemented selective retrieval as well (see page 4).

The information in both the text and on the diskette can stand alone as sources and techniques for making more effective use of the Apple II. You can learn a great deal about system-specific programming techniques for the Apple II and about its hardware/software architecture from the text. You can find out how particular sections of memory are used and about the characteristics of software in particular areas of memory. You can use the Atlas database tables or printouts from the diskette to identify software by specific names. And you can search the diskette database using the retrieval program on the

diskette to find hardware locations, parameters, or software descriptions which use keywords you're interested in exploring.

For example, if you're interested in all memory locations and software in the Apple that relate to the 'slots' used for plug-in of auxilliary cards, you could use the diskette to conduct a keyword search on the word 'slot'. To find out about hi-res graphics you could search on the word 'HI-RES'. (In both cases you would be flooded with more information than you are likely to want or use, but it would all be information relative to your request. However, with a little practice you will learn to narrow your requests to exactly the information you want.) You could, in principle, achieve the same results more laboriously (and with a higher human-error rate) by scanning through the forty-odd pages of Apple Atlas database printout in Part II of this book.

The Atlas will also help you find information you may already have available in sources such as your language-oriented reference manual (e.g., the *Applesoft Programmers' Reference Manual*) or a systems reference manual (e.g., the Apple Reference Manual).

You may want to know when you should use the diskette database and retrieval program and when you should use the tabular printouts in Parts II and III of this booklet. The answer is simple; if you already know about where in the Apple system a particular parameter, subroutine, or capability should be located, it is probably worthwhile to go to the appropriate area of memory in the tabular printout (Part III) and use the diskette only if you can't find what you are looking for. (After all, the information might be somewhere else in the Apple where you didn't expect it.) If you already know the standard Apple name for a parameter or subroutine, it is probably best to look it up in the alphabetic-by-name tabular printout (Part III). If you don't find it, try the diskette — it is often more difficult to determine standard Apple names for memory locations or subroutines than to find out information about what they do.

In some cases the information on the database diskette may be more brief than that in the tabular printouts. This was necessary to squeeze the information into the available space on the diskette. A larger version of the Programmers' Atlas database and retrieval program is available *via* timeshare from the Apple information library (APPLELIB ***) on the Dartmouth College timeshare computing system. This system is accessible to any qualifed user from anywhere in the nation *via* the communications facilities of TELENET.

Chapter II
The World of System-Specific Programming

2.1
BASIC Doesn't Have To Be A Straightjacket; Neither Does Assembly Language

Often inadequately-informed programmers feel they are boxed-in by the characteristics and limitations of the BASIC language, not realizing that the versions of BASIC available on the Apple (and many other microcomputers) are not as limited as they think. These versions do allow you to access and exercise significant direct control over the hardware and firmware of your system.

This work emphatically rejects the viewpoint that when using BASIC you must give up control of what is happening in the software and hardware of your computer.

This work also emphatically rejects the viewpoint that BASIC and assembly-language programming are such totally separate worlds that the programmer must choose one or the other, but never mix them in the same problem. It rejects the viewpoint that you must become an expert assembly-language programmer before you can understand and make effective use of the inner workings and hidden mechanisms of the software and hardware of your computer.

Instead, this book adopts the viewpoint that with the Apple (and other microcomputers) you can readily shift back and forth along a continuum of possibilities from (nearly) system-independent BASIC, to system-specific BASIC, to BASIC augmented by assembly or machine language, to assembly language using BASIC input-output and service routines, to full use of assembly language.

In fact, *What's Where in the Apple* suggests that the best results are obtained by taking a careful look at the circumstances and at the nature of the problems being attacked before deciding where to position yourself on this spectrum of alternatives. You'll discover new capabilities you can use in your own programming which will challenge your creativity and increase your willingness to undertake more difficult and interesting programming tasks.

It is amazing how many capable and well-informed microcomputer users fail to appreciate the full significance of system- and machine-dependent features (such as PEEKs, POKEs, and CALLs) built into their versions of BASIC. Often they may see interesting, but difficult-to-understand, published programs for the Apple which are made up almost exclusively of these commands. These programs are often written by highly experienced assembly-language programmers who use techniques not commonly covered in academic textbooks or computer programming courses. Many programmers get the false impression that they need esoteric knowledge if they do more than use an occasional PEEK, POKE or CALL.

2.2
System-Specific Programming: A Programming Approach for BASIC and Assembly-Language Programmers

Programmers often mistakenly assume that to take effective advantage of the inner workings and hidden mechanisms of system software and hardware capabilities they must abandon BASIC and become assembly-language experts. Under these circumstances it is not surprising that they don't feel strongly motivated to learn about machine-language code — information which is in their computers and potentially available for their use every time they run a BASIC program.

Assembly-language programming is not everyone's cup of tea. Many excellent programmers shy away from assembly and machine-language programming because they do not want to get bogged down with its limitations.

Well-written assembly-language programs do often run faster and use less memory than BASIC programs, but they usually take longer to write and longer to test and debug than BASIC programs. Human error rates in writing assembly language are often high. Assembly-language programs are harder to read and understand than BASIC programs. And, they are even harder to modify and update without extremely careful documentation — and most assembly-language programs are difficult to document well. There is also a significant investment in time and effort involved in learning how to control the operation of an assembler and how to use it effectively.

Often programmers associate these limitations not just with full-scale assembly-language or machine-language programming efforts, but with all aspects of hardware-dependent and system-specific programming. As a result many avoid any programming effort that seems to have any whiff of involvement with machine language or the details of their computers' internal architecture.

At one end of this spectrum you adhere so closely to ANSI BASIC standards that you greatly improve the chance that programs will be transferrable from one model or manufacturer of computer to another. But by completely eschewing any system-specific programming, you eliminate the effective power of your system. For example, all graphics and all sound-producing capabilities in the Apple are definitely system-dependent.

At the other end of this spectrum, assembly-language programming, you can save memory and improve the speed of response. It is not uncommon for assembly-language programs written by good programmers to have three to ten times the speeds of those of Applesoft BASIC. In occasional special cases speed gains of 10,000 times or more have been reported. However, assembly-language programming can be frustrating. It may be hard to write, hard to read, hard to maintain, hard to update and hard to move to other systems. You can, and usually do, exercise very direct and intimate control over the machine at a level of nit-picking detail that sometimes is as infuriating in its demands upon time, effort and human accuracy as it is powerful in releasing the capabilities of the machine. Anywhere on this continuum (except at the end where you deliberately ignore many of the features of BASIC to promote transferability of programs from one type of computer system to another) you can exercise any necessary degree of direct control over the performance of the system *while operating in a BASIC environment*. Although it is well worthwhile to learn to write programs in assembly language and to use an assembler, you can take significant advantage of system-specific and hardware-dependent capabilities without ever writing a program in assembly language or using an assembler.

2.3 *A Step-by-step Approach for BASIC Programmers Learning To Take Advantage of System-Specific Capabilities*

Programmers who know little or nothing about assembly- or machine-language programming can, as a first step, learn to take advantage of assembly/machine-language coding which is supplied in the firmware and other software of their Apple system by merely finding out where it is and what it does. Initially you can start by using PEEKs and POKEs documented in Apple manuals to change system hardware states or monitor parameters. Later, you can add use of CALLs to access monitor subroutines which can be used without any knowledge of machine language.

There are many additional routines in system firmware which can be used only if you know how to interface with them by putting appropriate information into particular hardware registers and getting results from those registers. You can perform this procedure indirectly without learning assembly or machine language, but it is often more convenient to look up a few machine-language instructions in the Apple Programmers' Reference Manual (or a book on 6502 assembly-language programming) and use them to load or unload the registers. You can do this without ever dropping out of BASIC by manually converting the few instructions needed to set-up or use calling parameters into BASIC PEEKs, POKEs and/or CALLs.

If you are willing to write an occasional assembly-language subroutine and imbed it in BASIC programs (in situations where the limitations of BASIC are most galling and the advantages of assembly/machine language are highest), you can improve the running speed or input/output capabilities of your program and more. This advantage can be achieved even though you write the majority of every program in BASIC. You can even write entire programs in assembly or machine language, but disguise them to the computer as BASIC programs, taking advantage of some of the conveniences of BASIC such as its very easy-to-use input-output features.

Of course, all these techniques are discussed at some length and illustrated with examples and case studies in the Programmers' Guide.

The information and techniques provided are system-specific. Unless specifically mentioned, the standard monitor, as opposed to the Autostart monitor, is assumed. It is also assumed that you have a full 48K of memory when you are using the disk operating system or high-resolution graphics. The higher-level language is Applesoft BASIC, although there are frequent references to Apple Integer BASIC as well. Most of the concepts are also applicable in principle Pascal, but since there are major differences between Pascal firmware and the 'standard' Apple environment for Applesoft, Integer BASIC and/or assembly- and machine-language programming, you must be very careful in extrapolating the information in this guide.

Most of these techniques do not involve any machine-language or assembly-language programming. Some of the more advanced techniques described involve occasionally writing very short machine-language links between ex-

isting machine-language firmware in the Apple and your BASIC programs. These links are then translated to Applesoft BASIC to make the machine-language firmware subroutines part of your BASIC programs.

2.4 When Should You Use Assembly Language?

In principle, assembly language, or machine language, is the most powerful language available to a given processor. Within the limits of human programming time, patience, and skill, it allows the most intricate manipulations of data, the smallest program size and the fastest execution time possible. This is why on most microcomputers the system monitor, disk operating system and BASIC interpreter are written in assembly language.

While it is true that anything you can do with the hardware of any given microcomputer can be done using that computer's machine language, most programmers today (with the possible exception of systems programmers and a few machine-language fans) quite properly use BASIC, Pascal or some other higher-level semi-machine-independent language for most of their programming. Machine-language programming can (but does not always) increase speed and/or decrease memory requirements quite dramatically. It is particularly useful when you wish to do bit manipulation, interrupt processing and other hardware-related activities. However, it involves too much onerous detail, is too prone to human errors, and is too difficult to read and maintain to make it the language of choice for most programmers. This is true whether they program maxicomputers, minicomputers or microcomputers.

Those who program large machines with huge amounts of memory, disk storage space and support software seldom need to expend the time and effort needed to pay close attention to the nitty-gritty details of the hardware and memory utilization of their computers. They usually make their programming as machine-independent as possible. In contrast, microcomputers — especially personal microcomputers — are often severely limited in internal memory, external storage and software support. Microcomputers seldom have the vast panoply of higher-level software, the sophisticated operating system and the large volume of high-speed external storage so useful in making machine-independent programming convenient and feasible for tough jobs.

Thus those who program microcomputers often find it necessary to pay careful attention to the hardware and software environments of their machines in order to do more with less — albeit at some cost of generality and transportability of their programs to other brands and models of computers.

Knowledge of specific characteristics of a machine's hardware, monitor, disk operating system firmware, or BASIC interpreter allows microcomputer programmers to perform complicated tasks easily. This is why the versions of BASIC supplied with most microcomputers contain specific provisions for interfacing with the hardware/software environment underlying their BASIC interpreters — e.g., PEEK and POKE commands — while the versions of BASIC available with maxicomputers often do not include such provisions.

2.5 When Should You Use System-Specific Quasi-BASIC?

When you begin to develop sophisticated programs which involve heavy use of computer graphics, sophisticated man-computer interfaces and real-time interfaces with the outside world, you often tend to get into programming situations which stretch the capabilities of a microcomputer to its practical limits.

For example, implementing graphics usually involves processes which extend beyond the system-independent capabilities of most microcomputer languages. There is little practical standardization between the graphics commands used in the various versions of BASIC which are today in widespread use on personal microcomputers. Although there is some effort being expended on standardization for the future, most microcomputers have their own extensions to BASIC which reflect hardware dependence. It takes significant software, memory, and processing time to raise computer graphics to system-independent levels.

Usually, as in the case of the Apple, machine-independent BASIC is modified and extended with pseudo-BASIC extensions which allow some degree of decoupling from the nitty-gritty details of hardware implementation, but are anything but system-independent. Thus, like it or not, when you try to do advanced or creative programming on a microcomputer, you are forced into the world of system-dependent programming.

The more interesting and sophisticated the problems you attack become, the more likely you are to find that the pseudo-BASIC extensions

begin to break down as the solutions to all your problems. Thus the Applesoft reference manual has several pages of PEEKs and POKEs — machine-level interfaces — which describe how to perform useful functions related to computer graphics.

Graphics provides understandable and visual examples of how advanced programming projects on microcomputers get forced into system-dependent programming. However, many aspects of man-machine interaction, the control of external devices, database management, word-processing and other areas of computer program development share with graphics the tendency to force you towards system-dependent programming. Such programming is often slow and inefficient. When you get into real-time programming, then system-dependent programming and the information in a programmers' atlas becomes critically inportant — whether or not you are willing to program at an assembly-language or machine-language level.

An example of such a problem arose when I was writing a program to provide a controlled display of optical illusions on the Apple. There is a well-known optical illusion in which lines of equal length with arrowheads appear to be of different lengths. The illusion appears if one line is provided with arrowheads pointing outward and the other has arrowheads pointing inward.

I was using game paddles to control the angle and length of the lines and wanted smooth, rapid animation when I changed the conditions. The easiest way to accomplish this was to move a copy of the currently displayed picture to the other display page of memory, change it while it was invisible, switch display pages and do it again. A BASIC program to copy the pages was much too slow.

I could have written a fast machine-language program which would do the page copying very rapidly, but information in the memory atlas made it unnecessary. The Programmers' Atlas told me that at memory location −468 (also known as memory location 65068 or $FE2C) in the Apple, there was a monitor subroutine which would do the job for me. It gave me all the information I needed to set up the computer to use it. You will learn how to do this and many other similar tasks.

These situations often occur when you undertake 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, word-processing, or any of a wide variety of interesting tasks. The knowledge available in a programmers' atlas becomes much more important.

When you get into real-time programming, adding your own specialized interfaces, performing activities where you must get the absolute maximum speed or make the most effective possible use of limited memory, then systems programming information becomes critical. This is true whether or not you ever intend to do any significant amount of assembly-language programming.

2.6 Examples of System-Specific Quasi-BASIC Programming

When you look at interesting programs described in magazines which provide programmers with ideas, information, and software, you often find programs which purport to be written in BASIC, but which in fact seem to be written neither in ANSI (American National Standard Institute) BASIC, nor in machine language. Instead, they are written in some kind of a strange hybrid of the two made up of PEEKs, POKEs, and CALLs.

The implementations of BASIC and Pascal in most microcomputers (including the Apple II) provide these and other features to facilitate system-dependent programming in BASIC or other higher-level languages. These system- and hardware-interface command-statements let you access instructions and data almost anywhere in the computer: in the monitor, in its operating system, inside the BASIC interpreter, in the peripheral interface areas, and in other parts of the computer hardware. But you must have adequate knowledge of how the hardware and firmware of the system are organized to make use of the potential provided.

This work will use several case study sample programs to allow you to determine whether you have the background to make effective use of these capabilities. Take a look at these programs now. You should be able to analyze them in detail and explain exactly how similar programs work by the end of Part I.

Often, as in case study sample programs 1 and 2, you can achieve highly significant results interacting with system firmware and hardware without any assembly or machine-language programming at all.

Case Study Sample Program 1

Screen and Printout Status Inquiry Subroutine

```
60000 PRINT "PRINTING SPEED ="; 256
      -PEEK(241)
60001 PRINT "LEFT MARGIN =";PEEK(32)
60002 PRINT "RIGHT MARGIN =";PEEK(32)+
      PEEK(33)
60003 PRINT "TOP MARGIN =";PEEK(34)
60004 PRINT "BOTTOM MARGIN =";PEEK(35)
60005 RETURN
```

Note: The value and convenience of this program can be increased by saving the current status of variables at the beginning of the program by means of POKE statements, resetting screen parameters by means of TEXT:HOME, then restoring the parameters by a second set of POKE statements at the end of the subroutine.

Case Study Sample Program 2

Quick Decimal-to-Hexadecimal Conversion

```
10 HOME : VTAB 7 : PRINT "ENTER
   DECIMAL NUMBER:"; : INPUT N : HOME :
   VTAB 7 : PRINT " DEC= ";N
20 MSP = INT(N/256) : POKE 0, MSP : REM
   MSP =    Location 0
30 LSP = N - 256 * MSP: 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 Hex Print Memory from
   A1 to A2 (0 to 1)
70 POKE 1064,160:POKE 1065,200:POKE
   1066,197:POKE 1067,216:POKE
   1068,189:POKE 1068,189:POKE 1069,160:
   REM POKE to output " HEX = "
80 VTAB 11: PRINT "PRESS ANY KEY TO
   CONTINUE"; GET R$:GOTO 10
```

Other times, as in case study sample program 2, you need to write only a few instructions; for example, to change a hardware mode or load hardware registers, and to insert them via POKEs and invoke them *via* a CALL.

Case Study Sample Program 3

Fast Move In Applesoft
Roger Wagner
(Published in July/August 1980 issue of
CALL A.P.P.L.E.)

```
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/265)
40 POKE 66,DEST - INT(DEST/256)*256:POKE
   67, INT(DEST/256)
50 CALL 768
```

Case study sample programs 3 and 4 are variants on this program applying the same monitor move subroutine to the specific task of moving pictures from text or low-resolution graphics page 1 to text or low-resolution graphics page 2. This can be a valuable function. There is no easy way to print directly onto text page 2. You can get information there only by POKEing in the fashion we demonstrated earlier or by moving it from page 1 as this program does.

Neither case study sample program 3 nor 4 uses machine-language. Sample program 4 uses the monitor 'SAVE' and 'RESTORE' routines to perform the function performed by machine language in sample program 1. Sample program 4 simulates keyboard entry of monitor commands to accomplish this function.

Case Study Sample Program 4

Text or Low-Resolution Graphics Fast Page Move
('SAVE' and 'RESTORE' Indirect Set Up)

```
100 GOSUB 500 :REM Copy page 1 to page 2
    and display there
110 REM Now use standard print instructions
    to print invisibly on text page 1 while text
    page 2 is being displayed
200 POKE -16300,0 :REM Display new
    (modified) page 1 ...etc.
500 REM: Subroutine to copy page 1 to page 2,
    then display it as page 2
510 CALL -182:POKE 71,0:Call -193 :REM
    Set Y-Reg = 0 using 'SAVE' and
    'RESTORE'
520 POKE 60,0:POKE 61,4 :REM Set parameter
    A1 = 1024 ($0400 - start of page 1)
530 POKE 62,255:POKE 63,7 :REM Set
    parameter A2 = 2047 ($7FF - end of page 1)
540 POKE 66,0:POKE 67,8 :REM Set parameter
    A4 = 2048 ($0800 - destination)
550 CALL -468 :REM Copy page 1 to page 2
560 GOSUB -16299 :REM Display copy from
    page 2
```

```
570 RETURN
580 REM: This subroutine may be used equally
    well for text or low-resolution graphics as
    it is. For high-resolution graphics use
    memory location 8192 instead of 1024,
    16383 instead of 1023 and 16384 instead of
    2048.
590 REM: WARNING! Don't try the inverse
    move from page 2 to page 1 unless you
    have made sure that the scratchpad
    memory locations used are properly set to
    be consistent with the page 1 values which
    will be wiped out.
```

Chapter III
PEEKing Can Be Informative

NOTE: It is not necessary to know anything about binary numbers or about the means of representing information in binary and hexadecimal form to follow this section, but it may help you to understand the 'why' of certain assertions made in this chapter. Chapters 6 and 7 cover the entire topic of information representation inside the Apple from an introduction of basic concepts, to details of importance only to the most sophisticated systems programmers. If you feel uncomfortable with binary or hexadecimal numbers mentioned here, please refer to Chapters 6 and 7 — especially Section 6.3 (Bit-Oriented Information Representation and Addressing in the Apple II System) for more background.

3.1
What Does A PEEK Do?

3.1.1 The BASIC Idea

A PEEK lets a programmer 'peek' into memory to determine the current information stored in a particular memory location. How that information is interpreted and used depends upon the context in which the PEEK is used. For example,

```
      LET X = PEEK(32)
and   LET Y% = PEEK(32)
```

use the PEEK in a numeric context. They treat the information bits in memory location 32 as a binary number and assign its numeric value to the real variable 'X' and the integer variable 'Y%' respectively. You can use this numeric value as part of an arithmetic expression or print it out directly without assigning it to a memory location, e.g.

```
      LET Z = 2000 + PEEK(32)
or    PRINT PEEK(32)
```

The number returned by a PEEK is always in the range 0-255. Why? The PEEK gives you the contents of one word of memory, which in the Apple consists of a single 8-bit byte. Eight bits can represent 2^8 combinations from 00000000 (0) to 11111111 (255).

You can also use a PEEK in an alphanumeric context. For example,

```
      LET A$ = CHR$ (PEEK(32))
```

treats the information in memory location 32 as the code-bits of an alphanumeric (ASCII) character. It assigns the character represented by those bits as the new value of the string variable A$. You can use the PEEK in a string expression or print it out directly without assigning its value to a string variable name, e.g.

```
      LET Z$ = B$ + CHR$ (PEEK(32))
or    PRINT CHR$(PEEK(32) )
```

Please note that the ASCII character obtained from a PEEK can be any code combination from 00000000 to 11111111 and thus need not be a 'printable' alphabetic or numeric character. For example, it might represent the character 'Control-G' which 'prints' by sounding a beeping noise.

It is also possible to treat the information as a hexadecimal number or as part of a machine-language instruction. These treatments will be deferred until we discuss the use of hexadecimal information forms and/or elementary machine language concepts.

3.1.2 Formal Statement and Hardware Implementation

To be specific and precise we may say that the PEEK commands of both Applesoft and Apple Integer BASIC are built-in functions that use a single argument — the decimal number address of the memory location which the programmer wishes to PEEK from. The value of the function becomes the contents of the memory location specified.

The context of use determines how the eight binary bits that make up the contents of that memory location will be interpreted: as a number, as an alphanumeric character, or as a part of a machine-language computer instruction.

For the sophisticated programmer who mixes BASIC and assembly-language programming at the hardware-register level it is also significant that a PEEK also leaves these 8 bits in the hardware A-register. We shall see later that the PEEK is merely a BASIC language function which allows you to perform the machine-language instruction 'LoaD Accumulator (LDA).' NOTE: In machine language you specify memory addresses in binary/hexadecimal format, while the Apple versions of BASIC use only decimal numbers. Thus the PEEK function also performs an implicit decimal-to-binary conversion needed to achieve the required functional equivalence.)

3.2
What Can You Learn from PEEKing?

Programmers PEEK to get information helpful in their programming or use of the computer.

3.2.1 Example: Find Current Cursor Position

Suppose that in a computer program you wanted to determine at exactly what line on the screen the program was currently printing. This is determined by the cursor vertical position. You could look in either the *Applesoft Programming Reference Manual* or the Programmers' Atlas and find that the current cursor vertical position is maintained by the system monitor in memory location 37 (decimal). Thus the statement

```
LET CV = PEEK(37)
```

in your program assigns the current Cursor Vertical position to the variable CV.

3.2.2 Example: Find Peripheral Slot Currently Active

Let's take a couple of other examples involving PEEKs not documented in the *Applesoft Programming Reference Manual*. For example, to find out which peripheral slot is currently active (Slot 0 is active if no peripheral device has been activated), you may PEEK into memory location 2040. Due to the internal design of the computer, this location does not contain the number in decimal form, but 192 + (SLOT #).

```
SLOT = PEEK (2040) - 192
```

NOTE: This location actually holds a hexadecimal number which is used in addressing ROM memory associated with the peripheral slot. This hexadecimal number is $CS where this is to be used as the more significant byte of the two-byte hexadecimal hardware address $CS00 (decimal 49152 + 256*S). S is the peripheral slot number. Since C is the hexadecimal symbol for 12, $C0 = 12*16 = 192 and $CS = 192 + S (decimal with S restricted to existent slot numbers, i.e. 0 > = S < = 7).

3.2.3 Example: Determine Printout Speed-Delay

Occasionally you may find that printout from the Apple seems unnaturally slow. This can happen if someone or some program has changed the SPEED variable. (The SPEED variable is used by the monitor to determine whether a delay should be inserted in the printout process to slow that process. This is sometimes done so that printouts or listings will appear, not near-instantaneously, but at a readable speed.) The Programmers' Atlas tells us that the SPEED variable is stored by the system monitor in two's-complement form (which measures the amount of delay to insert into printout operations) under the name 'SPDBYT' at memory location 241 (decimal). The two's complement for an 8-bit word is equivalent to 256 — the original number. Thus

```
PRINT (256 - PEEK(241) )
```

will print out the value of this variable. If SPEED < 255 an unnecessary delay is being inserted by the system monitor during every printout.

3.3 Case Study In Depth — Screen and Printout Status Inquiry

This case study illustrates a simple use of PEEKing.

Figure 3.3A

Case Study Sample Program 1

SCREEN & PRINTOUT STATUS INQUIRY SUBROUTINE

```
60000 PRINT "PRINTING SPEED =";256
      - PEEK(241)
60001 PRINT "LEFT MARGIN =";PEEK(32)
60002 PRINT "RIGHT MARGIN =";PEEK(32)
      + PEEK(33)
60003 PRINT "TOP MARGIN =";PEEK(34)
60004 PRINT "BOTTOM MARGIN ="; PEEK(35)
60005 RETURN
```

NOTE: The value and convenience of this program can be increased by saving the current status of variables at the beginning of the program. You PEEK them, then POKE the result of that PEEK into a temporary storage location. Then you reset the screen parameters to standard values by means of a monitor subroutine. You can execute TEXT: HOME, then PEEK the stored screen parameters and POKE those values back as the current screen parameters.

To analyze a program or subroutine which uses PEEKs, POKEs and/or CALLs, a good first step is to look up in the Programmers' Atlas the locations which are PEEKed, POKEd and/or CALLed.

In this case you'll find that the analysis is trivial. The program simply prints an identification for a group of monitor parameters, PEEKs at and prints out their current values.

3.4 Double PEEKing

Sometimes it is necessary to obtain or to change information which has a greater range of possibilities than the range 0 to 255 available with a single PEEK or POKE. For example, there

are 65,536 addressable memory locations in the Apple. BASIC line numbers may be anywhere in the range 0-32767 in Integer BASIC and up to 63999 in Applesoft. Integer numbers can have values from -32768 to +32767 in either Integer BASIC or Applesoft. Applesoft will also accept integer numbers in an address format in the range 0 to 65535. Thus if you want to PEEK into memory to find an address, a BASIC line number, or to look at an integer variable, a single PEEK can't give you the whole story.

To handle such situations you need to PEEK or POKE to more than one memory location at a time and combine the results into a single decimal number. The 'magic formula' for this combination is

PEEK(memory location) + PEEK(memory location + 1) *256

3.4.1 Example: Finding the Line Number of a BASIC Error

In its discussion of commands related to errors, the *Applesoft BASIC Programming Reference Manual* gives an example of how to find an error, but does not explain the hows and whys fully. It gives the 'magic formula'

340 X = PEEK(218) + PEEK(219)*256

and states: "This statement sets X equal to the line number of the statement where an error occurred if an ONERR GOTO statement has been executed."

The explanation is simple. Like Apple addresses, Applesoft line numbers have a permissible range from 0 to over 60,000. To allow adequate possibilities to give each line number a unique representation takes 16 bits or two eight-bit bytes or two words of Apple computer memory.

NOTE: 15 bits allow 2^{15} *** =32,768) possibilities, not quite enough; 16 bits allow 2^{16} (=65536) possibilities, enough to represesent all possible line numbers, all possible Apple addresses, and Apple integer number values and use the full capability of two Apple words.

The two words must be interpreted as a single two-byte or 16-bit parameter. When this is done one of the two bytes, that containing the more significant bits (the more significant digits) of the number is called the M.S.B (More Significant Byte) and the other the L.S.B (Less Significant Byte). The M.S.B. in the Apple is always assigned the higher memory location of the pair. In computing the decimal value of the total 16-bit number, each bit in the M.S.B. (since it is shifted left by 8 bits) has a value 2^8 (=256) times as great as the corresponding bit in the L.S.B. To make the conversion for a two-byte number it is assumed that each byte can assume values 0-255 (00000000 to 11111111). The largest value that can be represented is thus 255*256+ 255 = 65535 and the conversion formula is

Value of number = value of M.S.B. * 256 + value of L.S.B.

For PEEKing this relationship can be expressed as follows:

Value = PEEK (Address of L.S.B.) + 256 * PEEK (Address of M.S.B.)
= PEEK (Address of L.S.B.) + 256 * PEEK (1 + Address of L.S.B.)

Note that the address of the L.S.B. becomes the address of the two-byte parameter pair. Thus 218 (decimal) becomes the address of a two-byte parameter which contains the line number of the statement where an error occurred if an ONERR GOTO statement has been executed. Reference to the Programmers' Atlas confirms this use of the memory-word pair at addresses 218,219 (decimal).

3.4.2 Example: Finding Line Number of Current Data Statement

Now let's take several examples not documented in the *Applesoft Reference Manual*. Suppose your program READs a number of DATA statements. If it aborts and you want to find out what was the current DATA statement at the time it aborted, the Programmers' Atlas tells you that this information is maintained in memory location 'DATLIN', a two-byte location consisting of locations 123 and 124 (decimal).

PRINT PEEK(123) + 256*PEEK(124)

prints out the desired line number.

3.4.3 Example: Finding Where You Transfer When You Press 'RESET'

Suppose you want your Apple to do something different from what it does when you press the 'RESET' key. The Programmers' Atlas tells you that memory location 65532,65533 (-4, -3) contains an address pointer which tells you where control is transferred on a reset. To see where that is

PRINT PEEK (65532) + 256*PEEK(65533)
or PRINT PEEK (-4) + 256*PEEK(-3)

The two sets of addresses are equivalent for Applesoft. Only the latter may be used in Apple Integer BASIC.

3.5 More About Using Decimal Numbers, Decimal Numbers Modulo 256 and Hexadecimal Numbers to Handle Double-Byte Information

While reading this book you will decide that hexadecimal addresses are easier to use, more convenient, and more meaningful than decimal addresses. Until you do, it probably will make you more comfortable to have many tools and techniques for handling the byte-oriented decimal addresses of the kind BASIC uses in double-PEEKs and double-POKEs.

Surprisingly enough, even if you're not familiar with hexadecimal numbers, some hexadecimal concepts and techniques can be a useful supplement to the computational method we have seen thus far in setting up the kinds of decimal addressing necessary in BASIC.

3.5.1 Memory Pages and the Magic Numbers Decimal 256 or Hexadecimal $100

The Apple's memory is divided into 256 pages of 256 locations each. Thus the Apple memory contains 256*256 = 65536 locations with addresses from 0 to 256*256 – 1 (0 to 65535). Expressing the same information in hexadecimal we can say that the Apple contains $100 pages of $100 locations each, or a total of $10000 locations with addresses from $0 to $100*$100 – 1 ($0000-$FFFF). Thus *all* Apple addresses are exactly four hexadecimal digits long (if shorter addresses are padded out with leading zeros). The first two hexadecimal digits are the hexadecimal page number and the last two digits are the hexadecimal location within the page.

One memory location can hold only a single byte of information. One byte of memory can be thought of as space enough to contain eight binary digits, each with only two possible values, 0 or 1. It can also be thought of as two hexadecimal digits each with 16 possible values (0,1, 2,3,4,5,6,7,8,9,A,B,C,D,E, or F). Or a byte can be thought of as a total entity with 256 (or hexadecimal $100) possible values. The decimal values are 0 through 255; the hexadecimal by two hexadecimal digits, $00-$FF.

When you PEEK, POKE, or CALL using addresses or numbers outside the range of 0-255, one byte will not hold all the information. Two bytes will hold 256*256 = 65536 possible combinations. In hexadecimal this is $100*$100 = $10000 combinations. In any case there are enough combinations so that a different combination is available to address or uniquely identify each location in the Apple's memory.

If not used for addresses, bytes can be used for numeric data, alphabetic text, or even computer instructions. Two bytes are needed to hold integer numbers in the range 0 to 65535 (or signed numbers or addresses in the range –32768 to +32767) so the Apple format for integer numbers and addresses is the same: a two-byte module. Whether used for address, for data, or even for computer instructions, the contents of two bytes can always be represented by exactly sixteen bits or four hexadecimal digits.

One of the pair of bytes contains the more significant part (high-order digits) of a number or address, the other the less significant part (low-order digits) of a number or address.

3.5.2 A Decimal to Double-Decimal Conversion Procedure Using Hexadecimal Tables

The conversion of a two-byte-long decimal number into a pair of single-byte-long decimal numbers suitable for use in double-PEEKing or double-POKEing can be done by computation (as we did it in Section 3.4 or by table look-up).

As indicated in Section 3.4, an address pointer is stored in memory LSB first, then MSB. The 'MSB' or 'More Significant Byte' identifies the page of memory on which the address resides. Its value is the integer number of times that 256 (or $100) will go into the decimal address. The 'LSB' or 'Less Significant Byte' contains the location on the page. It is the remainder left over after integer division by 256 (or $100), that is the number 'modulo 256' (or 'modulo $100').

Even if you know nothing about hexadecimal numbers you can do an integer division by $100. You do it the same way you do a decimal division by 100. To get the quotient just drop the last two digits.

Addresses and integer numbers in the Apple are always four hexadecimal digits, or can be padded with leading zeros, if necessary, to put them into four-digit format. Then division by $100 involves nothing more than keeping the two more significant hexadecimal digits of a four-digit address or number.

The remainder (also called the value of the number modulo $100) is nothing more nor less than the last two hex digits — the ones that you drop off in the division process!

Thus, one way to do decimal MSB and LSB computations is to convert the decimal number

to hexadecimal, divide by $100, getting the hexadecimal MSB and LSB by inspection, then convert the MSB and LSB back to decimal. More specifically:

1. Convert the number (0-65535 or −32768 to +32767) to a four-hexadecimal-digit number — a task quickly and easily done with a conversion table.

2. Do integer division by $100 getting a quotient and a remainder. This process can be done by inspection: the first two hex digits are the quotient; the last two, the remainder.

3. Return to the table and look up the decimal equivalents of *each* of the two two-hex-digit (one byte) numbers obtained in step 2. The two decimal number results are the two single-byte decimal numbers required for double-PEEKing or double-POKEing. The one derived from the high-order digits or page information is the MSB; it goes to the higher of the pair of locations which are PEEKed or POKEd.

This method may seem like the long way around, but if the conversions can be made easily enough, it could be convenient and quite easy.

Actually this procedure can be significantly improved by a carefully planned package of shortcuts: 1. perform only a partial decimal-to-hexadecimal conversion for page information, but not intra-page information; 2. do integer division without remainder to get hexadecimal page information; 3. convert only page information back from hexadecimal to decimal; and 4. get intra-page information by decimal subtraction. Such a procedure is outlined in Section 3.5.4.

First, however, it is helpful to see the unembellished decimal-to-hexadecimal and hexadecimal-to-decimal table look-up procedures which are components of the decimal-to-byte-oriented-double-decimal table-look-up procedure.

3.5.3 Table Look-Up Procedures: Hexadecimal-to-Decimal and Decimal-to-Hexadecimal

To look up the decimal equivalent of a hexadecimal number in conversion tables is absurdly simple. Just follow the procedure in figure 3.5A.

Figure 3.5A
Converting Hexadecimal to Decimal by Table Look-Up

1. Given the hexadecimal number $WXYZ:
   The first two digits WX specify the memory page.
   The last two digits YZ specify the location within the page.

2. Use Figure 3.5C and hex digits WX to look up the start-of-page address.
   Multiply the address by 256.

3. Use the table and hex digits YZ to look up the inside-the-page offset.

4. Add the two together to get the decimal address.

The same tables can be used backwards to look up the hexadecimal equivalent of a decimal number. Just use the procedure specified in figure 3.5B.

Figure 3.5B
Converting Decimal to Hexadecimal by Table Look-Up

1. Divide the decimal number by 256 and look up the integer result in the table (Figure 3.5C). This result is the high byte.

2. Multiply the integer result times 256 and subtract this from the starting decimal number.

3. Enter the table (Figure 3.5C) with the decimal number computed in (2) and look up its hexadecimal equivalent. The result is the low byte.

4. Add the two hexadecimal numbers by inspection. (Note: the first is of the form $HH00; the second of the form $LL; so their sum is of the form $HHLL.

Figure 3.5C
Look-up Table for Hexadecimal-Decimal Conversions

Most Significant Byte	Least Significant Byte 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

3.6 *Hexadecimal Addressing for PEEKs, POKEs, and CALLs*

It is almost impossible to read Apple manuals or literature without finding information about where things are in memory specified in hexadecimal form, because the basic organizational pattern of $100 pages of $100 bytes makes hexadecimal form the easy and natural way to express and manipulate and use addresses.

As soon as you start to use hexadecimal addresses you naturally start to think how convenient it would be to be able to use hexadecimal as well as decimal numbers in PEEKs (and later in POKEs and CALLs as well.)

If you don't yet feel a need for such a hexadecimal PEEK/POKE/CALL capability don't worry about hexadecimal numbers now. Just make a mental note to come back to this area of the text if you need to learn the techniques later.

3.6.1 Hexadecimal PEEKing

If you try to PEEK (or POKE) using a hexadecimal address, *it won't work!* Neither PEEK ($WXYZ) nor PEEK($WX) : PEEK($YZ) will work, where W,X,Y, and Z are each an individual hexadecimal digit.

Despair not! You can still solve the problem of getting the decimal values you need to use in BASIC programs. Though it may not be quite so direct as a PEEK with an explicit hexadecimal address, you can get the information you need by computing the address conversion as part of your BASIC program or by doing a hexadecimal-to-decimal conversion when you write the program, perhaps using the table look-up procedure documented in figure 3.5A.

3.6.2 The All-Hexadecimal PRINTed PEEK Capability of the System Monitor

The system monitor allows you to examine the contents of any memory location using a hexadecimal argument as input and getting a printed hexadecimal result. The net effect is that of a PRINT PEEK with hexadecimal addresses and hexadecimal outputs. However, the word PEEK is not used anywhere in the process.

The process is very simple when you are at the program-building level; i.e., when the prompt ']' has just been given.

The procedure seems to take a number of words to describe in figure 3.5B. However, it takes only a few keystrokes to implement and is easy and convenient.

Figure 3.6A
Using the System Monitor to Get Contents of a Memory Location

(Hexadecimal address specification - Hexadecimal Results)

(1) To enter the monitor type in `CALL -144'.
(Don't type the ` or ' and be sure the press the return key.)
The prompt `*' will appear.

(2) Type in the hexadecimal address of the memory location you wish to PEEK at.
Do NOT precede the hexadecimal number by a `$'.

(3) When you press the return key its contents will be printed in hexadecimal form.

(4) Repeat as many times as you desire for more memory locations.
You don't have to hit the return key for each location, just when you want to see output.
If you want to see a group of consecutive locations just specify the first and last with a period between, e.g. for hex 300 to 305 type in `300.305'

(5) When you are finished peeking, return to the program-building mode and the `]' prompt by entering `3D0G'
(Be sure to use the number 0, not the letter O).

Getting the printed hexadecimal value of the contents of a memory location is not quite so convenient when you want to print the information from inside a running Applesoft or Integer BASIC program, but the need usually isn't as great either.

If you need hexadecimal output from your BASIC program, just convert the hexadecimal address to be PEEKed into decimal using one of the standard procedures we have described, getting a BASIC-style decimal answer, then use the Quick Decimal-to-Hexadecimal conversion subroutine in the Apple firmware to print the value in the desired hexadecimal format. (Case Study 5.4 explains how to do this.)

As an alternate approach you may use the monitor inside a running BASIC program to get the desired information. Use the techniques as described in figure 3.6A imbedded in a block of Pseudo-BASIC code. Section 8.3 describes the Pseudo-BASIC coding techniques which can let you use the monitor from inside a running BASIC program.

Chapter IV
POKEs Can Make Changes

4.1
What Can A Poke Do?

4.1.1 The BASIC Idea

A POKE lets the programmer 'poke' a particular value into a particular memory location.

The format of a POKE statement is

POKE < to decimal memory location >,
< decimal value >

In the Apple the value to be POKEd must be expressed as a decimal number in the range 0 to 255 and the memory address also specified as a legitimate decimal memory address.

4.1.2 Properties and Hardware Implementation Concepts

The binary equivalent of the value POKEd is left in the hardware A-register for the convenience of persons doing machine interface programming. This is not accidental, for the POKE is the BASIC-language implementation of the machine-language STA (STore Accumulator) instruction in the immediate addressing mode. The only difference is that BASIC does not use binary or hexadecimal numbers and hence has the user present the information to be stored in decimal form.

The computer does not have to use the information POKEd into its memory as a decimal number, even though it is entered as a decimal number. The POKE stores in computer memory eight binary bits that have a pattern which is the binary equivalent of the specified decimal value in the range 0 (00000000) through 255 (11111111). Depending upon where the information is POKEd the computer program in use may treat those bits as anything — even as part of a machine-language computer instruction.

The most common elementary uses of POKEing are to change parameters which are used by the system to control its operations. These parameters may be either parameters directly used by the hardware of the system, or parameters which are used by the firmware of the system — the system monitor or BASIC interpreter. The next two sections will describe samples of these uses. Later we will go on to POKEing machine addresses, data values and machine language instructions.

4.2
What a Single POKE Can Do

4.2.1 POKEing the Hardware
EXAMPLES: Changing Graphics Modes

POKEs can change the contents of memory locations which are used directly by the machine to control its hardware operations. For example, the Apple has four areas of memory, the contents of which directly map onto the display screen under different circumstances: 1. Text and Low-Resolution Graphics Page 1 (locations 1024-2046); 2. Text and Low-Resolution Graphics Page 2 (locations 2048-3071); 3. High-Resolution Graphics Page 1 (locations 8192-16383); and 4. High-Resolution Graphics Page 2 (locations 16384-24575). Appropriate information POKEd into these locations will appear directly upon the screen when the appropriate page is activated. The *Applesoft Programming Reference Manual* identifies the appropriate POKEs to activate each of the display options, for example:

POKE 49232,0 (or POKE -16304,0,0)
switches from text to graphics

POKE 49233,0 (or POKE -16303,0)
switches from graphics to text

POKE 49234,0 (or POKE -16302,0)
causes any graphic display to consist entirely of graphics, with no mixture of TEXT

POKE 49235,0 (or POKE -16301,0)
causes bottom portion of any graphic display to be reserved for TEXT from the bottom four lines of the corresponding TEXT page

POKE 49236,0 (or POKE -16300,0)
causes page 1 (whether it be TEXT, Low-Resolution Graphics or High-Resolution Graphics,0) to be displayed

POKE 49237,0 (or POKE -16299,0)
causes page 2 to be displayed

POKE 49238,0 (or POKE -16298,0)
causes the TEXT/Low-Resolution page (as opposed to a High-Resolution Graphics page) to be displayed

POKE 49239,0 (or POKE -16297,0)
causes a High-Resolution graphics page (as opposed to a Low-Resolution or TEXT page) to be displayed

Each of these location pairs, which may be addressed as part of the Apple's memory, access one side of a hardware flip-flop. When properly POKEd each of the four flip-flops mentioned above changes the hardware operation in the fashion specified; e.g. it changes the area of memory to be displayed or the type of display to be created.

POKEs can also be used to put information into areas of memory which are used for screen display, thus having the effect of printing on a particular spot on the screen if the page onto which the POKE is made is the currently displayed page. POKEs, of course, can also print or draw on pages which are currently not displayed, then the results POKEd instantaneously into view by POKEing one of the flip-flops described above.

Such POKEing is also not limited to locations which are accessable with the current printing restrictions established by the text window. (The window establishes left and right, top and bottom margins within which printouts can be made.)

Depending upon where POKEing is done it can, of course, set-up without current display or set-up and concurrently display either 1. Text, 2. Low-Resolution Graphics, or 3. High-Resolution Graphics. In our detailed example we will describe the Text option, because it requires the least prior knowledge of the Apple's hardware and programming. The graphic modes are closely analogous.

4.2.2 POKEing the Software
EXAMPLES: Changing Printout Speed;
Changing Normal/Inverse/Flashing Mode;
Changing Printout Window;
Changing Cursor Position

You don't need to restrict POKEing to locations in the computer which have direct hardware impact upon system operation. System operation is also dependent upon software imbedded in the system firmware; e.g. the system monitor, BASIC interpreter and disk operating system (DOS). POKEs to key parameter locations used by this firmware can markedly affect what the total system does. In many cases these POKEs can achieve exactly the same results as some of the system-dependent commands in Applesoft or Apple Integer BASIC. In other cases they can perform functions for which specific commands have not been created.

Often firmware parameters you would like to POKE to achieve a given effect are the same parameters you might also like to inquire about using a PEEK. For example, if you found by PEEKing that the printout speed had been artificially reduced (by lowering the speed variable from its default value of 255), you could restore it without use of the SPEED command. To do this would require the same knowledge you required to PEEK at the SPEED in the first place; i.e. remembering (or re-looking-up in the Programmer's Atlas) the fact that speed is controlled by 'SPDBYT' which is located at memory location 241 (decimal). This location contains the two's complement of the SPEED; i.e. 256 — SPEED. It can be reset to maximum speed (minimum delay) by POKEing 256-255 = 1 into it; i.e.

```
POKE 241,1
```

This particular POKE is functionally equivalent to the SPEED statement

```
SPEED = 255
```

Similarly the Programmers' Atlas tells us that the memory location named 'INVFLG' (INVerse video display FLaG) will have value 255 ($FF) for normal video display, 127 ($7F) for flashing display, or 63 ($3F) for inverse (white background) display. Thus

POKE 50, 63 is equivalent to the Applesoft command INVERSE
POKE 50,127 is equivalent to the Applesoft command FLASHING
POKE 50,255 is equivalent to the Applesoft command NORMAL

Where an extension of BASIC is built into Applesoft or Apple Integer BASIC there is usually little advantage in using a POKE rather than the extended-BASIC statement to which it is equivalent. Although Applesoft BASIC includes many extensions such as the key-word commands SPEED, INVERSE, FLASHING, and NORMAL, there are many important and useful functions available in the machine through the use of POKEs for which system-dependent extensions from ANS BASIC (American National Standard BASIC) have not been created. The *Applesoft Programming Reference Manual* contains several pages of POKEs, but many more may be found from a careful perusal of the Programmer's Atlas.

Typical examples of those mentioned in the *Applesoft Programming Reference Manual* are the POKEs used to change the text window (the area within which printing can occur):

POKE 32,	desired left edge of window. Range:0-39
POKE 33,	desired width of window (characters per line)
POKE 34,	desired top margin. Range 0-23
POKE 35,	desired bottom margin. Range top-24

Each of these parameters is used by the system monitor and/or the Applesoft BASIC interpreter to control where printing currently is and is not allowed to occur. (NOTE: The results of these POKEs will not be effective until the next time the monitor uses these locations.) Incidentally,

there is a 'SETWND' (SET WiNDow) subroutine in the Apple monitor. Unfortunately the Programmers' Atlas tells us that it is really a reset window subroutine which sets the window to its standard (default) condition.

There are many other POKEs not explicitly documented in the *Applesoft Programmers' Reference Manual*. One of these:

```
POKE 37,CV
```

will tell the monitor that the cursor should be moved to vertical position CV for the next output operation (regardless of where it might have been earlier). However, there is one important word of warning: if you directly POKE a parameter used by the firmware (as opposed to calling a firmware subroutine which changes it in the standard way the firmware intends that it be changed) you may find occasional strange or pathological results. First, the POKE will not become effective until the firmware next uses the parameter — so there may be a considerable delay before the effect of the POKE becomes evident. Next, the subroutine that changes this parameter may have to change other related parameters at the same time. If you don't change these parameters you might get unexpected results. Finally, the firmware may itself change the parameter to a value that it wants before it uses the parameter the way you wanted it used, so your change may have no effect.

4.2.3 POKEing Alphabetic/Numeric Information EXAMPLE: Changing the Screen Display

In the text mode the Apple can display 24 lines of characters with up to 40 characters on each line. Each character on the screen represents the contents of one memory location from the text page currently being used as the video display buffer and hence being displayed.

An extremely detailed description of the organization of the screen display area used for Text printout may be found in Chapter 14. Just a few of the basic ideas involved will be introduced here.

The area of memory which is used for the primary (default) text page extends from location 1024 to location 2047 (decimal); the secondary text page extends from location 2048 to location 3071. Normally you use and display only text page 1. However you can arrange to pass output to text page 2 (which is not being displayed). Then POKE −16299,0 changes the display instantly from that on page 1 to that written into the memory locations associated with the previously invisible page 2. POKE - 16300,0 changes the display back to show the information stored in memory associated with text page 1.

Since there are only 24 times 40 = 960 display locations but 1024 memory locations in a text page, there are 64 locations left over for other purposes. See Chapter 14 to find out how they are used.

There is a simple formula for assignment of memory locations to visible locations on the Apple output screen. A small bit of experimentation would allow you to derive that formula, but initially it is easier to use the diagrams given in Chapter 14 or the diagram of the text screen on page 16 of the *Apple* (System) *Reference Manual*.

At each memory location the 8 bits in the computer memory provide the ASCII (American Standard Code for Information Interchange) bit-combination which represents the printed symbol at the corresponding location on the screen. The table of ASCII Screen Characters in the *Apple Reference Manual* gives the decimal number which you must POKE into the specified location to get the desired printed symbol.

NOTE: For standard Apples the character set contains 26 upper-case letters, 10 digits and 28 special characters (punctuation, etc.) for a total of 64 characters. The ASCII code is a 7-bit code which includes 2∧7 (=128) assigned characters including lower-case letters and a group on non-printing 'control' characters. The Apple keyboard cannot produce lower-case ASCII letters.

Normally the firmware of the Apple automatically converts lower-case letters received from the outside world into capital letters, but you can bypass this by appropriate use of monitor subroutines documented in the Programmers' Atlas.

You can generate the 'control' characters from the Apple keyboard by pressing the 'CTRL' key and the corresponding letter-key concurrently, but they do not print on the screen. Instead they perform other functions: for example CTRL-G, known in ASCII as 'bell', sounds the attention beeper in the Apple.

Hardware adapters (e.g. the Don Paymar adaptor) can be installed in an Apple to give it the capabilities for handling the full ASCII character set.

Since there are 7 bits in an ASCII character and 8 bits in a byte (the amount of information which can be held in a memory location), there is an extra bit available. This is normally set to a one for printing characters in the Apple, so the Apple codes used for POKEing text characters are normally those with decimal code values 128-255. When this bit is zero, the ASCII code is mapped into either an inverse or flashing character.

Let's POKE some examples:

```
POKE 1024,163 : POKE 1063,163 :
POKE 2000,163 : POKE 2039,163
```

will replace whatever character is on the screen of the Apple at each of its four corners by a '#' symbol.

```
FOR I = 1448 TO 1487 : POKE I,170 : NEXT I
```

will draw a horizontal line of asterisks across the middle of the screen.

```
FOR I = 1024 TO 2000 STEP 128 : FOR J = 0
TO 5
POKE I + J,160
NEXT J : NEXT I
```

will put blanks into the leftmost six columns of the printout screen.

NOTE: Since 1024 is the top left corner, 2000 is the bottom left corner, and the increment between lines is 128, you might easily be forgiven for assuming that this program segment would clear the entire left column. It doesn't — just the top one-third. Why? The screen is interlaced. Locations 1024-1063 represent the top (or zero-th) line; locations 1064-1103 represent the eighth line of the screen; locations 1104-1143 represent the sixteenth line of the screen. Then there is a gap of 8 locations — 1144-1151. The contents of these locations are not displayed. As indicated in the Programmers' Atlas they are used as 'scratch-pad memory bytes,' each reserved for use by the peripheral device associated with a given 'slot' into which peripheral cards can be plugged in the Apple hardware. Then at 1152 the cycle repeats; and again at 1280; and again until the whole screen is covered with the last line beginning at location 2000. However, the only locations POKEd by the FOR...FOR...POKE are in lines 0-7. All locations in the bottom two two-thirds of the screen were skipped over by the increment of 128. To get the expected result you would have to replace the middle line of the program with

```
POKE I + J,160 : POKE I + J + 40,160 :
POKE I + J + 80,160
```

(The top limit of I could also be reduced to 1920 for clarity in understanding the program, but it makes no difference in execution.)

4.3
Sometimes You Should Double-POKE

If you want to POKE an integer number larger than 255, or a memory address in any area of memory other than page zero, there is not enough room for your information in a single byte so you must double-POKE.

4.3.1 Double-POKEing Without Aids

This process is exactly the other side of the coin from double-PEEKing. Just take the number which is too large and decompose it into parts that are individually in the range 0-255. For numbers up to 256*256 − 1 (= 65535), this can be done with two numbers in the range 0-255 and hence with two bytes and two POKEs, i.e., a double-POKE. First let's look at the theory.

Two words or bytes of memory can contain 2 * 8 or 16 bits of information; i.e. 256 different bit combinations. These can be used to represent the unsigned decimal numbers 0-65535 or they can be used to represent signed decimal numbers. The Apple system uses the twos-complement method for representing negative numbers. With this method of representation the 16 bits can represent the numbers −32768 to +32767. The conversion process is identical, but it is easier to follow if you think in terms of the unsigned numbers. To convert between them merely add 65536 to any negative number to get its unsigned equivalent which uses exactly the same bit pattern. To find what must be POKEd into each byte of a two-byte pair, you must do an integer division of the number by 256 in order to get the integer to be POKEd nto the M.S.B. (the More Significant Byte of the two-byte pair). The remainder created in the integer division process is POKEd into the L.S.B. (the Less Significant Bit of the two-byte pair). If no remainder is created, the remainder is zero and zero is POKEd into the L.S.B.

NOTE: An integer division is the kind you learned to do in elementary school before you learned about decimal fractions. The result of an integer division is always an integer (whole) number often with a second integer number (the remainder). This is different from the division done conventionally with pencil and paper after elementary school and the most common type of division done on desk calculators and computers. There is no separate remainder; the remainder is converted to a decimal fraction tagged onto the result (the quotient).

The required computations are obvious in Integer BASIC. Division gives an integer result. This goes to the M.S.B. The remainder (if any) is provided by the remainder of modulo function. It goes to the L.S.B.

Applesoft, however, does conventional divisions which often give decimal fraction results. Fortunately, whenever you try to stuff a number with a fractional part into a location which will accept only integers, any fractional part will automatically be chopped off. This what you

POKE will automatically be correct if you merely POKE the number/256. Nevertheless, it is good practice to make the integerization explicit by POKEing INT(Number/256).

4.3.2 Other Ways of Setting Up a Double-POKE

Several utilities have been written to permit you to double-POKE by using a statement such as:

&,< dec loc> , < address or number > or
CALL 768, < dec loc> , < address or number >

where < address or number > is not limited to single-byte size (0-255), but may be double-byte size (– 32768 to + 32767 or 0 to 65535). 768 is assumed to be the location of the utility. (It is a frequently used location for small user-written programs and utilities used with BASIC. Why? If you are interested, see Section 13.3.)

Except in exceptional circumstances, consider such utilities to be examples of bringing in a pile-driver to do a tack-hammer's job, but you may find them useful. BASIC code, which if inserted early in the main program will thereafter allow you to use the 'CALL 768' in the fashion indicated above, is shown in figure 4.3A. An illustration to implement the '&' is given in figure 4.3B.

Figure 4.3A
Applesoft Utility to Allow Modified CALL Statement to Perform a Double-POKE

```
10  REM  Put this code near start of your main program.The machine-language
         subroutine it loads to memory occupies $300-325 (decimal 768- 805
         With it you can double-POKE at any place in your program by using
         a CALL 768,<Low POKE address>,<Value to be POKEd (-32768 to +32768
         or 0 to 65535)>
20  A$ = "300:20 BE DE 20 67 DD 20 52 E7 A5 50 85 3C A5 51 85 3D 20 BE DE
          20 67 DD 20 52 E7 A0 00 A5 50 91 3C C8 A5 51 91 3C 60 N D823G"
30 FOR I=1 TO LEN(A$):POKE 511+I, ASC(MID$(A$,I,1))+128:NEXT:POKE 72,0:CALL
         -144
```

Figure 4.3B
Applesoft Utility to Allow a BASIC Statement Using an '&' Token to Perform a Double-POKE

```
10  REM  Put this code near start of your main program. The machine-language
         subroutine it loads to memory occupies $300-325 (decimal 768-805).
         With it you can double-POKE at any place in your program by using a
         &,<Low POKE address>,<Value to be POKEd (-32768 to +32768 or
         0 to 65535)>
15  POKE 1013,76:POKE 1014,0:POKE 1015,3: REM  Usually POKE 1013,76 can be
         omitted
20  A$ = "300:20 BE DE 20 67 DD 20 52 E7 A5 50 85 3C A5 51 85 3D 20 BE DE 20
          20 67 DD 20 52 E7 A0 00 A5 50 91 3C C8 A5 51 91 3C 60 N D823G"
30 FOR I=1 TO LEN(A$):POKE 511+I, ASC(MID$(A$,I,1))+128:NEXT:POKE 72,0:CALL
         -144
```

The analysis of these programs is quite involved, but well within the skills you should be able to acquire by the end of this book.

You might even consider the analysis of this code an appropriate 'final exam' on the techniques of this book. If you accept the challenge of undertaking the analysis, it is helpful to know the following facts:

1. The BASIC code hides a machine-language routine that uses software tools from within the Applesoft interpreter to seize control of the analysis of the pseudo-BASIC statements. It then analyzes them, implements them, and returns control to the next line of Applesoft code.

2. The machine-language code is converted to pseudo-BASIC by the Lam technique explained in Section 8.3.

3. The machine-language code uses as software tools the CHKCOM routine at $DEBE, the FRMNUM routine at $DD67, the GETADR routine at $E752, and zero-page memory location LINNUM at $0050.

4. Background knowledge treated in Chapters 6, 7, and 19 is important and the routine uses programming techniques similar to those described and partially analyzed in Sections 5.10 and 5.11.

Chapter V
CALLs Can Make Things Happen

5.1
CALL — A 'GOSUB-like' Statement Providing Direct Program-Control Access to Hardware Memory

5.1.1 The BASIC Idea

CALL is used in both Applesoft BASIC and Apple Integer BASIC to provide a subroutine-type transfer of control to a machine-language subroutine. A CALL that specifies a particular hardware memory location performs the same function for machine-language subroutines that a GOSUB that specifies a particular BASIC line number does for subroutines written in BASIC.

CALL, like the GOSUB, normally operates with automatic return at the end of the subroutine. After the called subroutine has been executed, control is passed back to the next statement after the CALL statement.

5.1.2 Formal Description of the CALL Statement

The CALL statement has the format

CALL < decimal-number location of machine-language subprogram >

NOTE: Although this use of CALL is common in microcomputer BASICs, it is not universal. Some microcomputer BASICs require that the location be specified in hexadecimal rather than decimal form.

Even more confusing to a microcomputer programmer may be the use of CALL in some maxicomputer versions of BASIC such as Dartmouth BASIC6, Dartmouth BASIC7, and Dartmouth SBASIC. These more sophisticated versions of BASIC translate BASIC statements to machine language *via* a compiler before beginning to run the program. This is in marked contrast to most microcomputer BASICs, which interpret them one-at-a-time as they are executed.

In this environment, the CALL statement is used to allow specially written BASIC subroutines to be treated as separate programs that may use the same variable names for entirely different variables. The main program and these subroutines have no intercommunication other than that set up by the subroutine definition statements and the CALL statement. Thus a typical Dartmouth BASIC6 CALL statement might be

```
CALL "SOLVEIT":A,B,C,X
```

with 'SOLVEIT' being the name of a BASIC subroutine defined outside the limits of the main BASIC program and A,B,C, and X being variables passed to it.

In both the Applesoft and Apple Integer versions of BASIC, as well as in most other microcomputer versions of BASIC, a CALL to a particular memory location is a BASIC language statement that generates a machine-language JSR (Jump to SubRoutine) to the specified location. (NOTE: BASIC uses only decimal numbers while machine-language instructions use addresses in binary/hexadecimal form.) Thus the BASIC statement uses a decimal address which is automatically converted to binary/hexadecimal form in the BASIC interpreter.)

It is up to the programmer to make sure that any pre-conditioning of hardware registers and/or memory locations required for proper operation of the called subroutine has been done in advance of issuing the CALL statement.

It is also up to the programmer to make sure that the subroutine execution ends with a RTS (Return Transfer from Subroutine) statement.

When you forget to use an RTS, it is the equivalent of forgetting a RETURN statement in BASIC, but machine language is not as gentle to the programmer when errors occur as is the BASIC interpreter. Instead of detecting a problem and printing out an error indication, the system may destroy anything in memory at the time.

Under certain circumstances the matter of transferring control to and from a machine-language subroutine can get quite complex. You'll have to learn about the 'stack processing' in the 6502 and the parameter passing conventions adopted by the monitor and Applesoft. This topic is covered in depth in Chapter 11.

5.1.3 Subroutine Transfer of Control Diagram (Program Mixing GOSUBs, CALLs and JSRs)

Meanwhile let's telegraph some of the basic ideas which link the handling of subroutines in BASIC with those in machine language by considering a subroutine transfer-of-control diagram which shows the essential similarities of the various BASIC and machine-language subroutine calling procedures.

Figure 5.1.3A shows the flow of control on transfers of control to subroutines

(1) GOSUB Statements
(From BASIC to BASIC subroutine)
(2) CALL Statements
(From BASIC to machine language subroutine)
(3) JSR (Jump to Subroutine) instructions
(From machine language to machine-language subroutine)

to return control to the calling program you use

(1) RETURN Statements
(To return from a BASIC subroutine)
(2) RTS Instructions
(To return from a machine-language subroutine)

5.2 Use of the CALL Statement Example: CALLing System Subroutines

When an Apple is operating in Applesoft or Apple Integer BASIC there are at all times sitting inside the Apple a large number of machine-language subroutines used by the monitor and/or BASIC interpreter. These subroutines are all potentially available to the user *via* CALL statements.

The capabilities of these subroutines range from trivial to extremely powerful and useful. There is a particularly large selection of input/output capabilities and options.

One of the major contributions of the Programmers' Atlas to the typical programmer is to make these subroutines readily available for exploitation.

Some of the subroutines which you may find in the Programmers' Atlas perform trivial tasks. Samples include the Applesoft subroutine 'OUTQST' at memory location 56154 (decimal) which prints out a question-mark, and System monitor subroutine 'CR' at memory location 64610 (decimal) which outputs a carriage return.

Some of the subroutines which you may find in the Programmers' Atlas perform very useful tasks which might be entered more conveniently from the equivalent Applesoft machine-dependent extensions of BASIC. Examples include subroutine 'TABV' at location 64347 (−1189) and 'HLINE' at location 63513 (−2023).

Yet other subroutines perform functions which are useful and required by the system, but do not seem to be accessible from BASIC *via* the Applesoft or Apple Integer BASIC machine-dependent extensions of BASIC. These fall into two major classes: 1. functions which are documented as capabilities of the monitor; e.g. MOVE, and 2. functions which are hidden inside either the monitor or interpreter because they are needed in the operation of the system but are not well documented. Often the subroutines perform their functions very rapidly and with greater flexibility than you could with a BASIC program, even if you knew how to write a BASIC program to do the task.

Two useful subroutines are 'CLREOP', at memory location 64578 (decimal), and 'CHRGOT' at memory location 183 (decimal). The former clears the Apple screen from the current cursor position to the end of the page. The latter is a hidden point in 'CHRGET', a very powerful input routine which Applesoft calls when it wants another character. 'CHRGOT' differs from the better-known routine of which it is a part because it does not change the pointer which identifies where to get the next character.

Finally, the always-available software in the Apple II firmware packages provide a wide variety of practical and useful services for programmers who decide that considerations like run-time efficiency make it worthwhile to write their own assembly-language or machine-language coding. These service routines load and save registers, convert data from one format to another and perform almost any function BASIC performs from within a machine-language program.

In summary, there are an amazing variety of useful subroutines hidden inside both the system monitor and the Applesoft interpreter. Many can be used without the slightest knowledge of machine language. Others require only minimal knowledge of machine language.

Many of the simpler ones do not depend upon setting up any linkages in advance of CALLing them. Others require some set-up, but usually no more than can be accomplished by a few POKEs. In the case of some of the more sophisticated machine-language subroutines some (or all) of those POKEs may most conveniently be synthesized from two or three (or more) machine-language instructions.

5.3
Passing Parameters: Communication Between BASIC and Machine-Language Routines

Traditionally the term 'parameter' has meant 'A constant or variable term in a mathematical function that determines the specific form of the function but not its general nature.' When dealing with computers this idea is extended from functions to the more general cases of computer programs, procedures, and subroutines.

Subroutines often need information or data from the calling program in order to determine specifically how to execute the general procedure specified by the subroutines. The process of getting that information to the subroutine is called 'passing parameters' to the subroutine.

The process is equally important whether the subroutines are written in BASIC or in machine-language, but it is one that you normally don't have to pay much attention to when you are programming in BASIC only. In BASIC (as implemented by interpreters such as Applesoft and Integer BASIC) all variables are treated as part of a common pool which is equally accessable from both inside and outside the subroutine. Thus parameters are automatically passed into programs just by using a variable inside a subroutine which has previously been given a value outside the subroutine. Similarly parameters can be passed out of subroutines just by using a varaible outside the subroutine that was evaluated inside the subroutine.

When you move from BASIC to a machine-language subroutine, such as those available in monitor firmware, that machine-language program does NOT use BASIC names. Thus there is no automatic communication through the use of common names.

If you want information to flow between BASIC and machine-language programs, creating the channel of communications is your responsibility. You must take some kind of action to set up a communications link — some method of telling the machine-language routine what information from outside itself is relevant inside and vice versa.

We will take up this communications problem in considerable detail using a number of case studies. First, however, we'll take up a simpler problem: passing parameters into a subroutine — in this case subroutines in monitor firmware — from the keyboard.

5.4
Passing Parameters to Monitor Firmware

5.4.1 Passing Parameters from the Keyboard

The Apple Monitor not only provides service and utility routines to BASIC and the Disk Operating System (DOS), but it is designed to permit the user with convenient direct access to the computer via the keyboard.

For example, from the keyboard you can examine the contents of any memory location or group of locations, change the contents of any location or group of locations, move the contents of any block of memory to another location (i.e. copy it) etc.

One word of warning: The monitor puts you as close to the hardware of the machine as you are likely ever to get. At the hardware level the computer is a binary device, so the monitor deals with memory addresses in BINARY form, or to be more precise, in the more human-conventional HEXADECIMAL abbreviations of binary form. Thus all addresses and values which you may give to or get from the monitor will be in hexadecimal form.

Don't let the fact that a 'number' which you give to or get from the monitor may contain the letters A-F as well as the digits 0-9 shake you up. (The counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, ...). Just accept it and don't let it bother you. We'll get around to all the theory of binary/hexadecimal numbers you will need to work at the machine-level later when you need the information to understand the inner workings and hidden mechanisms of the computer. If this approach bothers you and you want to understand more about binary/hexadecimal numbers now, jump ahead and read Section 7.1 or as much of Chapters 6 and 7 you feel is necessary to make you feel comfortable.

Figure 20.4A gives a summary of the monitor commands accessable from the computer's keyboard. At the moment we are more interested in the form and method of communications used than in the commands themselves.

The keyboard entry features of the monitor are designed to accept and decode simple instructions which specify what monitor operation is to be performed and what data is to be used. Some require no parameters, e.g.

Command	Meaning
I	Set Inverse display mode.

Others require one parameter, e.g.

Command	Meaning
(adrs)	Display the contents of one memory location, the address of which is (adrs).

If, as in this case, no explicit instructions are given what to do with the memory location, the monitor examines it and prints out the contents of the memory location specified. However if the command had been (adrs)G it would GOTO, i.e. transfer control to the address specified. If it had been (adrs)L it would LISt; if it had been (adrs)T it would TRACE, etc.

Other monitor activites require two parameters, e.g.

Command	Meaning
(adrs1).(adrs2)	Display the contents of each location in the range starting at (adrs1) and ending at (adrs2)

The monitor keyboard routine is designed to put parameters which are keyed in at the keyboard into a register and/or a special memory location named A1 if a single parameter is entered. If two are entered (adrs1) goes to locations A1 and (adrs2) to A2 if two are entered.

In this and other monitor commands a period between adresses [(adrs1).(adrs2)] specifies that they act as the start and end of a block of memory.

After the parameters are in place the monitor looks for a type-of-action designator. If, as in this case, no other explicit designation is present to specify what to do, the monitor transfers control to a block of monitor code which examines and prints out the range of memory specified by parameters A1 and A2.

However had the command had a type-of-action designator, e.g. (adrs1).(adrs2)W then it would transfer control to a block of monitor code which would WRITE the block of memory locations onto tape. If the command had been (adrs1).(adrs2)R then the monitor would transfer control to a different block of monitor code which would READ enough data from tape to fill that memory from the address specified by the address parameter in A1 to that specified in A2.

There are also monitor commands which specify three parameters. For example

Command	Meaning
(dest)<(start).(end)M	Copies the values in the range (start).(end) into a destination starting at (dest).

As before (start) [also known as (adrs1)] goes to A1; (end) [also known as (adrs2)] goes to A2. (dest) [also known as (adrs4)] goes to A4. The type-of-action designator M causes the monitor to transfer control to a block of code which MOVEs the block of memory specified by the address/parameters in A1 and A2 to the location specified the address/parameter in A4. (Had the designator been V it would have transferred to a block of monitor code which would VERIFY whether or not the block starting at A4 contained the same information as the block A1 to A2.)

5.4.2 Passing Monitor Parameters and Calling Monitor Routines from Inside a BASIC Program

In Case Study 5.4 we will use the firmware routine the monitor uses to implement the MOVE described above as part of a BASIC program. We can use this monitor routine to move any desired block of information inside the computer more quickly and more conveniently than if we tried to write out all the BASIC code to do the same operation.

To do so we must learn how to pass parameters (A1, A2, and A4) and call the monitor firmware from a running BASIC program.

The procedure is simple in principle:
(1) POKE the correct start, end and destination into A1, A2, and A4. (If you don't know where A1, A2, and A4 are in memory look them up in the Gazeteer section of this book.)

(2)CALL the monitor MOVE routine at its machine-language address. (This address may also be found in the Gazeteer section of this book.)

The need for this kind of capability and procedure is not restricted to routines built-into the system firmware, but is equally applicable to routines which the user him/herself may write or obtain from other sources.

5.4.3 In-Depth Case Study: Calling a Subroutine with Parameters in Monitor-Specified Memory Locations.

```
1   REM  ******************************
2   REM  *       CASE STUDY NO. 5.4     *
3   REM  *   WHAT'S WHERE IN THE APPLE  *
5   REM  *       QUICK DECIMAL TO       *
6   REM  *    HEXADECIMAL CONVERSION    *
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. Lines 10-15 clear the screen, tab part way down, and ask for and accept as input a decimal number. They then reclear the screen and print 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 lines 70-80 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 90 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 monitor subroutine subroutine at -589 was not designed to de exactly what was wanted. It printed unwanted output which had to be concealed by overprinting with "HEX=".

Finally the POKEd output onto the screen lines 70 and 80 might better be replaced by a BASIC 'PRINT "HEX=" ' 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?

5.5 *A More General View of Passing Parameters Between BASIC and Machine-Language*

If many of the routines in the monitor need parameters to tell them exactly what to do, and we are going to use these machine-language routines in our BASIC programs, then we must learn how to pass parameters to them within a running BASIC program.

There are many, many options for passing parameters between programs. Sophisticated programmers may or may not recognize such differing concepts as passing parameters by name or by value; passing via memory, via registers or via a stack; passing directly or passing indirectly via a pointer. Such sophistication is unwarranted here, since the intended audience includes many who know nothing of the machine-language programming techniques which are essential background to a meaningful discussion of many of the potential options. Thus we will take a very simple-minded approach.

5.5.1 What is Passed?

There are two basic ways of passing information into (or from) a machine-language program. You can pass the data itself or pass a pointer — an address which points to the location where the data may be found.

Whichever way you use the BASIC program and the machine-language program must be written to the same interpretation. The machine-language of the microprocessor in the Apple has both direct and indirect methods of addressing. This permits either method of specification to be used with comparative ease. (The relevant theory is described in Chapter 7.)

Internally the larger, more sophisticated and important data handling routines in the system firmware tend to specify information by address pointers.

For example, the monitor routines in Case Study 5.4 had you pass in addresses which specified the memory locations of the data to be moved into A1, A2, and A4. In this case, of course, the addresses may be thought of as the 'data' subroutine needs, but it is important to keep your eyes open for situations where the machine-language programmer's choice is not so clear-cut and obvious. Be careful! It is possible — and sometimes very easy — to confuse the address where data is located and the actual data values in that location.

5.5.2 How It Is Passed

We will consider four basic options:

1. Passing parameters via addressable memory locations

i.e. BASIC and the machine-language program agree to use the same memory location for the same information. The BASIC program puts the information there before CALLing the machine-language subroutine or the machine-language subroutine puts the information there before returning control to BASIC.

2. Passing parameters via hardware registers (Not Addressable in the Apple's microprocessor)

i.e. the BASIC and machine-language progams agree to exchange information in special hardware locations known as 'registers which are not addressable (and hence not directly accessible by PEEKs and POKEs), but are accessable by means of machine-language instructions and firmware subroutines.

3. Passing parameters via the system stack

This is a method we will touch on only lightly in this book. (We will use the technique in a program in Section 5.10) This technique is of special importance to interrupt processing, recursion and re-entrant coding.

4. Miscellaneous methods of passing parameters

Two miscellaneous methods we will touch upon lightly are (1) communications via the '&' token in Applesoft and the related '&-vector' (actually a JUMP instruction) in the monitor-vector page, and (2) communications via the USR() function, a function which combines a CALL capability with the ability to pass a single real (floating-point) variable.

We will cover the first two options in depth with explanations and case studies.

5.6 *Passing Parameters Via Preagreed Memory Locations*

5.6.1 Overview

An important method of achieving communication between a BASIC program and a machine-language program is preagreement upon the use of particular memory locations by the machine-language program. Then BASIC can put things into them whenever required by POKEing then they will be available to the machine-language program whenever it is CALLed.For communication back from the machine-language program all BASIC need do is PEEK at the contents of the locations after control is returned to it.

This was the technique actually used in Case Study 5.4. The BASIC program and the monitor agreed upon the locations to be used for parameters A1, A2, and A4.

If you determine from the Atlas or Gazetteer or some other source that a particular firmware routine uses specific memory locations for parameters that control its operation, you can just POKE the required values into them before CALLing.

If you determine that the firmware routine leaves desired results in particular memory locations, you can just PEEK at those locations to access the results.

5.6.2 Some Areas Which Contain Many Standard Common-Agreement Locations

Much of page zero of memory (memory addresses $0000-$00FF or decimal 0-255) is occupied by parameters used by the Apple system firmware; i.e., the monitor, Applesoft, and Integer BASIC interpreters and the Disk Operating System.

There is another important group of locations associated with hardware input/output operations and the peripheral 'slots' on memory page 192 (see Chapter 18 for details).

In earlier chapters we have already PEEKed and POKEd at some of the locations in these two memory pages. However, there are many, many more, some of which give you access at the most intimate level to the inner workings and hidden mechanisms of the Apple hardware-software system.

Most of the memory locations in page zero are permanently assigned to particular functions. However, many of these functions are of sufficient generality that they can be used as a useful medium of communications as well. These include the following:

1. Monitor general usage subroutine parameters A1 through A4.
2. The integer number (16-bit) pseudo-registers. (These include the Applesoft 16-bit pseudo-accumulator AC plus its extension XTND and its auxilliary AUX.
3. The Applesoft Floating Point Accumulator (FAC).
4. The 'Sweet-16' pseudo-registers R0 through R15. (Sweet-16 is a 16-bit pseudo-computer available to Apple users *via* an interpreter built into the non-autostart version of the Apple system monitor.)
5. The general-usage low-resolution graphics parameters identified in Chapter 14.
6. The general-usage high-resolution graphics parameters identified in Chapter 16.
7. Some of the key internal information-handling parameter locations in the Applesoft Interpreter such as LINNUM, TXTPTR, etc. Careful use of such locations (perhaps together with some of the Applesoft internal information-handling routines) can make the current or next line of Applesoft code or Applesoft variables or strings available not just to BASIC, but to machine-language routines as well. (Such techniques can be extremely powerful. They are, for example, used in the program of Figure 5.10A.)

Since all these locations mentioned are in addressable memory, they are directly accessable by means of PEEKs and POKEs. BASIC programs can set up parameters in them by means of POKEs before a machine-language subroutine is called and the values of parameters put into them by machine-language programs can be obtained by PEEKs in BASIC programs after control is returned to BASIC. The techniques involved were illustrated in Case Study 5.4 (Section 5.4.3).

5.6.3 The Simulated Registers In Addressable Memory

In addition to the built-in hardware registers, the Apple system has a number of simulated registers which Applesoft or the system monitor create in addressable memory. Many of the truly general-use parameter-passing locations are actually doing double-duty as simulated registers. That means that they are used not just for communications but as locations where significant processing occurs as well.

Whereas hardware registers have machine-language instructions which can be used to load from memory and unload or store information in them back into memory, simulated registers do not have built-in hardware instructions available to load or unload them. However, they normally have firmware routines which can be used for the same purpose, and these routines can be found in the Atlas and Gazetteer sections of this text. Since these simulated registers exist in addressable memory they can also be accessed by means of PEEKs and POKEs.

The special information-handling routines for the simulated registers can be particularly valuable when you are passing data (as opposed to the addresses of data) and dealing with real variables (floating point numbers). The format of real number data is messy enough to convert to and from to make direct POKEing and/or PEEKing a royal pain. When dealing with pseudo-registers and parameters which must be in the real-number (floating point) format, use the register-handling routines (which may be found in the Atlas and Gazetteer parts of this book) in preference to direct POKEing. NOTE: When using real (floating-point) parameters don't neglect the possible convenience of using the modified CALL of Section 5.10 or the Applesoft built-in function USR(P) of Section 5.11.2.

5.7
Discussion of Passing Parameters Via *Hardware Registers*

5.7.1 Hardware Registers in the Apple

Registers are very special hardware locations in the central processing unit of the computer where information can be both processed and stored.

The Apple has five hardware registers of special importance to machine-language programs. They are:

1. The A-register or Accumulator
2. The X-register or X index-register
3. The Y-register or Y index-register
4. The P-register or status register, and
5. The S-register or stack pointer

The special properties of these registers will become obvious in Chapters 6 and 7.

Registers are so much at the heart of machine-language programming that it is natural to want to communicate with and exchange parameters between BASIC and machine-language programs using information in these registers.

In some computers these registers are addressable as memory locations. This would permit you to access them directly by means of PEEKs and POKEs. In the Apple's MOS6502 central processor they are not addressable and you cannot directly access them in this way. In Section 5.7.2 I will discuss several approaches you can use to access them.

5.7.2 Major Options for Passing Parameters To and From Hardware Registers

In the Apple we have three major ways for a BASIC programmer to get information to or from machine-language code *via* registers:

1. The Direct Approach
   Use the hardware instructions designed — and built — into the Apple to load the registers from memory or to unload the contents of registers to memory. Those who suffer from machine-language phobia will not like this approach, but it is quite simple — even for non-programmers in machine language.
2. The Monitor SAVE/RESTORE Approach
   Use monitor subroutines which are designed to SAVE the contents of key registers to memory and to RESTORE the registers from memory to load or unload the registers.
3. The Modified CALL Approach
   Use a special machine-language program which, in effect, modifies the Applesoft CALL statement so that it accepts modified CALL statements which include within the CALL itself the parameters to be loaded to the hardware registers.

5.8
The Monitor SAVE/RESTORE Approach

5.8.1 The Concept

We consider the Monitor SAVE/RESTORE approach first because it requires *no use of machine language*. It takes advantage of the fact that the system itself must occasionally save and restore the status of its registers and that it has firmware subroutines which allow this to be done in addressable memory.

The steps involved are simple:

1. CALL -182
   This calls the monitor SAVE routine at memory location $FF4A. This routine saves the current contents of the A-register in memory location $45 (decimal 69), the X-register in $46 (decimal 70), the Y-register in

$47 (decimal 71), the P-register in $48 (decimal 72), and the S-register in $49 (decimal 73).

2. POKE Memory Where Registers Stored
   Change those registers which you wish to load by means of POKE statements. For example, if you wish to set the A-register to 5, the X-register to 1, and the Y-register to 0 then POKE 69,5 : POKE 70,1 : POKE 71,0.

3. CALL -193
   This calls the monitor RESTORE routine at memory location $FF3F. It returns the SAVEd, but altered, information back to the registers.

5.8.2 In-Depth Case Study: Analysis of an Animation Technique Using Fast Copying and Display Change And the SAVE/RESTORE Method for Loading Hardware Registers

There are occasions when it is desirable to animate a display, whether it be text, low-resolution graphics, or high-resolution graphics. Usually the animation requires a new picture which has some changes from the previous picture but, for the most part, is essentially the same as the old picture it replaced on the screen. Often, if the changes are made incrementally in full view on the screen, they draw too much attention to themselves and to the partially-altered state of the display and the desired visual effect is lost.

Finding a method of solution for this problem requires some knowledge about how graphics in general and Apple graphics in particular operate. You may easily have picked up the necessary background from reading the Applesoft programming manual you received with your Apple computer, or you may get a thorough briefing by jumping ahead and reading the chapter(s) on graphics which appear later in this text. (You might find it useful to read in Chapter 19 about the 'soft switches' and 'toggle switches' used in conjunction with graphics.)

However, you may wish to accept on faith that the key to solution of the animation problem is in the two-display-page capability in the Apple. This is available for text, low-resolution graphics, and high-resolution graphics. In each case you can show one display page and at the same time be writing (invisibly) on a second page that is not then visible.

You can solve your problem by writing or drawing onto the primary display page (page 1), then

1. Copy the primary display page (page 1) to the secondary display page (page 2). The copying has no visible effect upon the picture being displayed.

2. POKE the page-display switch so that visible display occurs from page 2. (Page 1 is now invisible.) You are showing the same picture, but from a different location inside the computer than that where the information was originally put.

3. Modify page 1 at leisure. When the modification is complete, cycle back through the same 1, 2, 3 process again, making sure that the first step each time is to POKE the page-display switch back to page 1 so that you are displaying from it while the copying occurs.

Fine theory, but does it work in practice? Not totally. It takes so long for a BASIC program to copy a display page that you don't really get to change the picture often enough to get animation effects.

However, moving a block of memory is something the computer must do quite often as part of its internal housekeeping operations. In fact, in the discussion of monitor capabilities in Section 5.4.2, the existence of such a routine was explicitly mentioned. There must be firmware in the system monitor to do this task if only we can find it and find a way of setting up the necessary parameters before calling it. As a high-usage machine-language routine it is likely to be well written and fast, probably much faster and involving much less wasted time and overhead than any BASIC program we could write.

A quick look in the Gazetteer under MOVE shows that there is indeed such a routine in the monitor. It uses the monitor general-usage parameters A1, A2, and A4. Great! We know how to use those, so we have our problem solved.

Nope! The description says that we also must set the Y-register to zero. We don't want to do any machine-language programming, at least not yet, so let's try the SAVE/RESTORE technique to set up the Y-register:

```
CALL -182: POKE 71,0: CALL -193
```

1. CALL -181 copies the registers into locations 69 through 73 with the Y-register copied into location 71.

2. POKE 71,0 puts 0 into location 71, the location where the Y-register was stored.

3. CALL -193 restores the registers to their original value (except that the contents of 71 which go to the Y-register are now 0 rather than the original value).

Thus we are ready to complete the re-

quirements for set-up of the subroutine we hoped to use.

Let's look at a highly documented version of a program to do the desired task:

```
                        Figure 5.9A
Fast Page Move for Animation (Using `SAVE' and `RESTORE' for Register Set-Up)

 Main Program - Create desired display on display page 1

100 GOSUB 500 : REM Go to subroutine which switches display to page 1 (not
                necessary first time thru, but necessary thereafter),
                copies page 1 to page 2 while page 1 is being displayed,
                then after move is completed displays same picture from
                page2

 Main Program - Now make desired changes invisibly on page 1 while page 2
                is being displayed

200 GOSUB 500 :REM Display modified picture, going through same process,
               first displaying it from page 1, moving it to page 2 and then
               displaying it from page 2 while page 1 is being modified
               Repeat GOSUB 500 each time a new picture prepared and ready
               to be displayed
.....................................................................
500 REM Animation Subroutine
510 POKE -16300,0 :REM Display page 1
520 POKE 60,0 :POKE 61,4 :REM Double-POKE parameter A1=1024 (Start of text/
               Low-Res Graphics page 1.) Change to A1=8192 if High-Res
530 POKE 62,255 :POKE 63,7 :REM Double-Poke parameter A2=2047 (End of text/
               Low-Res Graphics page 1). Change to A2=16383 if High-Res
540 POKE 66,0 :POKE 67,8 :REM Double-POKE parameter A4=2048 (Start of text/
               Low-Res Graphics page 2). Change to A4=16384 if High-Res
550 CALL -182 :POKE 71,0 :CALL -193 :REM Save registers; reset saved Y-reg
               to zero; Restore Registers.  Net effect: Set Y-reg to zero
               Parameter set-up for `MOVE' subroutine now complete
560 CALL -468 :REM Move (copy) display page 1 to display page 2
570 POKE -16299,0 :REM Display from page 2 and continue displaying from it
               until next time you enter this subroutine
580 RETURN
```

Notice that in this case study we had a mixture of parameters which were directly accessible in memory by means of POKEs and one parameter which was in a register and therefore was not directly accessible to POKEs (or PEEKs).

Many of the machine-language routines, especially the smaller ones, require you to set the A-, X-, and Y-registers. The methodology would be identical except that you would POKE the desired set-up values into locations 69, 70, and 71 between the CALL -182 and the CALL -193.

If you were *sure* that *all* registers you did not POKE had values which would be disregarded you could eliminate the CALL -182. However, the time and memory penalty for having it present unnecessarily is small and the possible penalties for leaving it out when it really is needed to avoid unintentional alteration of the registers can be very large. Thus, prudence dictates that you be *very* sure before you drop its use.

5.9
Direct Loading of Hardware Registers

5.9.1 The Concept

The direct approach is through the use of machine-language instructions. *Don't panic!* It's simple. If you don't like this approach after you've tried it, you always have the SAVE/RESTORE approach which doesn't require any machine language to fall back upon.

The Apple's microprocessor has a number of very simple and elementary instructions which will move information into and out of these registers — instructions that can easily be converted into POKEs and thus incorporated into a BASIC program. Among the available instructions are:

LDA — LoaD Accumulator
(from a specified memory location)
LDX — LoaD X-register
(from a specified memory location)
LDY — LoaD Y-register
(from a specified memory location)
STA — STore Accumulator
(in a specified memory location)
STX — STore X-register
(in a specified memory location)
STY — STore Y-register
(in a specified memory location)

In machine language these instructions consist of a single-byte operation code followed by a one- or two-byte hexadecimal address. You merely look up the hexadecimal form of the operation code, use the hex = > decimal conversion table we have previously used (Chapter 3) to convert it to decimal and POKE it. If you already know the decimal address where you wish to get or put the information, no conversion is needed. Just POKE (or double-POKE) it directly.

The process is surprisingly simple — *even if you don't know how to program in machine language.* Simple enough that I am willing to put it here before any discussion of machine-language programming.

5.9.2 In-Depth Case Study: Analysis of a Fast Data Copy Program Using the Direct Method for Calling A Subroutine Requiring Set-up of Parameters in Hardware Registers

This program is a fast data copy program which moves or copies an arbitrary block of information beginning at BEG and ending at EN to a new location starting at DEST. The same monitor routine used in Case Study 5.8.A is used. However, a distinctly different method of set-up is used for the hardware register (Y-register in both cases). In the previous case study the indirect method was used and the register set-up was accomplished by the line of code:

```
550 CALL -182 : POKE 71,0 : CALL -193
```

In this case a tiny machine-language program is written and converted into POKEs to accomplish

the same task. The corresponding line of code which sets up the Y-register is the following:

```
10 POKE 768,216:POKE 769,160:
   POKE 770,0:POKE771,76:POKE 772,44:
   POKE 773,254
```

Now let's analyze this program as if we had never seen it before and had no idea of its contents or method of operation.

```
1  REM ******************************
2  REM *                            *
3  REM *       CASE STUDY NO. 5.9   *
4  REM *   WHAT'S WHERE IN THE APPLE *
5  REM *   A FAST DATA COPY PROGRAM  *
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 users 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 pre-

sent 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 decimal location 768, i.e. hex location $300. For those interested in the mechanics of this program, enter the monitor (call -151) and disassemble starting at location $300 (300L).

9. We can analyze this code by using our single-byte hexadecimal = decimal conversion table:

POKE 768,216 =	300: D8
POKE 769,160 =	301: A0
POKE 770,0 =	302: 00
POKE 771,76 =	303: 4C
POKE 772,44 =	304: 2C
POKE 773,254 =	305: FE

Now we could go to the table of machine-language instructions in Chapter 6 or in the Apple Reference Manual and look up what these hexadecimal codes become when they are acted upon as computer instructions.

5.10 Modifying the 'Call' Statement to Include Parameters to be Passed

5.10.1 Concept

For the Modified CALL approach a short machine language utility program is written to allow a modified version of the Applesoft CALL to pass parameters to the hardware registers. The modified CALL first acts like a standard CALL to this utility. It transfers control to the utility program which uses software tools documented in the Atlas and Gazetter which are in the Applesoft interpreter to analyze the remainder of the non-standard CALL statement. As this particular utility analyzes each parameter it temproarily pushes the parameter to the system stack. Then it pops the information off the stack and uses it to load each register as required. With the registers loaded it transfers control to the subroutine to be CALLed.

The machine-lanaguage utility program can be converted into pseudo-BASIC and imbedded within the BASIc program or kept as a binary file and BLOADED as a patch to Applesoft.

Notice that the utility uses software tools available in Applesoft to do most of its work and that the program knows enough about how Applesoft stores and handles BASIC commands to skip around trouble which would normally occur from putting a non-standard CALL into Applesoft and letting Applesoft analyze it.

5.10.2 The New Applesoft 'CALL'

The purpose of t his Applesoft Utility program is to make it easier for Applesoft BASIC programmers to load registers by creating a new type of BASIC 'CALL' statement which allows them to specify the parameters to be loaded to the A-, X-, and Y-registers. Conventional CALL statments are not affected.

The utility may be entered as a few lines of Applesoft BASIC code (as shown in Figure 5.10A) imbedded in your Applesoft program, or treated as a binary patch to Applesoft, saving the 32 bytes of its machine-language version as a binary file and BLOADing the file before using the new feature of Applesoft.

Once the utility/patch is in place you will be able to use the following new form of the Applesoft 'CALL' statement:

CALL origin, A-expr, X-expr, Y-expr, location

where

origin = Decimal address of the entry to the utility program

A-expr = A single byte integer (0 < integer = < 255) or any variable or expression. The A-register or accumulator will be loaded with the parameter

X-expr = Same, but the X-register will be loaded with the parameter

Y-expr = Same, but the Y-register will be loaded with the parameter

location = Decimal address of the machine-language subroutine to be called with the specified parameters.

5.10.3 The Utility Program (or Binary Patch to Applesoft) which Implements the New 'CALL'

The machine-language utility program is a variant of one developed by C.K. Mesztenyi published in the Spring 1981 Apple Orchard. Its heart is a machine-language program shown in heavily-commented assembly-language form in figure 5.10B.

Even though it creates an extension to the 'CALL' capabilities of Applesoft, you might wonder why I put an assembly-language/machine-language description of fairly sophisticated machine-language program at this point in the book — Before I have introduced much about machine-language.

There are several parts to the rationale: (1) it is a useful software tool, (2) you don't need to know any machine language to put it into your BASIC programs or use it, (3) it fits neatly into the subject matter of this chapter, and (4) it is also a means of demonstrating the value and power of using small amounts of machine-language as a supplement to an Applesoft program while hiding it away to look and act like a part of the BASIC program.

In a sense the presence of the assembly-language/machine-language program here is a motivator for BASIC programmers to stick with me through the chapters on architecture/machine-language in order to learn how to do better BASIC programming. After you read Chapters 6, 7, and 8 you will find reading and understanding this program should be quite easy. After you read Chapter 19 you may be able to understand it at a much deeper level.

The secret to this program's simplicity and brevity (only 32 bytes in machine-language form) is that its author understands how the Applesoft interpreter works and how to use pieces of it to do much of his work for him. Using pseudo-instructions almost as a form of self-documentation, he tells the program how to find the zero-page LINUM location ($0050) of the next BASIC instruction (so that it can get back to the BASIc program easily) and that uses four firmware routines: FRMNUM ($DD67), CKHCOM ($DEBE), GETBYTC ($E6F5) and GETADR ($E752). These are routines used by Applesoft itself to analyze its instructions. This program is activated when a non-standard version of the CALL statement beginning with its location in memory appears right in the Applesoft program, and uses Applesoft's own software tools for analyzing this very special CALL (and to keep Applesoft from discovering the syntax error). It is an excellent example of the techniques advocated throughout this book.

Applesoft code which puts the entire machine-language utility into memory at location $300 (768 to decimal-oriented BASIC) is shown in figure 5.10A. Note that the 32 bytes which consitute the entire machine-language program fit into Pseudo-BASIC line 702. Also note that 3 of the 6 lines in the program are REMs which can be removed without affecting the operation of the program.

Incidentally, both the BASIC and machine-language parts of the program are deisgned to permit relocation. This means that the BASIC lines can be put anywhere in the BASIC main program. It also means that the $300 which specifies where the machine-language subroutine will be located may be changed to any value which will put the machine-language code into locations available for machine-language use — without any other changes being necessary to readjust the program to its new location in memory.

Figure 5.10A
Applesoft Utility to Allow an Additional CALL Statement Capable of Passing Arguments to Machine-Language Subroutines Used in Applesoft Programs

```
700 REM  Line 702 sets A$ to a monitor string that represents the loading
701 REM  address ($300)and the program to be loaded (expessed as hex
         digits after the colon)
702  A$ = "300:20 F5 E6 8A 48 20 F5 E6 8A 48 20 F5 E6 8A 48 20 BE DE 20 67
          DD 20 52 E7 68 A8 68 AA 68 6C 50 00 20"
703 REM  Line 704 Adds Monitor Commands for return to Applesoft; Line 705
         copies A$ into the keyboard input buffer, resets status
704 A$ = A$ + "N D823G"
705 FOR I=1 TO LEN(A$):POKE 511+I, ASC(MID$(A$,I,1))+128:NEXT:POKE 72,0:
         CALL -144
```

Figure 5.10B
Assembly- and Machine-Language Listing of Register-Loading Utility /Applesoft Patch

Pseudo-Instructions Used to Pre-Define Interface with Firmware (NO ML (NO ML Created)

```
Symbolic       |Comment
Pseudo-Instr   |
---------------------------------------------------------------------------
LINNUM  EQU $50   Where to find BASIC line number of next BASIC instruction
FRMNUM  EQU $DD76 Where to find subroutine to convert formulas to numbers
CHKCOM  EQU $DEBE Where to find subroutine to check commas
GETBYTC EQU $E6F5 Where to find subroutine to get expression, eval & put
                  into X-Reg
GETADR  EQU $E752 Where to find subroutine to convert number to 2-byte
                  address form
        ORG $300  Assemble code to be used at memory location starting at
                  locn $300
        OBJ $300  During Assembly process actually put it a location $300
```

Assembly-Language Symbolic Instructions and the Machine Language Bytes they generate

```
Symbolic      |Machine Language | Comments
Instructions  |Locn  |  Bytes   |
--------------|------|----------|-----------------------------------------
JSR GETBYTC     300  : 20 F5 E6  Get first parameter,evaluate it, convert
                                   to bytes =>X-Reg
TXA             303  : 8A        Move it from X-Reg where GETBYTC left it
                                   to Accumulator
PHA             304  : 48        Push it onto stack(makes program relocatbl
JSR GETBYTC     305  : 20 F5 E6  Repeat for 2nd parameter
TXA             308  : 8A
PHA             309  : 48
JSR GETBYTC     30A  : 20 F5 E6  Repeat for 3rd parameter
TXA             30D  : 8A
PHA             30E  : 48
JSR CHKCOM      30F  : 20 BE DE  4th param: check for comma present.
                                   Get ready for param
JSR FRMNUM      312  : 20 67 DD  Evaluate it as a number
JSR GETADR      315  : 20 52 E7  Convert to address bytes
PLA             318  : 68        Pull 3rd param from stack to A-reg
TAY             319  : A8        Move 3rd param from A-Reg to Y-Reg
PLA             31A  : 68        Pull 2nd param from stack to A-reg
TAX             31B  : AA        Move 2nd param from A-Reg to X-Reg
PLA             31C  : 68        Pull 1st param from stack to A-Reg;
                                   All registers loaded
JMP(LINNUM)     31D  : 6C 50 00  Jump to(line number of)next BASIC instruct
```

The Lam technique used in figure 5.10A for tricking the monitor into stuffing the machine-language program (figure 5.10B) into a BASIC program in such a fashion that it can be executed as part of a running BASIC program and then return control back to the running Applesoft program is described in detail in Section 8.3 (Tricking the Apple Monitor...).

Want a good educational mini-project? By the end of this book you should be able to figure out the whole story of what is happening not only in the process of stuffing, but in the utility program itself. You will want to decode the Hex instructions using figure 6.5B; look up in the Part II Atlas the called subroutines and the zero-page locations used; and check in Section 19.5.3 how the Applesoft Interpreter represents the CALL instruction. The task is detailed and onerous, but you will learn a great deal about putting the ideas in this book into practice if you decide to undertake it.

5.10.4 Example: Use of the New CALL with Monitor Subroutine P R N T A X as a Quick Decimal-Hexadecimal Converter

An example of the use of this utility would be a simple decimal = hexadecimal converter which consists of no more than one of the new modified 'CALL' statements.

CALL 768, V1, V2, 0, 63809

converts both V1 and V2 (which can be expressed as decimal numbers, variables or even as arithmetic expressions to be computed) to hexadecimal and prints the hexadecimal answers on the computer screen!

Control is initially transferred to 768 ($300), the location of the utility. It sends V1 to the A-register; V2 to the X-register and 0 to the Y-register.

Then it transfers control to 63809 ($F941), the monitor PRNTAX routine. This routine prints the contents of the A- and X-registers in hexadecimal.

In an additional suggestion, sure to gladden the hearts of addicts an & extension is suggested. Using Applesoft variable names such as

A (for Accumulator)
X (for X-register)
Y (for Y-register)
NAME (for a variable name mnemonic to the name of the machine-language subroutine specifying the location of that routine)

the extension would allow the arguments to be passed in the following form:

&, A, X, Y, NAME

Mesztenyi and his editor Val Golding imbed this idea in a program which accepts values to be put into the A- and X-registers, puts them there, then prints them using the PRTAX monitor subroutine, then prepeats the process endlessly to create a simple decimal-to-hexadecimal converter.

Figure 5.11A

```
]LIST

 10 REM

 AMPERSAND REGISTER LOADER
 BY C K MESZTENYI & VAL GOLDING

 Passes arguments to A, X, and Y
 Registers and Program Counter,
 using the Ampersand

 50 GOTO 500
100 INPUT A,X
110 & ,A,X,Y,AX
120 GOTO 100
500 POKE 1013,76:POKE 1014,0:POKE
    1015,96: REM
Set up Ampersand Vector

510 AX = 63809: REM
Set up variable as CALL
address for PRNTAX

520 Y=0
700 A$ = "6000:20 F5 E6 8A 48 20
     F5 E6 8A 48 20 F5 E6 8A 48 2
     0 BE DE 20 67 DD 20 52 E7 68
      A8 68 AA 68 6C 50 00 20 ND8
     23G"
710 FOR I = 1 TO LEN(A$): POKE
    511 + I, ASC ( MID$ (A$,I,1)
    ) + 128: NEXT : POKE 72,0: CALL
    -144
720 GOTO 100:REM

Above routine by S H Lam stuffs
all the machine code in memory.

To relocate it, change the POKE
1015 in Line 500 and the address
in Line 700
```

All that is necessary to create the & capability in addition to the CALL capability is to change the JUMP instruction associated with the '&' function in the DOS and Monitor Vector Table in page 3 of memory (see figure 13.2A) so that it points to the utility rather than to its default $FF65, normal reentry to the top of the monitor.

This JUMP is in locations $3F5-$3F7 (decimal 1013-1015). There is no need to change the JUMP part of the command in $3F5, just change the destination address. In our case, with our utility at $300, we would change the address to $300 (decimal 768). To do this all we need do is a double-POKE using the techniques discussed in Section 4.3. Since $300 is a page boundary and hence is divisible by 256, the decimal POKE looks just like the hexadecimal address — least significant part first, i.e. POKE 1014,00: POKE 1015,3.

To check that you still understand the double-POKE, try double-POKE conversion of Mesztenyi's chosen location $6000. If you do not get the same POKEs as that in his Ampersand Register Loader program, figure 5.10.B, you need to review Section 4.3. (*Note:* The program in figure 5.11.A also rePOKEs the JUMP — $4C into $3F4 — POKE 1013,76. This is unnecessary unless you have made an unorthodox modification to your jump table before using the program.)

Chapter VI
Apple Architecture I

6.1
Architecture in Perspective: Not Just for Assembly-Language Programmers

This chapter deals primarily with characteristics of the Apple II at the machine level. It contains reference material that can help you understand how the Apple II system is organized from a combined hardware software systems viewpoint.

Although most of the topics covered aren't for the typical beginning BASIC programmer, beginners will find these points increasingly valuable as their BASIC programming becomes more sophisticated and system-dependent. Assembly language or machine language programmers will find they are familiar with many of the topics.

However, this chapter is not aimed primarily at assembly or machine-language programmers. It is not intended to be an assembly or machine-language programming manual. Instead it is intended to provide important information about the innerworkings of the Apple II at a level of detail most helpful to the BASIC programmer (and the assembly-language programmer who does not have detailed familiarity with the Apple II system). Its greatest usefulness should be to those faced with the following problems:

1. Learning enough about Apple II hardware and system organization so that you can understand what hardware and software are in the system.

2. Learning enough about Apple II hardware and system organization so that you can intelligently follow both the later, more-detailed documentation in this work and other available documentation.

3. Learning to interface BASIC programs with the Apple II system hardware and firmware.

4. Learning to interface BASIC programs with machine-language programs written by others but not imbedded into the Apple II system.

5. Learning enough about Apple II hardware and system organization so that you can read and understand straightforward and well-documented assembly or machine-language software aids.

6. Learning enough about Apple II hardware and system organization so that with the aid of later sections of this work, and an assembly-language programming manual, you can write short segments of machine-language code which can be used in a BASIC environment.

This chapter does not attempt to teach programming techniques or to illustrate the ideas presented with an adequate number of actual programming examples to teach programming. It does present fundamental background information.

6.2
A Simplified Hardware Block Diagram and 'Programmers' Model' of the Apple II System

The Apple II system is built around the MOS 6502, an 8-bit microprocessor with a 16-bit program counter and addressing capability.

The key to understanding the operation of any processor is understanding the information manipulating capabilities, most of which are performed in special locations, known as registers.

Registers are memory locations, just like addressable memory locations, except that they have significant amounts of logic arithmetic, and other logical circuitry used to manipulate information and control its flow built into them. Each register has associated with it certain machine-language instructions or modes of operation at the machine-language level, wich give it capabilities that regular addressable memory locations lack.

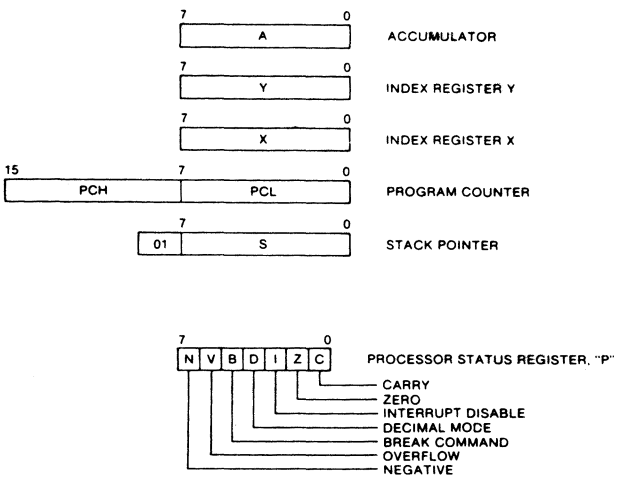
In some computers, the registers are given memory addresses and can be referred to by address number. In the MOS 6502 and the Apple II system, the registers are not assigned memory location addresses.

The Programming Model of a microprocessor identifies its main registers. I find it most useful

when expressed in the context of a simplified block diagram of the hardware bus system, which is the highway network that allows information to move between the registers and between the registers and memory. (See figure 6.2A) Later additional system components and information paths will be added to get a more complete diagram.

Figure 6.2A

PROGRAMMING MODEL



This model contains five 8-bit hardware registers (the A-, P-, S-, X- and Y-registers), one 16-bit register (the program counter) and 65536 words of memory:

1. A-Register (often called the accumulator) is the primary arithmetic and logical register in the computer. For example, in additions, the addend is in the A-Register before execution and the result is there after execution.

2. X-Register (often called an index register) has special capabilities for acting as an offset and/or as a counter. It has special instructions associated with it that allow it to be incremented, decremented, or compared in value with memory.

3. Y-Register (often called another index register) is similar in capabilities and use to the X-Register.

4. S-Register (often called the stack pointer) is used with a 'push-down, pop-up stack', which is used for re-enterant coding, e.g. for saving return addresses upon entry to subroutines.

5. P-Register (often called the processor status register), contains seven single-bit flags that identify special conditions of the computer: arithmetic carry/no-carry, zero/non-zero result, interrupt disable/normal, decimal/binary mode, break/no-break condition, overflow/non-overflow condition and negative/non-negative result.

6. PC—The Program Counter, a 16-bit register (divided into two eight-bit bytes) tells the computer where to get its next instruction.

Machine- and assembly-language programmers find that their work centers about the control of information flow to and from these hardware-registers.

Elementary BASIC programmers seldom, if ever, need to know anything about what is going on at the register-level of the system. However, if they do gain an appreciation of what goes on in the computer at this level they may be able to write better, faster, and more memory-efficient programs.

For advanced BASIC programmers who try to take advantage of system software permanently imbedded in the Apple II, a knowledge of the general architecture of the Apple II system at this level can be very valuable, even if they intend never to write any programs in assembly language. For example, many of the more powerful routines or subroutines imbedded in Apple II firmware require that information to control their actions be passed to them by pre-setting values in the A-Register, X-Register and/or Y-Register.

6.3 Bit-Oriented Information Representation and Addressing

(Abandon Decimal Numbers All Ye Who Enter Here!)

6.3.1 Hexadecimal as a Convenient Human-Oriented Method of Abbreviation (Not As A Strange Number System)

Since the Apple II is primarily a binary computer system, at the machine level it does its addressing with bits rather than with decimal numbers.

Conversion between binary-bit and decimal number format is straightforward, but laborious once you get beyond the 2-, 3-, or 4-bit numbers you can convert in your head.

Binary addresses in the Apple are typically 16 bits long. However, people have great difficulty dealing with long strings of 0's and 1's. How long could you remember the following binary address? 1101011101010001. It really is not as hard as it seems — providing you have a good technique for reorganizing the information into a better form.

Many microcomputers, such as the Apple II,

deal with information in 8-bit packets called bytes. Each 8-bit byte can be broken down into two, 4-bit nybbles. There are 16 possible values for each nybble. A commonly used assignment of symbolic abbreviations follows:

```
0000=0  0001=1  0010=2  0011=3  0100=4
0101=5  0110=6  0111=7  1000=8  1001=9
1010=A  1011=B  1100=C  1101=D
1110=E  1111=F
```

Using this table, a long binary number can be converted into a string of one-quarter as many symbols as per the example below: (spaces are inserted every four bits to make the conversion pattern more obvious).

Binary Form	0001	1001	1101	0011
Hex abbreviation	1	9	D	3

Obviously the abbreviated form is much easier for us to remember and use than the long 16-bit form, but the conversion is trivial and can be easily figured.

This particular abbreviation doesn't look like a common, everyday decimal number, but many do. For example,

Binary Form	0001	0111	0100	0011
Hex abbreviation	1	7	4	3

Both Integer BASIC and Applesoft BASIC provide the capability for doing integer number calculations. They use 16-bit integer numbers, not to represent the numbers 0 through 65536, but rather the more useful range of −32768 to +32767.

You might think these calculations could be figured by using one bit for the sign and 15 bits for the magnitude of numbers, just as in everyday arithmetic. However, there would be several undesirable by-products that would increase the complexity of the computer hardware. For one, you would have two numbers that had identical values: +0 and −0. This seems like a minor point, but it can be a problem in the design of electronic circuitry. Early computers often used the sign-and-magnitude form for numbers, but now, almost all computers represent negative binary numbers the way Leibniz recommended when he made the first analysis of their properties in 1679.

This procedure uses radix-complement numbers. In the case of binary numbers the radix (number of symbols used) is 2, so radix-complement numbers are two's-complement numbers. In the decimal system where there are 10 symbols used (0 through 9), the radix is 10 and the equivalent complements are called ten's complement numbers.

The complement of a number of a fixed length is that number, which when added to the number, adds up to all zeros. Suppose you had 90,000 miles on the odometer of your car and decided to roll it backward 1 mile; you would, of course, get 89999 on the odometer. But what if you started out with 0 miles and rolled it backward 1 mile? What would you get? 99999. The wheels on the odometer have no other way of representing the number −1. 99999 is the tens-complement method of representing −1.

With binary numbers you have only two symbols, 0 and 1, so you must use the two's complement instead of the 10's complement. If I run a five-bit-long binary odometer backwards one mile from zero, I get 11111 instead of 99999. It is the binary two's-complement representation of −1 for 5-bit numbers.

Interestingly enough I can convert any binary number to its negative two's complement form merely by changing all 0's to 1's and all 1's to 0's and adding 1 to the result. Thus if I have the 5-bit long number 00101 (decimal 5), I can get the 5-bit two's complement by switching the bits to 11010 then adding 1 to get 11011. This is the five-bit-long two's complement form of decimal −5. I can use exactly the same process to convert from two's complement (negative) numbers to their positive equivalents. Thus if I start the five-bit two's complement form of the number −5 (decimal) — 11011, I can convert it to its equivalent positive number by the same procedure. If I change each bit getting 00100, then add 1, I get 00101 (decimal 5).

In calculations, using either mechanical rotating wheels as on an odometer, or on a mechanical desk calculator, or on a modern digital computer, the use of the complement form of numbers lets you pass back and forth through the number zero with an absolute minimum of computational complications. That is why the two's complement form is almost universally used as the method of representing negative (integer) numbers in modern binary computers.

Nevertheless, the use of 16-bit signed integer numbers for addressing as well as for computation can lead to some interesting anomalies in their representation by equivalent decimal numbers.

As you climb up the binary/decimal number scale from 0000000000000000 (0) to 0111111111111111 (32767) there are no surprises or problems. However, when you reach the point where the sixteenth bit must come into play you go one further. Suddenly the new combination 1000000000000000 represents −32768. From then on you count backwards in decimal. The new 1 bit in the sixteenth bit position indicates that the new

value is negative. What is its value? You can find out by changing all zeros to ones and adding 1. The number is −32768. Quite a discontinuity! Moreover, as you continue to count up the binary address scale used internally in the Apple (and most other computers), you now find that you are counting backwards in decimal. When you finally reach binary 1111111111111111 the decimal value is −1.

This characteristic alone makes the use of decimal addresses a problem for the Apple II user. (*Note:* The problem for Applesoft BASIC users is not as acute as it is for Integer BASIC users. The Applesoft interpreter will accept unsigned (positive) integers greater than 32767, thereafter making no distinction between them and the signed binary integers that create the same bit pattern. However, Integer BASIC users cannot enter numbers larger than 32767, so they must bear the full brunt of the discontinuity and backward counting.)

It is important to note that while this looks like the decimal number 1743, it does not represent the same counting number. The symbol combination 1743 in decimal, the number system using 10 symbols, is an abbreviation for

> 1 thousand, 7 hundreds, four tens (forty) and 3 units = $1 * 10^3 + 7 * 10^2 + 4 * 10^1 + 3 * 10^0$.
>
> Our abbreviation system used 16 rather than 10 symbols. It turns out that mathematically it has identical characteristics to a hexadecimal or base-16 number system. The counting value of the number represented when converted to decimal numbers is
>
> $1 * 16^3 + 7 * 16^2 + 4 * 16^1 + 3 * 16^0$ ©4096 + 1792 + 64 + 3 = 5955.

To avoid confusion when you are using both decimal numbers and this hexadecimal method of abbreviation of binary numbers, you could always follow the numeric symbol with an explanation, e.g. 1743 (decimal) or 1743 (Hex). However, Apple programmers have adopted the general convention that hexadecimal numbers should be prefixed with a $ sign and decimal numbers left alone. Thus 1743 is the decimal number one thousand seven-hundred forty-three, while $1743 is the hexadecimal number 1743 (which has the same counting value as the decimal number 5955).

6.3.2 Hexadecimal Addresses and Negative Decimal Addresses

Using the hexadecimal method of binary-bit abbreviation the Apple II system address range:

Binary	0000	0000	0000	0000	to
	1111	1111	1111	1111	
becomes					
Hexadecimal	0	0	0	0	to
	F	F	F	F	

Once you begin to use these hexadecimal abbreviations you may soon find that the 4-digit hexadecimal addresses are not only shorter than the 5-digit decimal addresses for the same memory locations, but they are much more closely related to the natural break-points in the system architecture. Hexadecimal addresses are much easier to remember and use than decimal addresses for the Apple II system. To your surprise, you will begin to think of decimal addresses as annoying and wish that BASIC PEEKs and POKEs would accept hexadecimal as well as decimal addresses.

Decimal numbers are complicated to use as addresses, for several reasons. We have already mentioned that most of the interesting and significant addresses in the Apple II, when expressed in decimal form, are awkward numbers like 16384. But when they are expressed in hexadecimal, they are conveniently rounded numbers like $4000.

You will also note that many of the Apple manuals specify the use of negative decimal addresses in many of their PEEKs, POKEs, and CALLs. The idea of a negative address is annoying to many people, especially when address 0 is at one end of memory and −1 at the other. In the middle, adding 1 to 32767 may give −32768 as a result. It is interesting and instructive to examine why this occurs.

6.4 The Stored Program, The Program Counter and The FETCH-EXECUTE Cycle — The Heart of the Stored-Program Computer

6.4.1 Instructions and Data Both Stored in Memory as Binary Bits

At the machine-language level, as in BASIC, a running program requires two things: instructions and data. At the machine level both are expressed in binary bits and both can be stored in the computer's memory.

Some tasks require many, many instructions to define what must be done, but require only a small amount of data; others require huge amounts of data, but only a few simple instructions. Years ago

computer designers found that it was technically desirable and cost effective to build computers in which the instructions and data could be stored interchangeably in the same memory.

A computer determines whether a particular byte of information is a part of an instruction or whether it is part of a data item by examining the program in the computer. The program counter, which is set at the start of any computer run at the first instruction to be performed, tells the computer where to FETCH its first instruction and thereafter from where to FETCH every subsequent instruction. Since the program counter contains 16 bits, it can specify the choice of any one of up to 2^{16} or 65,536 memory locations as the location of an instruction to be FETCHed. Anything that is FETCHed is treated as an instruction.

When an instruction is EXECUTEd any information it uses will be treated as data. This is true even if the item of information being manipulated is itself part of the program being executed. (It may seem odd to BASIC programmers to consider doing computations on their own program, but it is perfectly feasible and widely done in machine language — even though it violates all of the tenets of 'structured programming'.)

6.4.2 The FETCH-EXECUTE Cycle and How the Computer Distinguishes Instructions from Data

The computer will FETCH each instruction, analyze and EXECUTE it in accordance with the following rigidly-defined cycle of operation:

1. Using the program counter to determine where to get it from, the computer FETCHes the byte of information at the location designated by the address in the program counter. The computer automatically changes the program counter to the address of the next instruction to be performed, and
2. EXECUTES the operation specified by the instruction.

'FETCH'ing the information brings the byte into circuitry where it decodes it as a code that specifies the next operation to be performed. Not surprisingly, this byte is called the operation code part of the computer instruction. The instruction will also contain information on finding the data that is to be used in the operation. Normally this is done by specifying the address of the memory location in which the data to be used may be found. The MOS 6502 has a number of different modes for addressing or specifying the address. Probably the most basic of these is by means of an absolute address. Each memory location is permanently assigned a number expressed in binary digits from 0000000000000000 to 1111111111111111 to distinguish it from any other memory location. An absolute address is just this 16-bit, (2-byte) permanently-assigned absolute memory location number.

The computer decodes the operation code and determines that the absolute addressing scheme is being used. It knows that an absolute address requires two bytes of memory and that this address will immediately follow the operation code to create a three-byte-long instruction. It also knows that the next instruction should be immediately after the completion of this instruction, so it sets the program counter to its original value +3, the location of the start (the operation code) of the next instruction.

The control circuitry of the computer continues its analysis of what the instruction tells the computer to do. It uses the results of this analysis to set up the computer to do whatever the instruction instructs it to do. This completes the FETCH phase of computer operation. Then the computer goes on to EXECUTE the instruction it has set itself up to do.

Once it has completed executing the instruction, the computer must FETCH the next instruction to tell it what to do next. The program counter tells the computer where the instruction is located. Then the computer goes to this location looking for the operation code of the next instruction, which is decoded to tell it what to do. It also finds the type — and hence the length of the address — included as part of the instruction. The program counter is automatically incremented by 1,2 or 3 — the relevant number to step beyond the address to point at the next instruction's operation code. The cycle continues:

FETCH	(Get and decode the instruction - increment the PC - set up to do whatever is required by the instruction)
EXECUTE	(Do it.)
FETCH	(Get the instruction - increment the PC - set up to do it)
EXECUTE	(Do it.)
F E T C H	(Get the instruction - increment the PC - set up to do it.)
EXECUTE	(Do it.)
...	
...	

6.5 The Repertoire of Hardware-Implemented Instructions Built into the Apple II System

6.5.1 The Total Repertoire

Figure 6.5A is a list of the hardware-implemented instructions built into the MOS6502 microprocessor used in the Apple II system. This list gives the symbolic abbreviation for each instruction and a brief description. Figure 6.5B is another list that may prove more useful if you find a byte in memory that you believe is the operation code of a hardware instruction. You may wish to find out what the instruction is and what it does. (*Note:* The Apple II includes a disassembler that will do this look-up process for you very easily.

Figures 6.5C1 through 6.5C5 provide expanded descriptions of exactly what each instruction does, each of the operation codes and addressing structures associated with it, and even the number of machine cycles of time required to execute it.

6.5.2 What Instructions are Most Important to Semi-BASIC Programmers?

You'll find that this repertoire of instructions is valuable and comprehensive. Even experienced assembly-language or machine language programmers will normally use only a modest number of these instructions in their everyday programming activities. A person using primarily BASIC programming, but who imbeds an occasional machine-language subroutine — perhaps written by someone else — should be able to read and understand the meaning of most instructions.

Most of these instructions will be used to move information into the hardware registers or the inverse instructions needed to move information back from the registers into memory. (Sometimes such operations are needed to transfer results from the machine-language routine back to where a BASIC program can use them.)

Figure 6.5A

6502 MICROPROCESSOR INSTRUCTIONS

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right one Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

Figure 6.5B

HEX OPERATION CODES

00 — BRK
01 — ORA — (Indirect, X)
02 — NOP
03 — NOP
04 — NOP
05 — ORA — Zero Page
06 — ASL — Zero Page
07 — NOP
08 — PHP
09 — ORA — Immediate
0A — ASL — Accumulator
0B — NOP
0C — NOP
0D — ORA — Absolute
0E — ASL — Absolute
0F — NOP
10 — BPL
11 — ORA — (Indirect), Y
12 — NOP
13 — NOP
14 — NOP
15 — ORA — Zero Page, X
16 — ASL — Zero Page, X
17 — NOP
18 — CLC
19 — ORA — Absolute, Y
1A — NOP
1B — NOP
1C — NOP
1D — ORA — Absolute, X
1E — ASL — Absolute, X
1F — NOP
20 — JSR
21 — AND — (Indirect, X)
22 — NOP
23 — NOP
24 — BIT — Zero Page
25 — AND — Zero Page
26 — ROL — Zero Page
27 — NOP
28 — PLP
29 — AND — Immediate
2A — ROL — Accumulator
2B — NOP
2C — BIT — Absolute
2D — AND — Absolute
2E — ROL — Absolute
2F — NOP
30 — BMI
31 — AND — (Indirect), Y
32 — NOP
33 — NOP
34 — NOP
35 — AND — Zero Page, X
36 — ROL — Zero Page, X
37 — NOP
38 — SEC
39 — AND — Absolute, Y
3A — NOP
3B — NOP
3C — NOP
3D — AND — Absolute, X
3E — ROL — Absolute, X
3F — NOP
40 — RTI
41 — EOR — (Indirect, X)
42 — NOP
43 — NOP
44 — NOP
45 — EOR — Zero Page
46 — LSR — Zero Page
47 — NOP
48 — PHA
49 — EOR — Immediate
4A — LSR — Accumulator
4B — NOP
4C — JMP — Absolute
4D — EOR — Absolute
4E — LSR — Absolute
4F — NOP
50 — BVC
51 — EOR (Indirect), Y
52 — NOP
53 — NOP
54 — NOP
55 — EOR — Zero Page, X
56 — LSR — Zero Page, X
57 — NOP
58 — CLI
59 — EOR — Absolute, Y
5A — NOP
5B — NOP
5C — NOP
5D — EOR — Absolute, X
5E — LSR — Absolute, X
5F — NOP
60 — RTS
61 — ADC — (Indirect, X)
62 — NOP
63 — NOP
64 — NOP
65 — ADC — Zero Page
66 — ROR — Zero Page
67 — NOP
68 — PLA
69 — ADC — Immediate
6A — ROR — Accumulator
6B — NOP
6C — JMP — Indirect
6D — ADC — Absolute
6E — ROR — Absolute
6F — NOP
70 — BVS
71 — ADC — (Indirect), Y
72 — NOP
73 — NOP
74 — NOP
75 — ADC — Zero Page, X
76 — ROR — Zero Page, X
77 — NOP
78 — SEI
79 — ADC — Absolute, Y
7A — NOP
7B — NOP
7C — NOP
7D — ADC — Absolute, X NOP
7E — ROR — Absolute, X NOP
7F — NOP
80 — NOP
81 — STA — (Indirect, X)
82 — NOP
83 — NOP
84 —STY — Zero Page
85 — STA — Zero Page
86 — STX — Zero Page
87 — NOP
88 — DEY
89 — NOP
8A — TXA
8B — NOP
8C — STY — Absolute
8D — STA — Absolute
8E — STX — Absolute
8F — NOP
90 — BCC
91 — STA — (Indirect), Y
92 — NOP
93 — NOP
94 — STY — Zero Page, X
95 — STA — Zero Page, X
96 — STX — Zero Page, Y
97 — NOP
98 — TYA
99 — STA — Absolute, Y
9A — TXS
9B — NOP
9C — NOP
9D — STA — Absolute, X
9E — NOP
9F — NOP
A0 — LDY — Immediate
A1 — LDA — (Indirect, X)
A2 — LDX — Immediate
A3 — NOP
A4 — LDY — Zero Page
A5 — LDA — Zero Page
A6 — LDX — Zero Page
A7 — NOP
A8 — TAY
A9 — LDA — Immediate
AA — TAX
AB — NOP
AC — LDY — Absolute
AD — Absolute
AE — LDX — Absolute
AF — NOP
B0 — BCS
B1 — LDA — (Indirect), Y
B2 — NOP
B3 — NOP
B4 — LDY — Zero Page, X
B5 — LDA — Zero Page, X
B6 — LDX — Zero Page, Y
B7 — NOP
B8 — CLV
B9 — LDA — Absolute, Y
BA — TSX
BB — NOP
BC — LDY — Absolute, X
BD — LDA — Absolute, X
BE — LDX — Absolute, Y
BF — NOP
C0 — CPY — Immediate
C1 — CMP — (Indirect, X)
C2 — NOP
C3 — NOP
C4 — CPY — Zero Page
C5 — CMP — Zero Page
C6 — DEC — Zero Page
C7 — NOP
C8 — INY
C9 — CMP — Immediate
CA — DEX
CB — NOP
CC — CPY — Absolute
CD — CMP — Aosolute
CE — DEC — Absolute
CF — NOP
D0 — BNE
D1 — CMP — (Indirect), Y
D2 — NOP
D3 — NOP
D4 — NOP
D5 — CMP — Zero Page, X
D6 — DEC — Zero Page, X
D7 — NOP
D8 — CLD
D9 — CMP — Absolute, Y
DA — NOP
DB — NOP
DC — NOP
DD — CMP — Absolute, X
DE — DEC — Absolute, X
DF — NOP
E0 — CPX — Immediate
E1 — SBC — (Indirect, X)
E2 — NOP
E3 — NOP
E4 — CPX — Zero Page
E5 — SBC — Zero Page
E6 — INC — Zero Page
E7 — NOP
E8 — INX
E9 — SBC — Immediate
EA — NOP
EB — NOP
EC — CPX — Absolute
ED — SBC — Absolute
EE — INC — Absolute
EF — NOP
F0 — BEQ
F1 — SBC — (Indirect), Y
F2 — NOP
F3 — NOP
F4 — NOP
F5 — SBC — Zero Page, X
F6 — INC — Zero Page, X
F7 — NOP
F8 — SED
F9 — SBC — Absolute, Y
FA — NOP
FB — NOP
FC — NOP
FD — SBC — Absolute, X
FE — INC — Absolute, X
FF — NOP

INSTRUCTION CODES

Figure 6.5C-1

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
ADC						
Add memory to accumulator with carry	A+M+C → A,C	Immediate	ADC #Oper	69	2	√√√--√
		Zero Page	ADC Oper	65	2	
		Zero Page,X	ADC Oper,X	75	2	
		Absolute	ADC Oper	6D	3	
		Absolute,X	ADC Oper,X	7D	3	
		Absolute,Y	ADC Oper,Y	79	3	
		(Indirect,X)	ADC (Oper,X)	61	2	
		(Indirect),Y	ADC (Oper),Y	71	2	
AND						
"AND" memory with accumulator	A∧M → A	Immediate	AND #Oper	29	2	√√----
		Zero Page	AND Oper	25	2	
		Zero Page,X	AND Oper,X	35	2	
		Absolute	AND Oper	2D	3	
		Absolute,X	AND Oper,X	3D	3	
		Absolute,Y	AND Oper,Y	39	3	
		(Indirect,X)	AND (Oper,X)	21	2	
		(Indirect),Y	AND (Oper),Y	31	2	
ASL						
Shift left one bit (Memory or Accumulator)	(See Figure 1)	Accumulator	ASL A	0A	1	√√√---
		Zero Page	ASL Oper	06	2	
		Zero Page,X	ASL Oper,X	16	2	
		Absolute	ASL Oper	0E	3	
		Absolute,X	ASL Oper,X	1E	3	
BCC						
Branch on carry clear	Branch on C=0	Relative	BCC Oper	90	2	------
BCS						
Branch on carry set	Branch on C=1	Relative	BCS Oper	B0	2	------
BEQ						
Branch on result zero	Branch on Z=1	Relative	BEQ Oper	F0	2	------
BIT						
Test bits in memory with accumulator	A∧M, M_7 → N, M_6 → V	Zero Page	BIT* Oper	24	2	M_7√---M_6
		Absolute	BIT* Oper	2C	3	
BMI						
Branch on result minus	Branch on N=1	Relative	BMI Oper	30	2	------
BNE						
Branch on result not zero	Branch on Z=0	Relative	BNE Oper	D0	2	------
BPL						
Branch on result plus	Branch on N=0	Relative	BPL oper	10	2	------
BRK						
Force Break	Forced Interrupt PC+2 ↓ P ↓	Implied	BRK*	00	1	---1--
BVC						
Branch on overflow clear	Branch on V=0	Relative	BVC Oper	50	2	------

Note 1: Bits 7 and 7 are transferred to the status register. If the result of A∧M is then the 1, otherwise Z = 0

Note 2: A BRK command cannot be masked by setting I

Figure 6.5C-2

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
BVS						
Branch on overflow set	Branch on V=1	Relative	BVS Oper	70	2	------
CLC						
Clear carry flag	0 → C	Implied	CLC	18	1	---0--
CLD						
Clear decimal mode	0 → D	Implied	CLD	D8	1	-0----
CLI						
	0 → I	Implied	CLI	58	1	---0--
CLV						
Clear overflow flag	0 → V	Implied	CLV	B8	1	0-----
CMP						
Compare memory and accumulator	A − M	Immediate	CMP #Oper	C9	2	√√√---
		Zero Page	CMP Oper	C5	2	
		Zero Page, X	CMP Oper,X	D5	2	
		Absolute	CMP Oper	CD	3	
		Absolute,X	CMP Oper,X	DD	3	
		Absolute,Y	CMP Oper,Y	D9	3	
		(Indirect,X)	CMP (Oper,X)	C1	2	
		(Indirect),Y	CMP (Oper),Y	D1	2	
CPX						
Compare memory and index X	X − M	Immediate	CPX #Oper	E0	2	√√√---
		Zero Page	CPX Oper	E4	2	
		Absolute	CPX Oper	EC	3	
CPY						
Compare memory and index Y	Y − M	Immediate	CPY #Oper	C0	2	√√√---
		Zero Page	CPY Oper	C4	2	
		Absolute	CPY Oper	CC	3	
DEC						
Decrement memory by one	M − 1 → M	Zero Page	DEC Oper	C6	2	√√----
		Zero Page,X	DEC Oper,X	D6	2	
		Absolute	DEC Oper	CE	3	
		Absolute,X	DEC Oper,X	DE	3	
DEX						
Decrement index X by one	X − 1 → X	Implied	DEX	CA	1	√√----
DEY						
Decrement index Y by one	Y − 1 → Y	Implied	DEY	88	1	√√----

Figure 6.5C-3

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
EOR						
"Exclusive-Or" memory with accumulator	A ∨ M → A	Immediate	EOR #Oper	49	2	√\ ---
		Zero Page	EOR Oper	45	2	
		Zero Page,X	EOR Oper,X	55	2	
		Absolute	EOR Oper	4D	3	
		Absolute,X	EOR Oper,X	5D	3	
		Absolute,Y	EOR Oper,Y	59	3	
		(Indirect,X)	EOR (Oper,X)	41	2	
		(Indirect),Y	EOR (Oper),Y	51	2	
INC						
Increment memory by one	M + 1 → M	Zero Page	INC Oper	E6	2	√√----
		Zero Page,X	INC Oper,X	F6	2	
		Absolute	INC Oper	EE	3	
		Absolute,X	INC Oper,X	FE	3	
INX						
Increment index X by one	X + 1 → X	Implied	INX	E8	1	√√----
INY						
Increment index Y by one	Y + 1 → Y	Implied	INY	C8	1	√√----
JMP						
Jump to new location	(PC+1) → PCL	Absolute	JMP Oper	4C	3	------
	(PC+2) → PCH	Indirect	JMP (Oper)	6C	3	
JSR						
Jump to new location saving return address	PC+2 ↓,	Absolute	JSR Oper	20	3	------
	(PC+1) → PCL					
	(PC+2) → PCH					
LDA						
Load accumulator with memory	M → A	Immediate	LDA #Oper	A9	2	√√----
		Zero Page	LDA Oper	A5	2	
		Zero Page,X	LDA Oper,X	B5	2	
		Absolute	LDA Oper	AD	3	
		Absolute,X	LDA Oper,X	BD	3	
		Absolute,Y	LDA Oper,Y	B9	3	
		(Indirect,X)	LDA (Oper,X)	A1	2	
		(Indirect),Y	LDA (Oper),Y	B1	2	
LDX						
Load index X with memory	M → X	Immediate	LDX #Oper	A2	2	√√----
		Zero Page	LDX Oper	A6	2	
		Zero Page,Y	LDX Oper,Y	B6	2	
		Absolute	LDX Oper	AE	3	
		Absolute,Y	LDX Oper,Y	BE	3	
LDY						
Load index Y with memory	M → Y	Immediate	LDY #Oper	A0	2	√√----
		Zero Page	LDY Oper	A4	2	
		Zero Page,X	LDY Oper,X	B4	2	
		Absolute	LDY Oper	AC	3	
		Absolute,X	LDY Oper,X	BC	3	

Figure 6.5C-4

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
LSR						
Shift right one bit (memory or accumulator)	(See Figure 1)	Accumulator	LSR A	4A	1	0√√---
		Zero Page	LSR Oper	46	2	
		Zero Page,X	LSR Oper,X	56	2	
		Absolute	LSR Oper	4E	3	
		Absolute,X	LSR Oper,X	5E	3	
NOP						
No operation	No Operation	Implied	NOP	EA	1	------
ORA						
"OR" memory with accumulator	A ∨ M → A	Immediate	ORA #Oper	09	2	√√----
		Zero Page	ORA Oper	05	2	
		Zero Page,X	ORA Oper,X	15	2	
		Absolute	ORA Oper	0D	3	
		Absolute,X	ORA Oper,X	1D	3	
		Absolute,Y	ORA Oper,Y	19	3	
		(Indirect,X)	ORA (Oper,X)	01	2	
		(Indirect),Y	ORA (Oper),Y	11	2	
PHA						
Push accumulator on stack	A ↓	Implied	PHA	48	1	------
PHP						
Push processor status on stack	P ↓	Implied	PHP	08	1	------
PLA						
Pull accumulator from stack	A ↑	Implied	PLA	68	1	√√----
PLP						
Pull processor status from stack	P ↑	Implied	PLP	28	1	From Stack
ROL						
Rotate one bit left (memory or accumulator)	(See Figure 2)	Accumulator	ROL A	2A	1	√√√---
		Zero Page	ROL Oper	26	2	
		Zero Page,X	ROL Oper,X	36	2	
		Absolute	ROL Oper	2E	3	
		Absolute,X	ROL Oper,X	3E	3	
ROR						
Rotate one bit right (memory or accumulator)	(See Figure 3)	Accumulator	ROR A	6A	1	√√√---
		Zero Page	ROR Oper	66	2	
		Zero Page,X	ROR Oper,X	76	2	
		Absolute	ROR Oper	6E	3	
		Absolute,X	ROR Oper,X	7E	3	

Figure 6.5C-5

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg N Z C I D V
RTI Return from interrupt	P↑ PC↑	Implied	RTI	40	1	From Stack
RTS Return from subroutine	PC↑, PC+1 → PC	Implied	RTS	60	1	------
SBC Subtract memory from accumulator with borrow	A - M - $\overline{C}$ → A	Immediate	SBC #Oper	E9	2	√√√--√
		Zero Page	SBC Oper	E5	2	
		Zero Page,X	SBC Oper,X	F5	2	
		Absolute	SBC Oper	ED	3	
		Absolute,X	SBC Oper,X	FD	3	
		Absolute,Y	SBC Oper,Y	F9	3	
		(Indirect,X)	SBC (Oper,X)	E1	2	
		(Indirect),Y	SBC (Oper),Y	F1	2	
SEC Set carry flag	1 → C	Implied	SEC	38	1	--1---
SED Set decimal mode	1 → D	Implied	SED	F8	1	----1-
SEI Set interrupt disable status	1 → I	Implied	SEI	78	1	---1--
STA Store accumulator in memory	A → M	Zero Page	STA Oper	85	2	------
		Zero Page,X	STA Oper,X	95	2	
		Absolute	STA Oper	8D	3	
		Absolute,X	STA Oper,X	9D	3	
		Absolute,Y	STA Oper,Y	99	3	
		(Indirect,X)	STA (Oper,X)	81	2	
		(indirect),Y	STA (Oper),Y	91	2	
STX Store index X in memory	X → M	Zero Page	STX Oper	86	2	------
		Zero Page,Y	STX Oper,Y	96	2	
		Absolute	STX Oper	8E	3	
STY Store index Y in memory	Y → M	Zero Page	STY Oper	84	2	------
		Zero Page,X	STY Oper,X	94	2	
		Absolute	STY Oper	8C	3	
TAX Transfer accumulator to index X	A → X	Implied	TAX	AA	1	√√----
TAY Transfer accumulator to index Y	A → Y	Implied	TAY	A8	1	√√----
TSX Transfer stack pointer to index X	S → X	Implied	TSX	BA	1	√√----
TXA Transfer index X to accumulator	X → A	Implied	TXA	8A	1	√√----
TXS Transfer index X to stack pointer	X → S	Implied	TXS	9A	1	------
TYA Transfer index Y to accumulator	Y → A	Implied	TYA	98	1	√√----

Figure 6.5C-6

THE FOLLOWING NOTATION APPLIES TO THIS SUMMARY:

A	Accumulator
X, Y	Index Registers
M	Memory
$\overline{C}$	Borrow
P	Processor Status Register
S	Stack Pointer
√	Change
—	No Change
+	Add
∧	Logical AND
-	Subtract
∀	Logical Exclusive Or
↑	Transfer From Stack
↓	Transfer To Stack
→	Transfer To
←	Transfer To
V	Logical OR
PC	Program Counter
PCH	Program Counter High
PCL	Program Counter Low
OPER	Operand
#	Immediate Addressing Mode

FIGURE 1. ASL-SHIFT LEFT ONE BIT OPERATION

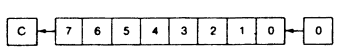


FIGURE 2. ROTATE ONE BIT LEFT (MEMORY OR ACCUMULATOR)

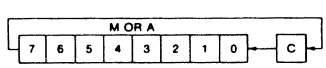
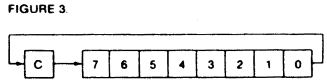


FIGURE 3.



NOTE 1: BIT — TEST BITS

Bit 6 and 7 are transferred to the status register. If the result of A ∧ M is zero then Z=1, otherwise Z=0.

Editor's Note: Previous figures in Chapter 6 have been reprinted from the Apple II Reference Manual, *with permission of Apple Computer, Inc.*

6.6 *Data Handling Instructions Equivalent to BASIC PEEKs and POKEs*

6.6.1 Load Accumulator Instruction — LDA The Machine-Language Equivalent to a 'PEEK'

A particularly important example of machine-language instructions is LDA — LoaD A-register (accumulator). The A-register or the accumulator is the register which is most frequently loaded with data in order to interface with a machine-language routine. LDA is the hardware-implemented instruction that moves information from any memory location into the A-register, the register used for most manipulative and arithmetic operations.

6.6.2 Store Accumulator Instruction — STA The Machine-Language Equivalent to a 'POKE'

The machine-language equivalent of a POKE is the inverse command: STA (STore A-register), the command which moves information from the A-register back to memory. The POKE command is little more than a STA Instruction in disguise.

6.6.3 An Example of Data Movement Using PEEKs and POKEs and its Machine-Language Equivalent Using LDA and STA

The Apple II System has a text video display buffer which occupies memory locations $0400-$07FF, i.e. memory locations 1024 through 2047. It also has a secondary text video display buffer which occupies memory locations $0800-$0BFF.

If you wanted to move the entire contents of the primary page to the secondary page, you could write a simple BASIC program consisting of PEEKs and POKEs. A simple straight-line-code version of this program is shown as

Figure 6.6A
Straight-Line BASIC Program to Move Data from Primary to Secondary Text Page using only PEEKS and POKES

```
Line  Instruction        Comment
100   A = PEEK(1024)     REM LoaD A with contents of location 1024 ($0400)
101   POKE 2048,A        REM STore A into location 2048 ($0800)
102   A = PEEK(1025)     REM LoaD A with contents of location 1025 ($0401)
103   POKE 2049,A        REM STore A into location 2049 ($0801)
104   A = PEEK(1026)     REM LoaD A with contents of location 1026 ($0402)
105   POKE 2050,A        REM STore A into location 2050 ($0802)
...
...
...

2148  A = PEEK(2047)     REM LoaD A with contents of location 2047 ($07FF)
2149  POKE 3195,A        REM STore A into location 3195 ($07FF)
```

With just these two instructions (each repeated 1024 times) you could write a computer program to move (copy) the entire contents of video text page 1 to video text page 2:

Figure 6.6B
Machine-Language Program using LDA & STA Equivalent to Straight-Line BASIC Program to Move Data from Primary to Secondary Text Page using only PEEKS and POKES

```
Mem  M/L       Short     Long
Loc  Instr     Comment   Comment
-------------------------------------------------------------------
6000 AD 00 04  LDA $0400 LoaD A with Page1, Location 0-$0400 or dec 1024
6001 8D 00 08  STA $0800 STore A into Page 2, Location 0-$0800 or dec 2048
6002 AD 01 04  LDA $0401 LoaD A with Page 1, Location 1-$0401 or dec 1025
6003 8D 01 08  STA $0801 STore A into Page 2, Location 1-$0801 or dec 2049
6004 AD 02 04  LDA $0402 LoaD A with P1+2* - $0402 or decimal 1026
6005 8D 02 08  STA $0802 STore A into P2+2 - $0802 or decimal 2050
...                      * Page x, Location y will be hereafter
...                        specified as base address Px + page
...                        location index y
6FFE AD FF 07  LDA $07FF LoaD A with P1+3FF (hex) or P1+1023 (dec) - loc
                         $07FF (hex)or  2047 (dec)
6FFF 8D FF 0B  STA $0BFF STore A into P2+3FF (hex) or P2+1023 (dec) - loc
                         $0BFF (hex) or 3195 (decimal)
.....................:.....................................................
NOTE: Machine language instructions are shown in the format they actually go
into the computer.
The instructions are actually 3 bytes long with the first being the
operation code, the second being the low-order byte of the address, and the
the last being the high-order byte of the address.
On paper the address seems to be backwards, but this is the way the computer
actually stores the information.
```

Similar instructions exist for moving information to and from the X-register (LDX and STX) and to and from the Y-register (LDY and STY), the next most frequently used registers for passing information between programs or subroutines. These three pairs of data movement instructions, plus three sequence control instructions described in the next section, may be all that many BASIC-oriented programmers ever need to interface quite freely with a broad spectrum of registered-oriented machine language programs.

Techniques for using a user-written machine-language program, even for interface with machine-language code permanently imbedded in the Apple system, are not introduced until Chapter 8. When they are introduced and case studies are provided, the instructions mentioned here more than suffice to meet almost all routine needs for moving information.

6.7
Symbolic Instructions and Programming Machine-Language in Symbolic Assembler Format

Although hexadecimal abbreviations are much easier to use than the binary bits they represent, writing machine-language instructions in hexadecimal form is still not very convenient. Not only do they not have any mnemonic (memory-aiding) characteristics to help you in remembering what a particular instruction means, but almost any time you make a change in a program, perhaps by adding or deleting an instruction, many of the memory locations and addresses used may change, thus causing major housekeeping or bookkeeping changes to get the program back into running order.

It is very clear that there is a one-to-one correspondence between the names of instructions (or their three-letter short-form abbreviations) and the binary/hexadecimal codes used inside the computer to represent the specific operation.

Thus it should be perfectly feasible to write the operation codes for instructions in the mnemonic (memory-aiding) form given in the table of operation codes (Figure 6.5A) and later, do a rote table look-up to get the hexadecimal form of the operation code to enter into computer memory. This should make both the writing of instructions and the later process of reading them much more convenient.

It should be equally feasible to assign names to addresses as well. However, the names for addresses need not be permanently assigned. It would be much more convenient to assign them temporarily for each problem in such a fashion that memory addresses become easily remembered names for the parameters of the specific problem being solved. These assignments also may be put into a table so that when writing or reading the program the programmer can use the convenient symbolic name rather than a hard-to-remember numeric address.

Thus the programmer can write his code in a symbolic form no different from the short-form symbolic abbreviation of the comments which we have been using, particularly in Figure 6.6B. Later he can look up the hexadecimal equivalents of the symbols used and feed those symbols to the computer.

Figure 6.7A shows the same instructions written in symbolic form as a program to be translated in such a fashion. Note that only the leftmost column of this figure contains the symbolic instructions to be translated. The other columns contain the results of translation and explanatory remarks or comments.

The three instructions which form a declarations section preceding this code turn out to be instructions for controlling the translation processing. How and why they are used will be discussed in some detail later.

In the early days of computing, translation/look-up was done by hand. Now the task of assembling the actual binary/hexadecimal bytes of

information to go into the computer from instructions written in symbolic form is done by computer programs. It should come as no surprise that these programs are usually called 'symbolic assemblers' or 'assemblers'.

Figure 6.7A
Symbolic (Assembly-Language) Expression
of Machine Language Program
to Move Text Page 1 to Text Page 2
(Straight-Line Form)
(with Addressing Relative to Beginning of Page)

```
<DECLARATIONS (pseudo-instructions) TO ASSEMBLER - NO CODE CREATED>

Label  Symbolic Form  Machine-Language        Remarks
       of Instruction Locn   Code-Generated
.....................................................................
P1     EQU 400              <none>   Makes name P1 equivalent to add $400
P2     EQU 800              <none>   Makes name P2 equivalent to add $800
       ORG 6000             <none>   Makes program origin occur at $6000
.....................................................................
<ASSEMBLY-LANGUAGE INSTRUCTIONS TO BE TRANSLATED TO MACHINE-LANGUAGE>

Label  Symbolic Form   Machine-Language        Remarks
       of Instruction  Locn   Code-Generated
.....................................................................
       LDA P1+0        <6000>  AD 00 04 LOAD A-Register from Page1($400)
       STA P2+0        <6003>  8D 00 08 Store A-Register to Page 2($800)
       LDA P1+1        <6006>  AD 01 04
       STA P2+1        <6009>  8D 01 08
       LDA P1+2        <600C>  AD 02 04
       STA P2+1        <600F>  8D 02 08
       ...             ...
       ...             ...
       ...             ...
       LDA P1+3FF      <77FE>  AD FF 07 Last of $400 locations on Page 1
       STA P2+3FF      <77FF>  8D FF 0B Last of $400 locations on Page 2

(NOTE: <xxxx> indicates where instruction created by the assembler is to
       go in memory. It is not part of the instruction code generated by
       the assembler)
(NOTE: The machine-language code bytes are shown in the order they go into
       the computer, that, is the low-order byte first, then high-order.
       Thus address $0400 is shown as 00 04.
```

Perhaps the greatest advantage of using a symbolic assembler is that it allows you to concentrate more on the problem to be solved and less on the details of how the machine functioned in solving the problems.

When an assembler is used you usually go through a two-phase process to get a program ready to run. First you write the program in symbolic form and use this 'source' version as input to a processing procedure carried out by the computer using the assembler as a translation program and generating as output a binary/hexadecimal version of the program. In phase II this program is loaded into the computer and used with data input to get the desired program results.

At an early stage in the development of modern computing, programmers found that in addition to using symbolic form for the instructions built into the system, it was convenient to have other short-form symbolic instructions, not to be translated into machine language, but to instruct the assembler whenever the programmer wanted the translation to be done in a particular way. For example, the programmer might want to specify that the variable name 'x' was assigned to a particular memory location, the variable name 'y' to another, and so on.

While such assignments may be made just by whim there is often a reason to want to use a particular location. Sometimes the location may be used in a special way by the computer hardware. For example, there are specific memory locations associated with interfacing the computer to specific input-output devices and peripherals that determine whether output is going to be text or graphics, etc. Of course the programmer may also want to interface the new assembly-language program with one previously written that used particular memory locations for specific things which he or she wants to share with the program now being written.

For example, in the sample problem we were doing in Figure 6.7A, it would be quite reasonable to specify that a name such as 'PAGE1' or 'P1' be used as a synonym for the beginning of text page 1 (a location predetermined by the Apple II system hardware and monitor as location $0400 (decimal 1024). Similarly, 'PAGE2' or 'P2' might be convenient to use as a name equivalent to the beginning of text page 2, $0800 (decimal 2048).

To implement such a requirement a programmer would have to notify the assembler in advance not to assign the variable names used, say 'p1' and 'p2', to the first conveniently available location, but instead to use those names as equivalent to memory locations $0400 and $0800 respectively.

Most assemblers use the pseudo-operation code 'EQU' (for equivalent or equivalence) to specify such assignments.

Now that the program uses symbolic addressing it has considerably more generality. 'START' 'PAGE1' AND 'PAGE2' need not always refer to their values in figure 6.7A ($6000, $0400 and $0800 respectively)—they can be defined at the time the program is assembled into machine language form rather than at the time the program is written. In fact, if you use an assembler and loader that produce relocatable code, their final locations may not be firmly fixed until the program is run.

If you get confused as to where instructions are located, and the program counter tells the machine to FETCH its next instruction from a location that really doesn't contain an instruction, the results can be highly unpredictable. When the contents of the memory byte are FETCHed, the instruction-decoder circuitry will examine its binary bit-pattern and treat that pattern as if the data were an instruction. It will set the computer up to do the specified operation and then EXECUTE it.

The results can be catastrophic to the integrity of the remainder of the program and data in the machine, because this instruction may send the

computer off on a long wild-goose chase. Again and again it would FETCH and EXECUTE random data-bit combinations as instructions until it finally comes to an instruction it recognizes as a signal to halt or an instruction it doesn't recognize, whereupon the system will 'hang'. By this time the previous contents of memory may have been reduced to rubble.

Programmers also often need to specify where the program is to be put into the computer's memory after translation. Usually this is done by specifying the location of the origin (the beginning of the code). Not surprisingly, the pseudo-operation-code 'ORG' is commonly used for this specification.

Different assemblers support different pseudo-operations. Almost all provide some means of reserving a block of memory for a table or an array (the equivalent of a BASIC DIMension statement), and a means of assigning numeric constants and character strings.

It is good programming practice to make such specifications as declarations at the beginning of a program. This is convenient both for the programmer who wants to write well-structured and well-documented code, and for the assembler, the program that does the translation that uses this information.

Some assemblers require advanced delcaration of all variables, others do not. Of those that require advanced declarations, some do it as a matter of doctrine — as a means of forcing the programmer toward the use of structured programming techniques. Others do it to avoid running into a specification that a memory location be used for a particular purpose after it has already been used for another.

In machine-language programs written in symbolic form to be translated by an assembler, the preliminary declaration pseudo-operations are usually followed by the actual computer instructions (in symbolic form) as illustrated in figure 6.7A.

Some assemblers will accept not only those instructions that are built into the hardware of the computer, but additional instructions. Sometimes these additional instructions are pre-defined. For example, they might be instructions for implementing floating point arithmetic capabilities not built into the hardware of the computer. Other assemblers will allow the programmer to declare and define his own pseudo-instructions of this type.

In either case, such pseudo operations are implemented by subroutines or short blocks of code that are inserted in the computer-generated machine-language output of the assembler in much the same way as a single, normal, hardware-implemented instruction might be.

Since a single one of these instructions might cause two or ten or a hundred machine-level instructions to be performed, their effect can be that of a macro-instruction, which does tasks that would otherwise require many single, individual hardware instructions.

Such instructions and their operation codes are called macro-instructions, or macros for short. Assemblers that have such built-in capabilities are often called macro-assemblers.

Now let's take another look at our program to move the contents of Text Page 1 to Text Page 2 written in symbolic form suitable for conversion to machine-language binary/hexadecimal form by an assembler. It is shown as figure 6.7A. Note again that only the leftmost column of this figure contains the symbolic instructions to be translated. The other columns contain the results of translation and explanatory remarks or comments.

This program has two very desirable properties: 1. It is super-fast — many, many times faster than the fastest BASIC program you could ever write to do the same task, and 2. It is simple, straightforward, and easy to understand.

This program also has a good deal more generality than either the BASIC or the non-symbolic machine-language version. A change in a single instruction 'p1 equ $400' or 'p2 equ $800' can change what block of $400 memory locations is moved or its destination. A change in the 'org' statment can be used to relocate the program itself to a different area of memory.

Although it has these desirable properties, this program is useless for practical purposes. It takes two instructions of three bytes length each to move each of the 1024 bytes of information a total of 6144 bytes of program. Not only is this a huge amount of memory, but the amount of labor involved in preparing and entering all these instructions would be prohibitive in all but the most unusual circumstances.

Moreover, while this program is more general than its predecessors, it is not yet a general purpose data mover capable of moving an arbitrarily sized block of information from any desired location in memory to another. Such programs do exist, take up only a tiny part of as much memory as this one and are easy to write. In fact, one is permanently imbedded in the Apple II and Apple II+ systems as part of the system monitor. Where?

Look it up in the 'Gazetteer' under the name of 'MOVE'.

Clearly there must be, and of course there are, other computer machine-language programming techniques and instructions that can take advantage of the high degree of near-repetition (1024 LDA-STA pairs, each of which differs from its nearest neighbor by single location in each of its instructions).

To avoid using such horrendous amounts of memory you must be able to re-use the same instructions over and over again. This means an ability to change the sequence of instructions from what we have used thus far, each one immediately following its predecessor. BASIC does this with 'GOTO', 'IF...THEN' and 'FOR...NEXT' statements.

Actually, such sequence-changing instructions are not quite enough in themselves. The instruction should not be exactly the same each time it is used, or it would do exactly the same thing and accomplish nothing new.

In addition, for a really useful, general-purpose program, you should also be able to set the number of repetitions to any desired value so that you can move as small or as large a block of memory as you desire.

Let's dig into the additional features of machine language that give us these capabilities.

6.8 Instructions Which Change the Normal Sequence of Operation (The Key to Repetition and the Computer's Decision-Making Capability)

There is a very special group of instructions in the repertoire of the Apple II (or any other modern computer) which, when executed, may change the location from the next number in normal sequence to an entirely different number. Such instructions are usually called Jump, Transfer or Branch instructions.

These instructions, known collectively as control or sequence control instructions, let the computer repeat sequences of instructions and make decisions.

The instructions have the same effect at the computer hardware level that the corresponding control instructions have in BASIC, except they refer to hardware locations, not BASIC line numbers. For example,

JMP xxxx (JuMP to xxxx) has the same effect as the BASIC GOTO statement;
JSR (Jump to SubRoutine) has the same effect as the BASIC GOSUB statement; and GOSUB statement; and
RTS (ReTurn from Subroutine) has the same effect as the BASIC RETURN statement.

Hardware sequence-changing instructions, like BASIC IF statements, may cause a change in sequence only if a particular condition is met (or not met). Typical instructions are:

BPL (Branch on result PLus)
BMI (Branch on result MInus)
BNE (Branch if result Not Equal to zero)

If the condition for branching is not met, no change is made in the program counter's contents — except for the change automatically made with every FETCH, setting it to the next instruction. Thus the program continues on to the next instruction in normal order. If the condition for the branch is made, the program counter is advanced the number of locations specified by the second byte of the instruction.

Chapter VII
Apple Architecture II: Addressing in the Apple II Microprocessor

7.1
Addressing Modes of the Microprocessor in the Apple II System

The hardware-implemented instructions in the Apple II can identify the location of the data which they are to use in many different ways. Figure 7.1A summarizes them and gives an example of each:

Figure 7.1A
Machine-Language Addressing Modes Available in the Apple II System

Addressing Mode	Example	Machine Code	Micro-seconds Required to Execute Instruction
Implicit/Implied	TYA	98	2
Immediate	LDA#$A0	A9 A0	2
Absolute	LDA $7FA	ADFA07	4
Zero Page	LDA $80	A4 80	3
Indexed absolute	LDA 7FA,X	BDFA07	4 (5 if page boundary crossed)
Indexed zero pg	LDA $80,X	B5 80	4
Indirect Indexed	LDA (80,X)	A1 80	6
Indexed Indirect	LDA ($80),Y	B1 80	5 (6 if page boundary crossed)
Relative	BCC $3360	90 0F	2 if no branch occurs 3 if branch to same page; 4 if different

Most, but not all, of these addressing modes were illustrated with the LDA (Load Accumulator) instruction — a common and quite representative instruction. No single instruction has all of the addressing modes. Many have only one.

Some readers who may be particularly astute or who have prior hardware-level experience may be surprised to notice that an instruction different from the LDA was used to illustrate hardware-implemented relative addressing. Does that mean that, in spite of the fact that we have used a form of relative addressing for the LDA in our previous examples, this capability is not supported by the hardware? Yes and no. Relative addressing in the MOS 6502 microprocessor used in the Apple is limited to addressing to locations relative to the instruction being executed. This is not the kind of relative address we were using in our symbolic (assembly-language) code — we were using addresses relative to the start of an array or some table of data, specifically relative to the start of an array of bytes which determined what was to be displayed as output on the Apple's screen.

Assemblers usually let you write instructions in a relative-address form — whether it be relative to the current instruction (usually symbolized by '*') or any other predetermined base address. Then the assembler makes the conversion (as part of its translation process) to whatever addressing mode is convenient for use by the machine. As we shall see later the indexed form of the hardware instruction provides the capability that we need to implement such a concept neatly, conveniently and efficiently. (The assembler we use may not use this fact and may just compute an absolute address unless we specifically tell it to use indexing.)

In the 6502 microprocessor used by the Apple II true hardware-implemented relative addressing is limited to sequence-changing instructions.

Some of you may have noticed that for each of the 7 addressing modes used for LDA in the above table there is a different hexadecimal representation for the LDA operation code. Were you to look at all the operations codes in the Apple II/MOS 6502 microprocessor on a bit-by-bit basis rather than as hexadecimal characters you might note that individual bits in the operation codes tend to specify address mode while others tend to specify the overall operation to be performed (e.g. LDA, STA, etc.). This kind of design is common to most computers and microcomputers.

In some systems it is conventional to document such a design by means of a short operation code with separately specified address-mode modifiers. In the MOS 6502/Apple system the designers thought it less confusing to use full-byte operation codes and to treat the different address-mode versions as differently coded versions of the operation code.

7.2
Simple Addressing Modes of the Microprocessor in the Apple II System

Simple addressing modes are those which do not involve using *computed* addresses.

7.2.1 Implied Addressing (No Address Required)

Some instructions don't require an address to specify what they must do. Examples:

CLD — CLear Decimal mode
CLI — CLear Interrupt disable bit
CLV — CLear oVerflow flag
DEX — DEcrement X-register (index register X) by 1

Instructions that use implicit addressing are only a single byte long, so the Program Counter always advances by one for such instructions.

7.2.2 Immediate Addressing (Data Value Included in Instruction)

Many instructions have an immediate address option that allows a data value, which immediately follows the operation code, to be used as the value on which the instruction operates. The address, implicit rather than explicit (stated in numbers), is the location immediately after the operation code. The MOS 6502 restricts immediate addressed data to a single byte in length and hence to data values in the range 0-255. Instructions which use immediate addressing are two bytes long, so the Program Counter always advances by 2 for such instructions.

7.2.3 Absolute Addressing (16-Bit Address Specifies Absolute Location of Data)

This is the classic von Neumann method for specifying how to find the data to be used with a particular instruction. Each memory location in the computer is permanently assigned an identification number — an absolute address. The Apple II permits direct addressing of 2^{16} or 65536 memory locations, so addresses for absolute locations must be 16 bits or two bytes long. The absolute address of the memory location where the data may be found immediately follows the operation code, so absolute-addressed instructions are three bytes long and the program counter always advances by three for the next instruction.

7.2.4 Zero Page Addressing (8-Bit Absolute Addressing)

The Apple II memory of 65,536 bytes is divided into 256 pages of 256 bytes each. A single-byte address can address $2^8 = 256$ locations, so a two-byte address can be thought of as using one byte to specify the memory page and the second to specify the memory location within the page. In this addressing mode the page of memory to be used is always page zero, so the byte to specify the page is not needed. Instructions using the zero-page addressing technique are always two bytes long and the program counter always advances by two for the next instruction.

7.2.5 Relative Addressing

Whereas absolute addressing specifies permanently-assigned and unchanging addresses, relative addressing specifies them relative to the instruction in which they appear. Thus if an instruction at memory location $0300 specifies a relative address of $10, the effective address is at location $310 (=$300+$10). If the same instruction were located at $350, then the effective address would be $360 (=$350+$10).

Many assemblers allow users to write programs in a relative-addressed form, whether or not the computer being used has hardware-implemented relative addressing.

Usually some convention, such as using '*' for the current instruction, is adopted so that an instruction with an address of * + $10 would refer to the memory location $10 positions beyond the location of the instruction in which it was being used. Assemblers usually allow specifying locations relative not only to the current instruction but to any named address.

Many computers offer hardware-implemented relative addressing, either for all types of instructions or just for a particular class of instructions such as the sequence changing instructions.

Historically there have even been computers that offered no absolute addressing whatsoever, only relative addressing.

The MOS6502 used in the Apple II offers hardware-implemented relative addressing only for the 'branching' instructions: BCC, BCS, BEQ, BMI, BNE, BPL, BVC, and BVS. Moreover, it offers no other addressing mode for these instructions. We have already covered these instructions and I am sure that no reminder is needed that the binary/hex code they generate contains only a plus or minus displacement of the program counter from its original value if the condition for branching is met.

All instructions that use the relative addressing mode occupy exactly two bytes of memory and take two microseconds to execute if the branching condition is not met, three if it is met to a location on the same page, and four if it is met to a location which requires crossing a page boundry.

It is interesting to note that the availability of relative addressing in an assembler does not guarantee that there is a corresponding relative-addressing mode in the hardware. Nor does the use of non-relative addressing by assembler instructions guarantee that the assembler will not use relative addressing. For example, we used a form of relative addressing for our LDA and STA instructions in our earlier example of the text-page moving program even though the 6502/Apple II processor does not provide a relative addressing hardware mode for such instructions. (The actual machine-code produced used absolute addressing.)

7.3 *Overview of Computed Address Concepts*

7.3.1 The Key to Understanding Computed Addresses

There is a significant difference between addressing covered in the previous sections and those covered in later sections. In each of the previous cases, the location specified by the instruction was fixed and known when the program was written. In the cases covered by this section,the location specified in the instruction is variable; the programmer knows how to compute it, but doesn't necessarily know its current value.

The techniques of computed addressing are useful and particularly important if you want to read and understand the code written by systems programmers and imbedded in the firmware of the Apple II system.

There are basically three types of computed addressing. Each will be briefly characterized here, then the two which are widely used in the Apple II firmware will be developed more thoroughly.

7.3.2 Computing Addresses By Treating Them As Data

Instructions are represented in computer memory by bit patterns. At the hardware level in most computers there is no hard-and-fast rule that they can only be accessed during FETCHing. Computer instructions can manipulate and perform arithmetic on these bit patterns just as if they were data. Thus it is perfectly possible to add or multiply or do other arithmetic operations to the address portion (or even the operation code) of any instruction in any fashion specified by the programmer. Not only is it possible but it is done, although it was done much more frequently twenty years ago than it is today. This process is out of favor today because errors are very easy to make and often catastrophic in impact and almost anything that a programmer can do in this fashion can be done more easily by indexing or indirect addressing.

7.3.3 Computing Addresses By Hardware Indexing

This method of addressing uses a basic address (a fixed address such as an absolute or zero-page address) which is automatically modified by the hardware of the machine into a different address. The modification normally consists of adding to the address before it is used in execution. Both the X-register and the Y-register in the Apple II are equipped to act as index registers.

Indexing is closely akin to subscripting in BASIC; the subscript of a variable in BASIC is sometimes described as the array-index. An array is a block of memory locations given the same name, the equivalent of the same base-address at the hardware level. The size of an array is determined by the DIMension statement. The location of a particular subscripted variable is determined by the subscript that tells you how many elements to skip down within the array. The name of the array may be considered as the base-address and the value of the subscript as the index.

7.3.4 Computing Addresses By Indirect Indexing Techinques

Indirect addressing bears the same relationship to absolute addressing as absolute addressing does to implied addressing. Whereas implied addressing gives you the value to use directly without pointing to a memory location that holds it, absolute addressing points to where the data is by specifying the address of the memory location where it is held. Indirect addressing points to a memory location too — but not the one that holds the data. Instead it points to a memory location that in turn contains an address which points to the data. Thus indirect addressing points indirectly (i.e. through an intermediate address) to the data.

The advantages of indirect addressing are very real and very significant in certain programming situtations, but it is difficult for the person who has not done much advanced machine-level programming to see why or how. Indirect addressing is used most frequently in situtations where the address you must use is not know at the time the program is written, but must be calculated by the program during its operation.

Such situations often occur in the inner workings and hidden mechanisms of systems software, such as the BASIC interpreters and Disk Operating System. For example, the BASIC interpreter must frequently refer to the location of variables such as X, Y, and Z, which the user has defined in the BASIC program. However at the time the interpreter was written, the programmer could not have possibly predicted which variables the user would use in which order. Instead, the programmer could only establish a pattern for their assignment that could be maintained in a table used to look up (or compute) the relevant addresses.

Such situations are conveniently handled by indirect addressing and by indexed indirect addressing.

7.4 Elementary Indexing

7.4.1 Indexed Absolute Addressing and Indexed Zero-Page Addressing

Absolute addresses and zero-page absolute addresses may be automatically modified at the time of execution by using an indexed form of the absolute address. This modification involves executing the instruction with an effective address, which is equal to the sum of the absolute address and the contents of the index register specified.

For most instructions, neither additional memory space nor additional execution time occurs when one uses indexed absolute rather than absolute addressing.

Instructions that use zero-page indexed addressing rather than ordinary zero-page addressing require no more memory, but one microsecond additional execution time.

For example consider the instruction

```
LDA $400, X
```

If the X-register currently holds the number $5, it has the identical effect as LDA $405. If the X-register contains $10, it has the same effect as LDA $410.

The instruction LDA P1,Z has as its base address the absolute (symbolic) address 'P1.' This address is indexed by whatever number is in the X-register.

The effect is much the same as the assembly-language addressing technique $P1 + X, where X is a pre-defined variable. The effect is also quite similar to using the subscripted variable P1(X) in BASIC. In both cases P1 can be thought of as the start of an array (or block of data or table) and X as a particular position within that array (or block of data or table).

Since the index registers in the AppleII/MOS 6502 microprocessor are only 8 bits (one byte) long, the range of address modification by the index register is only 0 to 255.

If we wanted to use indexing to shorten our data movement program of figure 6.7A, it would be convenient to arrange for the pair of instructions

```
LDA P1,X
STA P2,X
```

to be placed in a loop that repeats itself exactly $400 (1024 decimal) times. The value of the X register starts at zero and increments by one each time the loop is traversed until it reaches $3FF (or start at $3FF and decrement one each time the loop is traversed). Then we would have a very short program that does what our very long program did.

Unfortunately the X-register can count only $100 (decimal 256) because it is only a single byte long. However, by using four LDA/STA pairs within the loop, e.g.

```
LDA P1,X
STA P2,X
LDA P1 + $100,X
STA P2 + $100,X
LDA P1 + $200,X
STA P2 + $200,X
LDA P1 + $300,X
STA P2 + $300,X
```

you can transfer the whole text display screen as we did in the previous program.

One can also get the same result by putting the loop traversed 256 times inside another loop traversed four times. But you must provide some means of address modification between successive traverses of the loop.

A particularly convenient means of doing the required address modification can be implemented if you use the indirect indexed addressing technique rather than the indexed absolute addressng technique discussed here. That technique will be discussed and illustrated later.

Let's be satisfied for now with moving only one memory page, or ¼ of the text display page. We can set up the proper environment by using some of the machine-language instructions that manipulate and test the status of the X register.

Index registers (the X-register and Y-register) are counters. They can be set to any value within the range of their 8-bit capacity ($000-$FF or 0-255 (dec)). They can be incremented by 1 or decremented by 1, and the value can be tested against the value of the number in any memory location.

The following examples are basic manipulative commands for the X-register:

```
LDX #$FF    uses the immediate addressing
            mode to put the number $FF into
            the X-register.
INX         is an implied address instruction
            that increases the contents of the
            X-register by 1.  When it is ex-
```

ecuted the 'z' bit in the 'P' or status register is set according to whether X-register does or does not contain zero.

DEX — is an implied address instruction that decreases the contents of the X-register by 1.

CPX #$FF — uses the immediate addressing mode to compare the contents of memory (in the immediate addressing mode that of the next two locations as explicitly named in the instruction) with the contents of the X-register.

However, the real value of indrect addressing is not in just having the address in a fixed location where you can always find it and access it indirectly, but in having the capability of having the address automatically modified as well.

Indeed the designers of the MOS 6502 microprocessor used in the Apple II didn't even bother to implement simple indirect addressing as described above. They implemented it only with built-in modification capability. In fact they provided two different modes that differ in what is modified (by indexing) and what is not:

Indirect Indexed Addressing Mode

In indirect indexed mode, the zero-page address is used without modification, but the address to which it points, the indirectly-obtained address, is indexed. A curious, and occasionally annoying, restriction in the Apple II/MOS 6502 microprocessor is that only the Y-register can be used for indirect indexed addressing.

Indexed Indirect Addressing Mode

In the indexed indirect mode the zero-page address (which points to the indirect address) is indexed but the indirectly-obtained address to which it points is not. In the Apple II/MOS 6502 microprocessor only the X-register can be used for indexed indirect addressing. This too can occasionally be annoying, but the combination of the two restrictions does help you from using one of these two modes when you think you are using the other.

You may find it interesting to scan through the page zero memory locations as documented in the Programmers' Atlas portion of this book to see how many of them are allocated to use as pointers for use by indirectly addressed instructions.

7.4.2 Mini Case Study: Using Elementary Indexing Techniques for Moving Data

Using these new indexing instructions and a branching instruction (BNE), we can now rewrite program 6.7A. Using the old straight-ling technique to move $100 (256 decimal) bytes from text display page 1 to text display page 2 would require 1536 bytes of memory for the program and two milliseconds of execution time. Using the index incrementing method shown in figure 7.4B only 13 bytes of memory but 3.8 milliseconds of execution time are required. With the improved decrementing method, only 11 bytes of memory and 3.3 milliseconds of execution time are required.

Notice the nature of the trade-off, one which is quite typical in machine-language programming. Straight-line coding is the fastest, but wastes memory. Looping is slower (because of the time required to execute the looping instructions), but it can save a great deal of memory.

Each of these new programs will move arbitrary bytes of data in a fashion equivalent to one of the two BASIC programs shown as figure 7.4A.

Figure 7.4A
BASIC Program Using Looping Rather Than Straight-Line Technique to Move 256 Bytes of Text Screen Display Page 1 to Page 2

```
Incrementing Version          or    Decrementing Version
 98 LET P1=1024                      98 LET P1=1024
 99 LET P2=2048                      99 LET P2=2048
100 FOR x = 0 to 255                100 FOR x=255 to 0 step-1
110 LET a = PEEK(P1+x)              110 LET a=PEEK (P1+x)
120 POKE P2+x,a                     120 POKE P2+x,a
130 next x                          130 next x
140 <next program statement>        140 <next program state-
                                        ment>.
```

Figure 7.4B

Assembly-Language Program Using Looping & Indexing Techniques to Move 256 Bytes from Screen Display Text Page 1 to Text Page 2 (Incrementing Version)
(NOTE: The quantity used in the CPX instruction controls how many bytes are moved. Maximum = 256)

```
Symbolic Form      Hex Form          Remarks
P1    EQU  $400    <none>            Name P1 equivalent to address $400
P2    EQU  $800    <none>            Name P2 equivalent to address $800
      ORG  $6000   <none>            Set start of program at $6000
START LDX  #$0     6000:A2 00        Load the X-register with zero
LOOP  LDA  P1,X    6002:BD 00 04     Put contents of memory location
                                     $400+(X) into the A-register.
                                     ($400 first time through;
                                      $4FF last time through)
      STA  P2,X    6005:9D 00 08     Store contents of A-register in
                                     memory location $800+(X) ($800
                                     first time through; $8FF last)
      INX          6008:E8           Increase contents of X-register by
                                     1 and set status register to test
                                     for zero value
      CPX  #$FF    6009:E0 FF        Compare contents of X-register
                                     with the value $FF. Set 'Z' flag
                                     in P-register
      BNE LOOP     600B:D0 F5        If Z-flag<>0 then branch backward
                                     to current location -11 (=$F5 in
                                     two's complement notation)
      <next program statement>
```

Figure 7.4C

Assembly-Language Program Using Looping & Indexing Techniques to Move 256 Bytes from Screen Display Text Page 1 to Text Page 2 (Decrementing Version)

Symbolic Form		Hex Form	Remarks
P1	EQU $400	<none>	Name P1 equivalent to address $400
P2	EQU $800	<none>	Name P2 equivalent to address $800
	ORG $6000	<none>	Set start of program at $6000
START	LDX #$FF	6000:A2 FF	Load the X-register with the value $FF (255 decimal)
LOOP	LDA P1,X	6002:BD 00 04	Put contents of memory location $400+(X) into the A-register. ($4FF first time through; $400 last time through)
	STA P2,X	6005:9D 00 08	Store contents of A-register in memory location $800+(X) ($8FF first time through; $800 last)
	DEX	6008:CA	Decrease contents of X-register by 1 and set status register to test for zero value
	BNE LOOP	6009:D0 F7	If Z-flag<>0 then branch backward to current location -9 (=$F7 in two's complement notation)
	<next program statement>		

Figure 7.4D

SUBROUTINE THAT USES INDEXING TO LOOK UP AND PRINT OUT ASCII DATA FROM A TABLE
This subroutine uses monitor subroutine COUT to print ASCII (text) characters in table 'DATA.'

The number of characters is specified by the immediate address (data value) field of the CPX instruction.

A carriage return is added at the end of the printout, so the next printout will start a new line.

```
Symbolic Form     Hex Form      Remarks
*** DECLARATIONS
COUT  EQU $FDED   <none>        Use standard Apple name for monitor character
                                output routine from monitor located at $FDED
      ORG $300    <none>        Set Program Counter for program to begin at $300
*** LOOP INITIALIZATION
START LDX #$00    300: A2 00    Initialize index register to zero
*** MAIN PROGRAM LOOP
LOOP  LDA DATA,X  302: BD 13 03 LoaD into A-reg. the Xth byte from table DATA
      JSR COUT    305: 20 ED FD Jump to SubRoutine COUT to print byte in A-reg
      INX         308: E8       INcrement X-reg. to next byte in table
      CPX #$05    309: E0 05    ComPare X-reg. with the number of chars. to be
                                printed. 5 (immediate address value '$05')
      BCC LOOP    30B: 90 F5    Branch back to LOOP to get & print another
                                character, unless comparison number exceeded.
                                Break loop and go to next instruction, if done.
*** END-OF-SUBROUTINE WRAP-UP
      LDA #$8D    30D: A9 8D    LoaD A-reg. with end-of-line carriage return
      JSR COUT    30F: 20 ED FD Jump to COUT to print c/r (ASCII $8D)
EXIT  RTS         312: 60       ReTurn from Subroutine to calling program
*** PSEUDO-OP TO LOAD DATA INTO MEMORY
DATA  HEX C1D0D0CCC5
                  313: C1       ASCII 'A' + $80 (high bit set for COUT)
                  314: D0       ASCII 'P' + $80
                  315: D0       ASCII 'P' + $80
                  316: CC       ASCII 'L' + $80
                  317: C5       ASCII 'E' + $80
```

Several things should be noted about these assembly-language programs:

1. Both versions, though slower than straight-line coded versions, are still lightning fast compared to their BASIC equivalents (3.8 and 3.3 milliseconds respectively).

2. Both use very little memory — only 13 bytes in the incrementing version or 11 bytes in the decrementing — much less than its BASIC-language equivalent.

3. It is hard to read in its assembly-language form and much harder yet in the machine-language hexadecimal byte form.

4. As shown above, neither program will do the job we originally intended it to do!

   Consider the decrementing version. It moves $4FF = > $8FF, $4FE = > $8FE ... $401 to $801, but stops before it can do $400 = > $800. That is, when (X) = 0, it does not branch to move $400 = > $800. This 256th move is easily obtained by doing an LDX #$00 instead of LDX #$FF. Then on the very first move $400 = > $800 but the X-register is decremented to $FF before the branch test. This is a wrap-around operation — the page does not change.

5. The programs would have to be rewritten to provide a page-changing capability before they could move more than 256 bytes of memory.

6. Within this constraint, by changing parameters P1, P2, and the value loaded to the X-register, this program is quite general, capable of moving any amount of information from anyplace to anyplace.

7. It is easy to miss the existance of this generality because the program is not easy to read without careful study.

These programs illustrate, in a microcosm, many of the advantages and disadvantages of programming at the assembly-language/machine-language level.

7.4.3 Mini Case Study: Using Indexing to Search Through and Print Data from a Table

This is an example of a program that should not be done in machine language unless it is done as part of a large assembly-language program in an environment where BASIC is not readily available.

This program is an excellent example of how assembly- or machine-language often makes you do everything yourself in excruciating detail. The whole program has the same effect as a single BASIC statement:

```
print "APPLE"
```

Indeed what is shown here is not the complete assembly-language program to do the job from scratch. Most of the actual work is done by the 'COUT' (Character OUTput) routine in the system monitor which this program calls upon to do the actual printout.

This program uses the indexed addressing mode to scan through a data table to print text. In this case the word 'APPLE' is stored in the table 'DATA.

The theory of operation is simple. After the initial declarations and initialization of the index register to zero, the program loops repeatedly, each time getting successive bytes from the data table. It finds each successive byte by using the start of the data table as its base address and the contents of the X-register as the offset to determine the effective address of the current character. At the end of the loop the program checks the current offset against the number of bytes to be printed (which is stored as the immediate access address field of the CPX instruction). The program loops back if it has not reached this end-test value. When a match occurs the program breaks out of the loop, loads and prints a carriage return and then returns control to the routine that called it.

The table is put into memory by a new type of pseudo-operation 'HEX.' there is considerable variability among assemblers in their handling of such functional requirements. Pseudo-operation 'HEX' enters the string of hexadecimal bytes that follow the pseudo-op-code. Some assemblers have in addition, or instead, an ASCII pseudo-operation (usually abbreviated ASC) that allows one to enter ASCII characters directly.

Such data can also be entered by the system monitor without use of an assembler merely by specifying the memory location, colon, the data, e.g.

313:C1 D0 D0 CC C5

7.5 *Indirect Addressing*

7.5.1 An Overview of Indirect Addressing in the Apple II

In solving many problems it is convenient to have an address that is truly a computed value, not just a base address with some type of offset, but a calculated value. Indirect addressing provides this capability.

In the Apple II/MOS 6502 microprocessor indirect addressing is always accomplished using a pointer located on the zero-page. Thus the basic form of indirect addressing is that of an instruction consisting of an OPERATION CODE followed by a ZERO-PAGE ADDRESS. The microprocessor obtains the effective address by picking up at the zero-page address the effective address of the operation. Instead of using the zero-page address directly, the indirect addressing mode uses the contents of the zero-page address to point to the address to be used.

Let's compare this to the classic absolute addressing. In absolute addressing the value in the program counter is used as the address to pick up the lower byte of the effective address. One is added to the program counter to pick up the high byte of the address. In the case of indirect addressing, the next value after the operation code, as addressed with the program counter, is used as a pointer to designate the low byte of the effective address and one is added to the pointer to pick up the high byte of the address. The computer then goes on to use this address as if it were an absolute address. Thus the zero-page address in the instruction is really the address of an address used like an absolute address.

Why go to this additional complication? When is its use worthwhile?

Indirect addressing becomes valuable when the address to be used is not known at the time the user writes the program. The indirect address is really an address that would have been coded directly — and more efficiently — as an absolute address had its numeric value been known when the program was written.

For example, to minimize the coding of a subroutine or a general purpose set of coding, it is often desirable to work with a range of addressing that is not possible to cover in a normal index. Or, in the case of a subroutine where it is necessary for the addresses to be variable depending on which part of the whole program called the address. Let's look at the latter case more closely.

It should be fairly obvious to the user that a general purpose subroutine cannot contain the address of the operations. Therefore if you wish to load the accumulator in this situation you do not have an absolute address to use in an LDA instruction to do the job. Instead you will have to compute the address from available information and store the result. The most efficient place to store it is in page zero, because it takes less time to put an item or retrieve it from there.

What you would really like to do is use the address you have computed (and is now stored in page zero) as the absolute address in your LDA instruction.

You could do this by keeping track of exactly where in memory the address bytes of that particular instruction were located, treating them as a data location and putting the computed address into them. Of course every time you made even the

slightest change in your program this location might shift, you would store the wrong information in the wrong place and your program would blow up!

Indirect addressing will let you achieve the same results with a minimum of fuss and bother, and with a minimum chance of corrupting your program if a minor change is made in it. Indirect addressing gives you the capability of addressing anywhere in memory with a calculated address.

7.5.2 Indirect Indexed Addressing, e.g. LDA (\$06), Y

Indirect indexed addressing is an address technique that provides a great deal of flexibility for advanced assembly-/machine-language programming. It is available on only eight instructions: ADC, AND, CMP, EOR, LDA, ORA, SBC & STA.

The standard way of designating indirect indexed addressing is

OPC (ZPL),Y

where OPC designates OPeration Code, e.g. LDA or STA.

(ZPL) The 'ZPL' designates that this instruction uses a Zero-Page Location. The one specified is actually the lower byte of a two-byte address. The ( ) indicates that the CONTENTS OF this zero-page address act as a pointer to point to the base address of the operand to be used by this instruction.

Y designates that the Y-register is to be used for indexing. This means the contents of the Y-register are used as an offset added to the base address to get the effective address to be used. ONLY the Y-register, never X, may be used with this addressing mode.

REMINDER: In Apple/MOS 6502 assembly- or machine-language programming the symbology ( ) means 'the contents of' the specified register or memory location, the name of which is enclosed in the parentheses.

Lets look at a typical indexed indirect instruction and see in some examples exactly what happens. Let us assume that at the time this instruction is fetched the following data is in the locations indicated:

```
(Y) = $20
($06) = $00;($07) = $08;  (06,07 is a pointer
        to $0800)
($820) = $C0.

LDA ($06), Y
```

specifies that the accumulator is to be loaded using a base address and offset technique. The pointer to the base address may be found in memory location \$06, or more specifically the LSB of the pointer may be found in \$06 and the MSB in \$07. Since the pointer \$06-\$07 contains the value of \$0800, the base address is \$0800. The offset is the contents of the Y-register or \$20. Thus the effective address is \$800 + \$20 = \$820 and the operation will result in the loading to the accumulator of the value which can be found in \$0820, the quantity \$C0.

In this addressing mode you have an instruction with the normal counter offset capability of indexed instructions. However the address which is indexed is not in the instruction itself, it is an address in a predetermined, fixed zero-page location and it is capable of pointing anywhere in memory.

Such a stand-alone address can be easily computed or modified by an entirely different portion of the program than that in which the instruction is located. Such flexibility can become especially important in system programming, e.g. the development of general-usage high-resolution graphics routines, interpreters, etc.

The fixed, predetermined location of the pointer-address means that when the program is altered you can avoid the hassle of keeping track of exactly where in memory the instruction is located and changing all of the instructions involved in direct modification of an address field, which is part of an instruction that can move from place to place.

This method of addressing exacts an execution time penalty. Operations using this addressing mode require five central processing unit time cycles compared to four time cycles for absolute, absolute indexed or zero-page indexed addressing, three for plain zero-page addressing or two for immediate addressing.

Now let's see how we can use this indirect method of addressing to rewrite our machine-language program as a subroutine for moving text page 1 to text page 2. In so doing we will remove the earlier restriction on moving a maximum of 256 bytes so that the rewritten program will transfer the whole \$400 bytes of the complete screen display.

The theory of operation of this program is as follows: First we declare where the program is to be located and where the pointers to the screen display page 1 and page 2 are to be located. Next we initialize these pointer locations to contain the addresses of the start of page 1 and page 2. Then we enter a loop that moves \$100 (a full memory page) of bytes from page 1 to page 2. Then we modify

both pointers to specify the next memory page. However before doing anything with these modified pointers, we check to see if we have already completed the task. If not we loop back to move another $100 bytes, but if we have completed we exit the subroutine.

Figure 7.5A

Program to move Text Page 1 to Text Page 2 using Indirect Indexed Addressing

```
Symbolic Form     Hex Form      Remarks
-------------------------------------------------------------------------------
* DECLARATIONS

       ORG $5FFA    <none>       Start program at arbitrary memory location $5FFA

PP1    EQU $06      <none>       Declare PP1 (Page 1 indirect address pointer) to
                                 occupy zero-page memory location $06 (LSB)
                                 & $07 (MSB)
PP2    EQU $08      <none>       Declare PP2 (Page 2 indirect address pointer) to
                                 occupy zero-page memory location $08 (LSB)
                                 & $09 (MSB)

* INITIALIZE PAGE POINTER 1 (PP1 AT $6,7)
   TO HOLD BASE ADDRESS P1 ($0400 - START OF PAGE 1)
  INITIALIZE PAGE POINTER 2 (PP2 AT $8,9)
   TO HOLD BASE ADDRESS P2 ($0800 - START OF PAGE 2)

ENTRY LDA #$04     5FFA: A9 04  Load $04 (MSB of $0400 - start of text page 1
                                into A-register to start moving it to MSB of PP1
      STA PP1+1    5FFC: 85 07  Store $04 as MSB of PP1 (Page Pointer 1)
      LDA #$08     5FFE: A9 08  Load $08 (MSB of $0800 - start of text page 2
                                into A-register to start moving it to MSB of PP2
      STA PP2+1    6000: 85 09  Store $08 as MSB of PP2 (Page Pointer 2)
      LDY #$0      6002: A0 00  Load the Y-register with zero for use as LSB
                                of both text page pointers (PP1 & PP2)
      STY PP1      6004: 84 06  PP1 (6,7) now fully loaded with $0400
      STY PP2      6006: 84 08  PP2 (8,9) now fully loaded with $0800

* ENTER A LOOP THAT RUNS Y-REGISTER FROM 00 TO FF, MOVING $100 BYTES
  ($00-$FF = ONE MEMORY PAGE = 1/4 OF TEXT SCREEN DISPLAY PAGE)
  FROM TEXT DISPLAY PAGE 1 TO TEXT DISPLAY PAGE 2

LOOP  LDA (PP1),Y  6008: B1 06  Move from Text Page 1 to A-register
      STA (PP2),Y  600A: 91 08  Move Page 1 byte from A-register to Page 2
      INY          600C: C8     Increase offset by 1
      BNE LOOP     600D: D0 F9  Branch backward to LOOP (-7 dec = $F9) if
                                Y-register has not counted down to zero

* INCREMENT MSB OF BOTH PAGE POINTERS BY 1 TO SET UP FOR MOVING NEXT MEMORY PAGE

NXT   INC PP2+1    600F: E6 09
      INC PP1+1    6011: E6 07

* TEST TO SEE IF YOU HAVE REACHED THE END OF THE MOVE.
    IF NOT, BRANCH BACK TO MOVE ANOTHER MEMORY PAGE,
    OTHERWISE, DROP THROUGH TO EXIT AND RETURN TO WHEREVER CALLED FROM

      LDA PTR+1    6013: A5 07
      CMP #$08     6015: C9 08
      BCC LOOP     6017: 90 EF  Branch back to LOOP (-17 dec = $EF bytes if
                                comparison test shows whole move not completed.
EXIT  RTS          6019:        Finished - Return to wherever called from
```

There are several things you should notice about this program:

1. A complete explanation of how the program works is built into the text of the program. This makes the program look longer (and perhaps more complicated) but this self-documentation makes it much easier to use.

2. The program is still lightning fast, taking about .016 seconds to move the contents of 1024 memory locations.

   It is slower than the previous programs because 1. it moves the contents of four times as many locations, 2. the use of indirect addressing added two microseconds to each indirectly addressed instruction, a seemingly trivial amount that begins to add up when one applies it to several instructions in a loop which is traversed many times, and 3. some extra set-up and address modification overhead is also added to the program.

3. Although it illustrates a way that indirect addressing can be used and can be useful, this is not a particularly good program for the use intended.

   We would have achieved the same results in both a faster (less than .01 second) and less complicated manner if we had just modified figure 6.9.4B or 6.9.4c by replacing

LDA P1,X by	LDA P1,X
STA P2,X	STA P2,X
	LDA P1 + $100,X
	STA P2 + $100,X
	LDA P1 + $200,X
	STA P2 + $200,X
	LDA P1 + $300,X
	STA P2 + $300,X

4. However the program using the extra loop and indirect addressing could be easily modified to do other jobs which the absolute-indexed version of the program even using a modification similar to that above would not have been suitable for.

   For example, to move high-resolution graphics display page 1 to hi-res graphics page 2, you would have to insert 32 LDA/STA pairs in order to use simple indexing, while if one used the indirect indexing version of the program you merely change P1 to $2000, P2 to $4000 and CMP #$08 to Cmp#$40.

5. GOTCHA! There is one serious (but totally unnecessary) fault that I suspect most of you missed in the use of this program for moving hi-res page 1 to page 2. Part of the program ($5FFA - $5FFF) is located in hi-res page 2 and hence would be destroyed by the move. You must be careful.....

   For programs up to about $90 (dec 240) bytes in length, all except the very top of memory page 3 is normally safe and convenient to use — as long as you don't try to put two programs in the same space at the same time.

6. It sounds like a great idea to generalize this program slightly so that one could have a generalized data movement program, right? Wrong!

   Why go to the trouble of writing your own machine-language program when a program to do exactly the same task — only better — is inside the system monitor firm-

ware waiting to be used every time you run your computer.

7.5.3 Indexed Indirect Addressing, e.g. LDA (\$80,X)

The indexed indirect addressing mode is the last of our addressing modes used for sophisticated assembly/machine-language programming. It is available for only eight instructions: ADC, AND, CMP, EOR, LDA, ORA, SBC, and STA.

Indexed indirect addressing is commonly used in picking up data from a table or list of addresses in such activities as polling I/O devices or in performing string or multi-string manipulations.

Whereas indirect indexed addressing could use only the Y-register, indexed indirect addressing can use only the X-register. This sometimes causes inconvenience, but keeps you from confusing one mode for the other.

In this mode the contents of the index register X is added to the zero-page address. This allows you to compute and change a specific indirect pointer. Consider the following situation:

(X) = \$04

(\$80) = \$00 (\$81) = \$02 80,81 is pointer to \$0200
(\$82) = \$00 (\$83) = \$03 82,83 is pointer to \$0300
(\$84) = \$00 (\$85) = \$04 84,85 is pointer to \$0400

```
LDA ($80,X)
```

takes the base address \$80, indexes it by the contents of the X-register, \$04 to get an effective address of \$84. It then uses the pointer at \$84 to obtain the indirect address \$0400.

You pay an execution-time penalty for this form of addressing. It always takes six processor cycles to fetch a single operand compared to five processor cycles for indirect index. Also, the processor will not cross over page boundaries, but will wrap around to the beginning of the page.

Chapter VIII
Machine-Language Programs Can Live Happily in a BASIC Environment

Machine-language programs can be made available to a BASIC program in several different ways. Each section following covers a different technique.

8.1
The Simplistic Approach: Using a Binary Disk File for the Machine-Language Program Loaded and Called by the BASIC Program

Suppose, for example, you had a machine-language program consisting of the following 20 bytes to be inserted in memory starting at memory address $0300.

FF EE DD CC BB AA 99 88 77 66 55 44 33
11 00 01 02 03 04
(Don't try to decipher the program,
it is nonsense code)

Working directly from the keyboard (NOT inside a program) you could follow the procedure shown in figure 8.1A to use this machine language code available:

Figure 8.1A

Procedure for Saving & CALLing Machine Language Program as Binary Disk File
(Illustrated by Means of Sample Program `NONSENSE')

```
] CALL -151   (to enter the monitor)
   (If your Apple doesn't have the Autostart ROM, then just press RESET)
 * 300: FF EE DD CC BB AA 99 88 77 66 55 44 33 22 11 00 01 02 03 04
   (Program is now in memory locations $0300-$0314)
 * BSAVE NONSENSE, A$300,L20
   (Program is now saved on disk as a binary file named NONSENSE)
   (Note that the starting location and length can be specified in either
    Hex (A$300,L$14) or Decimal (A768,L20) or mixed (A$300,L20), as here)
```

To get this binary disk file to work with your program you merely load and call the program some time before it is needed as follows:

```
 50 PRINT CHR$(4);"BLOAD NONSENSE"
          (CHR$(4) is the infamous CTRL-D)
```

Any time you wish to execute the program call it:

```
 600 CALL 768
```

(768 is the decimal equivalent of $300. Yes, it would be nicer if one could CALL $300, but as we have seen Applesoft BASIC allows only decimal numbers for PEEKs, POKEs and CALLs.)

8.2
POKEing Small Machine-Language Programs into BASIC

8.2.1 POKEing Each Byte Individually

With this approach you must convert both the addresses of the memory locations into which you will put the code and the code itself into decimal form.

Various utility programs are available to assist in this task or you can use Hexadecimal = Decimal Conversion Tables such as those reproduced as figures 7.1B and 7.1C.

This technique was used in the program of Case Study 5.9, which showed how the inverse procedure for converting backwards from the POKEs to the machine-language instructions could be accomplished.

Figure 8.2A

Procedure for POKEing Machine Language Instructions One Byte at a Time
(Illustrated by Means of Sample Program `NONSENSE')

(1) Find the decimal equivalent of the hexadecimal starting location for the program. You may perform the conversion manually, use the computer or use the table look-up procedure described in Section 3.5 which uses the table Figure 3.5C

For example program `NONSENSE', the starting location $300=>768

(2) Find the decimal equivalent of each byte. You have the same choice of methods. I find look-up in Table 7.1B the easiest

For example program `NONSENSE' the conversions are as follows:

Hex	FF	EE	DD	CC	BB	AA	99	88	77	66
Dec	255	238	221	204	187	170	153	136	119	102
Hex	55	44	33	22	11	00	01	02	03	04
Dec	85	68	51	34	17	0	1	2	3	4

(3) Then POKE the first byte into the start-of-program location and and each subsequent byte into the next consecutive location.

For sample program `NONSENSE' this is shown below:

```
50 POKE 768,255:POKE 769,238:POKE 770,221:POKE 771,204:POKE 772,187
51 POKE 773,170:POKE 774,153:POKE 775,136:POKE 776,119:POKE 777,102
52 POKE 778,85:POKE 779,68:POKE 780,51:POKE 781,34:POKE 782,17
53 POKE 783,0:POKE 784,1:POKE 785,2:POKE 786,3: POKE 787, 4
```

Calling remains unchanged, e.g.

```
600 CALL 768
```

Note that even though this process is very easy to understand it can become tedious and error-prone if the program to be POKEd is very long.

Incidentally, this procedure is just as applicable to INTEGER BASIC as it is to Applesoft.

8.2.2 Using Read and Data Statements to Simplify POKEing

Since a program goes into consecutive memory locations you can reduce typing and program overhead by using a FOR loop to specify the consecutive memory locations and put the program DATA into the POKEs by means of READ statements. Figure 8.2B shows how the example of figure 8.1 appears when this technique is used.

Figure 8.2B
POKEing Machine Language into BASIC
Abbreviated Form using DATA & READ Statements

```
 50  FOR I=768 TO 787
 52  READ BYTE:POKE I,BYTE
 54  NEXT I
 56  DATA 255,238,221,204,187,170,153,136
 58  DATA 119,102,85,68,51,34,17,0,1,2,3,4
```

Note how much shorter and easier this technique is to follow than a separate POKE Instruction for each location, even with a program as brief as the sample.

8.3
Tricking the Apple Monitor into Working Inside a BASIC Program

This technique is often called the Lam technique after its originator. It involves storing a set of monitor commands in a string variable inside a BASIC program, then tricking BASIC into thinking that this string was entered via the keyboard, executing the monitor commands and returning to Applesoft. This method of imbedding a machine-language program was used in the Utility Program /Applesoft Patch of Section 5.10 (figure 5.10A).

Figure 8.3A describes the set-up procedure and uses the same program 'NONSENSE' that was used in the two previous examples.

Figure 8.3A

MONITOR ROUTINE WORKING INSIDE BASIC

(1) Set a string variable (Z$) equal to exactly what you would have entered directly into the monitor in Section 8.1 (Line 50 of program below).

(2) Add another monitor command, the'running return to Applesoft' routine "Z$=Z$+" N D823G"' (Line 52 of program below).

(3) POKE this combined string into the Keyboard Input Buffer, the same place it would go if one had typed it in at the keyboard (Line 54).

(4) Set the STATUS register (which holds the old contents of the program counter when one transfers into the monitor) to zero (Line 56).

(5) On execution do this set up and immediately CALL -144 to trigger the keyboard scan and execution of the machine-language program. (Line 360).

(6) It is convenient to form the set-up as a subroutine. It is best not to perform the CALL-144 from inside a subroutine.

.....Sample Program.....

```
 50  Z$ = "300: FF EE DD CC BB AA 99 88 77 66 55 44 33 22 11 00 01 02 03 04"
 52  Z$ = Z$ + " N D823G
 54  FOR I = 1 TO LEN(Z$): POKE 511 + I, ASC (MID$(Z$,I,1)) + 128: NEXT
 56  POKE 72,0 : RETURN : REM Use RETURN only if to be used as a subroutine
```

.....Main Program CALLing Procedure for "NONSENSE".....

```
 360  GOSUB 50: CALL -144: REM CALL -144 should not be inside a subroutine
```

To execute the program you perform this set-up then CALL -144. This scans and executes monitor commands in the keyboard-input buffer area (Memory Page 2). The Apple thinks it is in monitor mode and has just received a command from the keyboard. It loads the program, then transfers control to $D823 and makes a 'running return' to Applesoft ready to decode and execute the next line of the Applesoft program.

This technique retains the obvious visible identity of the machine-language program rather than obscuring it by conversion of the bytes to decimal form. In terms of memory requirements this technique is quite efficient, using less than half the number of bytes of memory that are required for POKEing bytes individually. For medium length programs the technique using READ and DATA statements is about equally efficient in use of memory.

One final thing. This same exact technique is equally as applicable to Integer BASIC as it is to Applesoft. There are only two differences:

.1. Character strings in Integer BASIC are a bit more primitive than in Applesoft. They must be DIMensioned before use.

    The Statement DIM Z$(100) at the beginning of line 50 will take care of that requirement.

2. The running return to Integer BASIC should be appended directly onto the program rather than by a separate concatenation operation. (This can be done in Applesoft too, but I think the procedure is more self-explanatory if the monitor string is separate.) The running return routine is at $E88A for Integer BASIC rather than at $D823, the Applesoft running return location.

    Thus for Integer BASIC delete the explicit concatenation of line 52 and instead do it implicitly by putting 'NE88AG' the end of the character string in line 50.

3. Line #54 must be changed to ...,ASC (Z$(I)):Next I.

8.4
Imbedding Machine Language in an Applesoft Program 'Transparently'

8.4.1 When Relocatable Machine-Language Code is Available

If you have a machine-language program that is 'relocatable', i.e. which will work equally well even though it is moved from one location in memory to another, you may imbed it within your Applesoft program transparently. When binary code is so imbedded, it automatically goes with the Applesoft program wherever the Applesoft program goes, even when it is STOREd to a

diskette or LOADed back from a diskette. This saves the separate binary file and BLOAD required of the simplistic technique.

Accomplishing this feat involves reading and resetting the PRGEND (PRoGram END) pointer that is located in zero-page memory locations $00AF and $00B0 (decimal 175 and 176). If you want to know all about this pointer you can find a complete explanation of Applesoft memory allocation and the function of PRGEND in Chapter 15.

The machine-language program is entered after the original end of the BASIC program. Then the pointer is reset so that Applesoft (and the rest of the Apple) think that the program ends at the end of the added binary code.

A strange characteristic of Applesoft makes it necessary for machine-language code to be relocatable. Each time a line of code is added or deleted or any other change (such as resequencing) is made that changes the length of the program, all of the memory allocations for variables (and the machine-language program to be added) are pushed upwards to allow the extra program space needed.

Thus if you allow any changes whatsoever to the program, the machine language must be relocatable in the sense that it must work when it is slipped upward or downward as a result of this process. (Such changes are almost inevitable. If nothing else, you must imbed a CALL or CALLS to the machine-language program. The exact values are less easily precomputed than the location of the program.)

The set up process is shown in figure 8.4A.

Figure 8.4A

Procedure for Imbedding Machine Language in Applesoft 'Transparently'

1. Read and record the Program Pointer, i.e., PRINT PEEK (175)+256*PEEK(176)
2. Add 20 to this value.
3. Convert to Hexadecimal.
4. Enter the monitor (CALL -151) and start entering the machine-language program at this address. Make sure the program ends with an RTS (ReTurn from Subroutine).
5. Record the total number of bytes of program entered, as well as the address of the next available memory location.
6. Set the Program Pointer(s) to the address of the next available location beyond the machine-language code.
7. Save the Applesoft program (which now extends to the end of the machine-language code) to a diskete using any desired name.

To recover and use the program one should:

1. Clear the sytem of any residue of the program, e.g., use a FP or a NEW.
2. LOAD the program.
3. At the point in the Applesoft program where you want to call the machine-language program, enter the following line of code:

```
600 X=PEEK(175)+256*PEEK(176):Y=X-N:CALL Y
```

(N=the number of bytes of machine code. Now that Y has been computed, CALL Y can be used as often as desired for subsequent CALLs.)

8.4.2 When Relocatable Machine-Language Code is not Available

A number of utility programs have been developed to allow you to put the machine-language program to be bound to an Applesoft program, at a fixed location (usually in front of the BASIC program).

You can accomplish the same thing that these utilities provide by using the memory-patching procedure described in Section 15.4 (and 15.8) The idea is this:

1. Make the first module of the program a single statement: 1 GOTO 2.
2. The first module will always take the same amount of space which can be determined by the procedures described in Chapter 15. The machine-language program can be written and/or assembled to fit immediately after this module, its size determined and the space it occupies blocked off as a prohibited zone for BASIC.
3. Use the allocation patching technique and start the second module of program just beyond the end of the prohibited zone.

Warning: When you do this kind of patching you are tricking the memory allocation procedures in Applesoft to do things differently from the way they normally would. Don't try this procedure until you have read and understood the Applesoft memory allocation concepts presented in Chapter 15.

Chapter IX
Overview of Apple System Memory Allocation

9.1
The Easy Way and the Hard Way to Look at Apple System Memory Organization

Early in this book, before we formally introduced hexadecimal numbers, we discussed the organization of Apple memory into 256 pages of 256 bytes each and the simplifications that were possible when you use hexadecimal abbreviations.

We suggested you think of memory as $100 pages of $100 bytes each, the $ being an indication that you were counting with 16 symbols (0,1,2,3, 4,5,6,7,8,9,A,B,C,D,E, and F) rather than with the normal 10 symbols (0,1,2,3,4,5,6,7,8, and 9). We showed how, even when you used decimal addressing for double-PEEKs and double-POKEs, you could take advantage of hexadecimal tables to make the determination of decimal addresses easier.

You have been gradually familiarized with hexadecimal numbers from that point in the book to this. Hopefully hexadecimal numbers no long shock you.

You have seen the simple logic behind the hexadecimal system and you have seen the intimate relationships of the hexadecimal viewpoint to the architecture and the internal structure of the Apple system. Do you still consider hexadecimal numbers to be useless and undesirable intellectual baggage to be avoided whenever possible? If so you can continue to avoid them. I will continue to present all memory information in both decimal and hexadecimal form.

However, even if you have absolutely no interest in machine language, you will find that it is much easier to acquire knowledge of the logical structure and layout of Apple memory if you think in hexadecimal terms. You will find it much easier to remember its layout, subdivisions, and even individual key memory locations if you think of hexadecimal addresses.

As you have read repeatedly, a fully implemented Apple II system can contain 65,536 directly addressable memory locations. That's $10000 locations in hexadecimal terms. 65,536 is an odd-sized, hard-to-remember number; $10000 is a nice, round, easy-to-remember number. The high-resolution graphics primary display occupies memory locations 8192 to 16383 — not easy numbers to remember; in hexadecimal it is much easier to remember: $2000 to one less than $4000 ($3FFF).

In hexadecimal addressing, locations are counted consecutively from $0000 to $FFFF ($10000-1).

In decimal memory addressing for Integer BASIC, memory locations are always counted as locations 0 to 35,735, followed by locations -35736 to -1. Of these 65,536 decimal numbers, the two that are furthest apart (at the opposite ends of the address range) are 0 and -1. Adjacent to one another in the middle of the range are +32767 and-32768.

For Applesoft BASIC environment this is still the most common method of addressing, but a second set of unsigned decimal addresses are also commonly used.

These differences in simplicity/complexity and convenience are significant. They are also no mere isolated coincidences. When the breakdown of functions within the Apple System is considered in decimal terms, it consistently requires the use of odd-sized, hard-to-remember addresses. When the breakdown is described in hexadecimal terms, it fits exactly to the internal logical structure of the system and thus tends to follow a straightforward, easy-to-remember pattern.

Now let's review some key aspects of the logical simplicity we get from hexadecimal memory addressing.

Architecturally the Apple system memory is broken down into $100 pages of $100 hexadecimal digits each. All addresses are two bytes or 4 hexadecimal digits long (or can be padded into that form by adding leading zeros).

For any arbitrary hexadecimal address $HHLL, the two high-order hexadecimal digits, HH, that is the high-byte or MSB, specify the page of memory, while the two low-order digits, LL, the low-byte or LSB, specify the particular location within that page.

9.2
The First Cut: RAM, ROM and 'SPECIAL I-O'

The bottom 48K of memory in the Apple II system (K stands for binary Kilo = 1024 so 48K =

49152) is reserved for user-changeable RAM (Random Access Memory). This memory is supplied in modular packages: 16, 32, or 48K, but the vast majority of Apple computers contain the full quota of 48K.

RAM memory is volatile; that is, the information in it evaporates and is lost whenever power is turned off.

The top 16K (16K = 16384) bytes of Apple II system memory is reserved for Apple System Firmware and also for special functions associated with the input-output system and expansion slots of the Apple System. It is subdivided into a 12K ROM section and a 4K Special I-O section.

Figure 9.2A
Apple II+ System Memory Map
(Lowest Level of Detail)

RAM Pages				Special I-O Pages				ROM Pages			
K	Pge (Hex)	Pge (Dec)	Use	K	Pge (Hex)	Pge (Dec)	Use	K	Pge (Hex)	Pge Dec	Use
1st	$00	0	Zero Page	49t	$C0	192	Built-in IO	53d	$D0	208	A/S Intptr
	$01	1	Sys Stack		$C1	193	Slot#1 ROM		$D1	209	A/S Intptr
	$02	2	Keybd Bufr		$C2	194	Slot#2 ROM		$D2	210	A/S Intptr
	$03	3	ML&DOS Vec		$C3	195	Slot#3 ROM		$D3	211	A/S Intptr
2nd	$04	4	Text&LoRes	50t	$C4	196	Slot#4 ROM	54t	$D4	212	A/S Intptr
	$05	5	Text&LoRes		$C5	197	Slot#5 ROM		$D5	213	A/S Intptr
	$06	6	Text&LoRes		$C6	198	Slot#6 ROM		$D6	214	A/S Intptr
	$07	7	Text&LoRes		$C7	199	Slot#7 ROM		$D7	215	A/S Intptr
3rd	$08	8	Either	51s	$C8	200	SlotROMExp	55t	$D8	216	A/S Intptr
	$09	9	BASIC DorP		$C9	201	SlotROMExp		$D9	217	A/S Intptr
	$A	10	or		$CA	202	SlotROMExp		$DA	218	A/S Intptr
	$B	11	2nd Txt&LRG		$CB	203	SlotROMExp		$DB	219	A/S Intptr
4th	$0C	12	BASIC DorP	52d	$CC	204	SlotROMExp	56t	$DC	220	A/S Intptr
	$0D	13	BASIC DorP		$CD	205	SlotROMExp		$DD	222	A/S Intptr
	$0E	14	BASIC DorP		$CE	206	SlotROMExp		$DE	223	A/S Intptr
	$0F	15	BASIC DorP		$CF	207	SlotROMExp		$DF	224	A/S Intptr
.	.	.	BASIC DorP						.	.	A/S Intptr
.	.	.	BASIC DorP						.	.	Intr/Mon
.	.	.	BASIC DorP						.	.	Monitor
9th	$20	32	BASIC DorP					64th	$FC	252	Monitor
to	to	to	or						$FD	253	Monitor
16t	$3F	63	HiResGphic						$FE	254	Monitor
			Display #1						$FF	255	Monitor
17t	$40	64	BASIC DorP								
to	to	to	or								
24t	$5F	95	HiResGphic								
			Display #2								
.	.	.	BASIC DorP								
.	.	.	Da,Pr or S								
.	.	.	BASIC PorS								
48t	$BC	188	BASIC PorS								
	$BD	189	BASIC PorS								
	$BE	190	BASIC PorS								
	$BF	191	BASIC PorS								

The 12K ROM section normally contains the system monitor and a BASIC interpreter. This memory is provided in the form of manufacturer-supplied ROM chips. Whenever the computer is turned on the software or firmware in these chips is immediately available for use. It is not volatile; the information contained in ROM memory is not lost when power is turned off. However it also cannot be altered by the user during normal system operations.

The remaining 4K of the top 16 is rather queer. Some of it is addressable, but never exists and never will exist. Some of it exists, not in the Apple main computer, but on peripheral cards that may be plugged into the Apple's expansion slots. The internal organization of this area is not simple and is covered at length in Chapter 17.

(NOTE: Although the top 16K of memory is normally considered systems space within which the uninitiated user roams around at his own risk, you can buy a special RAM card that provides optional additional RAM memory which can be switched into this address space. Under some circumstances the user can write information into this memory as well as read information from it. But it can also be locked to prohibit unauthorized or unintentional writing. There are definite constraints on the use of this optional additional memory. What this involves is beyond the scope of our initial discussion.)

Figure 9.2A presents this broad overview of memory allocation semi-pictorially, and identifies exactly which pages are assigned to which category of use.

9.3
The Second Cut: Functional Allocation of Pages

Each page of memory within the Apple has its own allocated function or functions. Many are just part of the free space normally available for the user to use as he sees fit, but many others have specific functional responsibilities within the plan of the Apple II system. The remainder of this section starts at the bottom (Page 0) and works its way up to the top (Page $FF or decimal 255). As you go up the memory from page zero towards page 256 you will find:

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

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

Page 1 ($01) Used for the System Stack

This is a special area used primarily for subroutine returns (both machine language and BASIC), inter-

rupts, and parameters used in re-entrant coding. Only the most careful and experienced programmers should ever fool with this area.

Page 2 ($02): 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 ($03): 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 locations 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.

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

The 1024 locations on these four 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 eight macro-lines of 128 bytes, each of which contains 3 text lines (one on the top ⅓ of the screen, one in the same relative position in the middle ⅓, and the last in the same relative position in the bottom ⅓ of the screen). The remaining eight 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 $08-$0B): 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 bring the text here. (It must be POKEd in or moved from page 1.)

Pages 8 upward ($08 upward): Default Applesoft BASIC Program and Data Space or Default Integer BASIC Data Space

Note: Unless an overt use of LOMEM by the user alters the situtation, 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 $20-$3F): High-Resolution Graphics Primary Video-Display (HGR pg 1)

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 × 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 eight text macrolines each 128 'standardized' characters long which break into three screen lines (top ⅓, middle ⅓, bottom ⅓) plus eight 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. eight slices each containing 7 dot positions stacked one over another.

Thus the 40 'standardized' character positions also represent 7 × 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 ½ position left or ½ 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 four 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 four 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 eight slices (eight bytes of memory) stacked one above another per displayed 'standardized' high resolution graphics character, there are 8 × 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 initially 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 $40-$5F): High-Resolution Graphics Secondary Display Page (HGR pg 2)

The interior layout is the same as HGR pg 1 but $2000 higher.

Pages before 150 (before Page $96): 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 is 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 $96-$BF): 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 ($C0-$CF): Special Hardware I-O Area

This area is reserved for Input/Output and 'slot' (pheripheral) 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 eight 16-byte chunks, each of which is assigned to one of the eight 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 ($D0-$FF): 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 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'.

Chapter X
The Apple System Quick-Access Area (Memory Page 0 ($0000-00FF))

10.1
Zero-Page Addressing as a Memory Saver and Means of Speeding Computation

Page zero of memory is a very special place. The microprocessor in the Apple II has a mode of addressing, known as zero-page addressing, which allows locations in page zero to be specified with a single byte of address rather than the normal two-bytes. This means shorter, faster programs.

The System Monitor, Applesoft and Integer BASIC interpreters and the DOS are all program packages that can benefit from the shorter, faster programs produced by heavy use of page zero. You can see the speed advantages of zero-page addressing by comparing the number of machine cycles (a measure of time) it takes for any instruction when zero-page addressing is used, compared to addressing on any other page. (See tables in Chapters 6 and 7).

10.2
Zero Page Usage

Figure 10.2A — 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

For details of how each zero-page location as indicated above is actually used look up the location in the Apple Atlas database.

Chapter XI
The Apple System Stack Page

11.1
Introduction to the System Stack — A Last-In First-Out Storage Area

The page of 256 memory locations from $100 to $1FF (decimal 256-511) is called the Apple System Stack. The Stack is different in characteristics from page zero, but like page zero, it is a very special area of memory. In fact, the stack page is sometimes described as a hardware area not totally under program control because of the influence of the S-Register.

The stack is used in conjunction with the S-Register or Stack Pointer to provide positive control of the system in situations where control is passed from one portion of a program to another.

The Applesoft or Integer BASIC interpreter will automatically take care of all required stack operations for the BASIC programmers as long as they follow the established rules for subroutine set-up and calling. This is true for routine machine-language programming as well. However, machine-language programmers must use the JSR (Jump to SubRoutine) where the BASIC programmer uses a GOSUB and the RTS (ReTurn from Subroutine) instruction where the BASIC programmer would use a 'RETURN'.

When programming becomes more sophisticated, conditions arise that make it valuable for programmers to understand and exercise their own control over the operations of the system stack. But this control is exercised with the caveat that programming errors involving the system stack are often among the most difficult to diagnose and most destructive to program integrity.

11.2
Subroutines, The Program Counter Store, Pushing and Popping

If you have a program (either in BASIC or machine language) where control is passed to a subroutine, the system must keep track of where control must be returned after the subroutine is executed. When the system executes a GOSUB in BASIC (or its equivalent, a JSR in machine language), the program counter is set to the start of the subroutine. Meanwhile, the location of the next instruction in normal sequence is PUSHed into a special location which we will tentatively identify as the program counter store.

The subroutine return address is held there until you reach the end of the subroutine, marked in BASIC by a RETURN command or in machine language by an RTS (ReTurn from Subroutine) instruction. At that point the return address is POPped back into the program counter, and the program continues in its normal sequence as if it had never been diverted into the subroutine.

In some early computers the program counter store was a single pre-defined memory location or register, but this created problems. When the contents of the program counter was PUSHed onto the program counter store, its old contents could be destroyed and with it any possibility of returning control to the original spot in the main program.

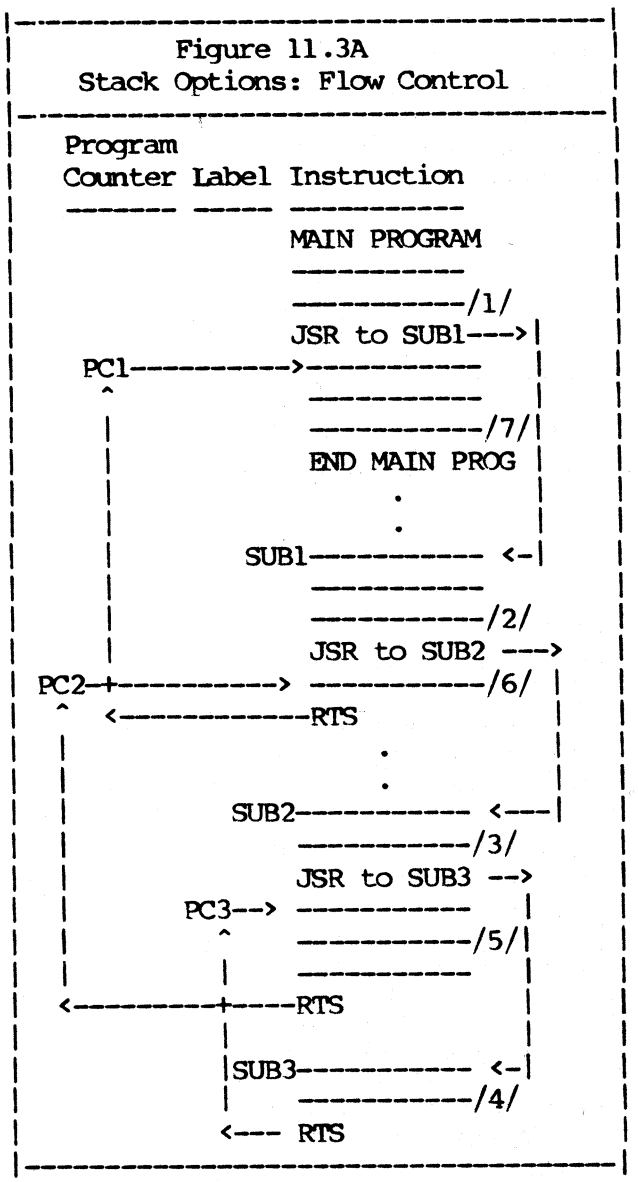
To handle two levels of subroutines you need two levels of program counter store; for three levels of subroutine, three levels of program counter store, and so on. With the system stack used in the Apple II, up to 128 levels may be handled, a number not likely to be exceeded in most programs.

The rule for operation would be similar to that for plates in the push-down stacks used in many cafeterias: the plate most recently pushed onto the stack will be the first popped off. Businessmen and accountants call this procedure a last-in-first-out (L-I-F-O) procedure.

11.3
Combined Operations of the S-Register and Stack Page in the Apple II

The current size of the stack is controlled by the S-Register, an eight-bit register capable of specifying $2^8 = 256$ locations. The stack is empty when this register is initialized to the bottommost location of the stack — position $FF within the system stack page $01. Thus the active address of the top of the stack is $01FF and the stack is functionally empty.

Figue 11.3A diagrams the flow of control in a situation where a main program calls SUB1, which in turn calls SUB2, which in turn calls SUB3.



At each of the stages marked with slashes, the contents of the stack are shown. (NOTE: For locations beyond the current location of the stack pointer the stack is effectively empty so it is shown in that fashion, even though a monitor check of the memory might show a residual value that has not been zeroed out.) At the start of the main program (/1/) the stack has not been used. Its status and that of the S-Register are as shown in figure 11.3B1.

Figure 11.3B1
S-Register & Stack Status
(Main Program)

S-Register: $01FF ($FF)
Stack Empty

When entry is made from the main program to the first level subroutine, the main program return address is PUSHed — stored at the top of the stack, which at this moment is also the bottom of the stack. Since the program counter is a 16-bit or 2-byte long register, the memory location that you are to return to (PC1) must be stored as two bytes: PC1H and PC1L, and the S-Register is decremented by two to $FD, the new top of the stack (i.e. next program counter store location). This situation occurs at location /2/ in figure 11.3A. The S-Register and stack status are shown in figure 11.3B2.

Figure 11.3B2
S-Register & Stack Status
(Early in SUB1 at /2/)

S-Register: $01FD ($FD)

Stack Status:

$00FE	PC1L
$00FF	PC1H

Note that the return address occupies two memory locations: $01FF and $01FE. The S-Register keeps track of the next available unoccupied location so it decrements twice to $01FD.

When entry is made to the second-level subroutine the return address PC2 in the first-level subroutine is PUSHed into locations $01FD and $01FC and the stack pointer decrements twice to $01FB. See figure 11.3B3.

Figure 11.3B3
S-Register & Stack Status
(Early in SUB2 at /3/)

S-Register: $01FB ($FB)

Stack Status:

$00FC	PC2L
$00FD	PC2H
$00FE	PC1L
$00FF	PC1H

When you jump out of the second-level subroutine (SUB2) to a third level subroutine (SUB3), the return location PC3 is PUSHed into location PC3s $01FB and $01FA and the stack pointer decrements two bytes to $01F9. (See figure 11.3B4.)

Figure 11.3B4
S-Register & Stack Status
(Early in SUB3 at /4/)

S-Register: $01F9 ($F9)

Stack Status:

$00FA	PC3L
$00FB	PC3H
$00FC	PC2L
$00FD	PC2H
$00FE	PC1L
$00FF	PC1H

When you finally hit an RTS (the machine language equivalent of a BASIC 'RETURN' statement), the stack is POPped and return address PC3 is restored to the program counter. The program can continue the execution of SUB2. Stack and S-Register conditions at /5/ are shown in figure 11.3B5.

Figure 11.3B5
S-Register & Stack Status
(Late in SUB2 at /5/)

S-Register: $01FB ($FB)

Stack Status:

$00FC	PC2L
$00FD	PC2H
$00FE	PC1L
$00FF	PC1H

When the RTS at the end of SUB2 is reached, the top return address in the stack (PC2) is POPped, thus returning the control counter setting to a position late in SUB1 and creating the stack status shown in figure 11.3B6 at location /6/ in SUB1:

Figure 11.3B6
S-Register & Stack Status
(Late in SUB1 at /6/)

S-Register: $01FE ($FE)

Stack Status:

$00FE	PC1L
$00FF	PC1H

When the RTS at the end of SUB1 is reached, it POPs the stack again, putting the top address in the stack PC1 back into the program counter and returning control to the main program. Thus the status of the stack at /7/ is once again empty as shown in figure 11.3B7:

Figure 11.3B7
S-Register & Stack Status
(Late in Main Prog at /7/)

S-Register: $01FF ($FF)

Stack Status:
Stack Empty

11.4
Use of the Stack by the Programmer

Thus far we have discussed the stack only as a means of handling control information, but instructions exist in the hardware repertoire of the microprocessor used in the Apple which allow you to push data into the stack and pull it out as well. These activities may be undertaken under program control. Properly used, they can allow you to write programs or subroutines which call upon themselves.

The most common and straightforward techniques for communicating with a subroutine is by exchange of program parameters at pre-agreed on memory locations. These pre-agreed upon locations may themselves contain the data to be used or they may contain pointers that point to the location of data. Another common method is to place the parameters to be exchanged at the start of the subprogram. In more general locations there is a chance the parameters may be altered by a piece of program that doesn't know they belong to the particular subroutine.

However, situations arise that require that the program not work with data of known currency. Indeed in such situations it may not even be necessary to know exactly where the data is located. You have to know how to get to it and how to make sure it is associated with the right subroutine and with the correct relative order of creation.

Use of the stack for holding and passing subroutine parameters is common in situations that occur when you are processing real-time interrupts or using a subroutine to call itself recursively.

In recursive programming a subroutine may call on itself using different parameters for the second call than for the first.

A different method of cummunicating parameters between programs or subprograms is relevant to these situations. It involves use of the stack itself rather than a conventional register or addressed location to pass the information.

This method allows the programmer to make sure that the data is associated with or is of the same degree of currency as a particular subroutine call. Otherwise input parameters or results from one moment in the time sequence of processor operations can become confused with those for another moment in the operations. Of course, in normal routine programming the kinds of time-sensitivity that this re-entrant coding protects you from is not often a problem. It tends to become a problem only when you attempt to share the performance of two or more different tasks using the same resources where one use overrides another. Thus it is a common problem for systems programmers trying to get a simple piece of hardware to do many things concurrently. It is no problem at all to the typical BASIC programmer.

The stack, with its built-in precedence of operations, can be used to make absolutely sure that parameters or results obtained from one call of that subroutine do not become confused with those from another call. It can also make sure that information that gets into a register as a result of an interrupt processing procedure does not become confused with the information in that same register just before or just after the interrupt occurred.

Special commands are made available in the machine language of the microprocessor used by the Apple System to simplify both re-entrant coding and the concommitant tasks of putting data in and taking data out of the stack. Some of the more significant commands are:

PHA — PusH Accumulator onto Stack

This instruction transfers the current value of the accumulator to the next location on the stack, automatically decrementing the stack to point at the next empty location.

PLA — PulL Accumulator from Stack

This instruction adds 1 to the current value of the stack pointer, uses it to address the stack and load the contents of the stack into the accumulator. Figure 5.10A is an example of a subroutine that uses these instructions to store calling parameters on the system stack and later recover them.

There is another matched pair of instructions for PUSHing and PULLing processor status information between the P-Register and the stack:

PHP — PusH Processor stack onto stack

This instruction transfers the contents of the processor status register unchanged to the stack, as governed by the stack pointer.

PLP — PulL Processor status from stack

This instruction transfers the next value on the stack to the Processor status register, thereby changing all of the flags and setting the mode switches to the values from the stack.

These instructions are particularly valuable as part of a process of rapid saving or restoration of system status when an interrupt occurs. Another pair of stack-affecting commands, BRK (BReak — an interrupt under program control) and RTI (ReTurn from Interrupt), are even more closely related to interrupts.

There is a special instruction which affects the stack pointer only: TXS - Transfer indeX to Stack pointer. It is particularly useful in setting up a third method of passing parameters to or from a subroutine: storing them immediately after the jump to subroutine instruction.

11.5 Interrupts

11.5.1 Overview

An interrupt causes the computer to stop processing and embark on an entirely new activity — the appropriate interrupt processing activity. When the interrupt processing is completed, an RTI (ReTurn from Interrupt) instruction returns control to the original part of the program.

Interrupt processing sometimes has to be done in real-time. For example, in data communications, sometimes a computer has to process each incoming byte or bit before the next one arrives, or else it will lose the information forever. In such cases the time-savings gained with zero page addressing may be significant.

However, interrupt processing may also occur in situations where time is not particularly significant. Then its major advantage is that it allows you to perform the interrupting operation at any time or place in a program.

An interrupt is triggered by a special signal, perhaps an electrical signal from an outside device, or a peripheral connected to the computer. Or, it can be triggered by pressing the

'RESET' button or by using the BReaK (BRK) instruction.

11.5.2 What Happens When an Interrupt Occurs?

When an interrupt occurs the Apple microprocessor automatically

1. Pushes the Program Status Register (P-register) onto the system stack for safekeeping and later re-use when normal processing is resumed;

2. Pushes the current value of the program counter onto the stack. There it will be available for restarting the program at the point where it was interrupted, and

3. Transfers control to the location specified by one of the three interrupt vectors: $FFFA,$FFFB; $FFFC,$FFFD; $FFFE,$FFFF.

$FFFE,$FFFF is used for run-of-the-mill IRQ's (Interrupt ReQuests) including those produced by executing the machine-language BRK (BReaK) instruction. $FFFC,$FFFD is used for 'RESETs' and $FFFA,$FFFB is used for NMI's (Non-Maskable Interrupts.)

The computer does an indirect jump. It uses these addresses as a vectored pointer, i.e. it points to the address where the next instruction can be found. The jump is called an Indirect Jump because indirect addressing incorporates this concept.

11.5.3 Programming Concepts and Techniques

The memory locations shown in figure 11.5A are relevant to interrupt processing:

Figure 11.5A

Memory Locations Important in Interrupt Processing

Address	Function or Reason for Importance
0000-00FF	Zero page, Used for indirect addressing
0100-01FF	System Stack
03FD-03FF	Monitor Special Locations (See Section 11.6)
FFFA-FFFB	Vector for Non-Maskable Interrupts (NMI's)
FFFC-FFFD	Vector for 'RESET' (may require CTRL key also)
FFFE-FFFF	Vector for Interrupt ReQuests (IRQ's) and BReaK requests (BRK's)

The specific details of interrupt processing at the machine-language level deserve explanation at greater length than is possible here, but a few ideas should be emphasized:

1. On interrupt, the microprocessor always stores the program counter location for the instruction that was interrupted, and the processor status (P-register).

This is the minimum information needed to recover and proceed again from the point of interruption (providing that none of the other registers is altered during the interrupt processing). The stack provides a last-in-first-out storage capability. If an interrupt occurs within the interrupt processing routine, the system can handle it too without becoming confused as to where it should return control when the second-level interrupt is completed.

Only after this minimum recovery information is safely salted away is control transferred to the interrupt routine indirectly via the appropriate vector pointer: $FFFA,$FFFB; $FFFC,$FFFD; or $FFFE, $FFFF.

2. The RTI-ReTurn from Interrupt instruction performs the inverse operation. It takes three bytes from the top of the stack and puts them back into the program status register and program counter. This gives it the information needed to restore the system to the same status and location in the program that it was in before the interrupt transferred you off into the interrupt processing routine.

3. Between the interrupt and return you are free to do whatever processing you wish.

4. When you start writing machine-language code for interrupt processing, you will find yourself continually dealing with the system stack and with indirect addressing, usually combined with some form of indexing — i.e. indexed indirect addressing or indirect indexed addressing.

The Zero Page of memory is particularly convenient for doing indirect addressing using vectored pointers. If you use the Atlas to examine the specific allocations that are permanently assigned to the zero page you will see many vectored pointers used by the system monitor, Applesoft, Integer BASIC, and the Disk Operating System in order to implement their internal operations. You may frequently find yourself using some of these standard firmware pointers.

5. In many cases the processing you want to do while interrupted will involve using the A-, X- and/or Y-registers. If you plan to change them it is your responsibility to restore them to pre-interrupt status before return from interrupt.

A convenient way to do this is to have the first few instructions in your code push them onto the stack using PHA (and register

interchange instructions TXA and TYA and/or TXS). At the end of interrupt processing do the inverse using PLA to pull the information back from the stack into the accumulator (and by using TAX and TAY into the X- and Y-registers, as relevant.)

Taking such precautions, you can undertake any processing you desire during the interrupt, yet completely restore the central processor to its pre-interrupt status when you are finished, just as if the interrupt had never occured.

6. A hardware condition or a machine-language BRK instruction used by Applesoft for implementing the '&' token can generate an interrupt. It is possible for your interrupt-processing program to distinguish between the two and route your processing different places. At the beginning of your interrupt processing program, just PLA to get the pre-interrupt status register contents into the accumulator; PHA to get it back to the stack where it belongs; AND #$10 to isolate the B-flag; and BNE to the code applicable for the BRK case, falling through for the hardware-initiated interrupt.

An excellent way to familiarize yourself with interrupts in the Apple][system is to analyze the Apple System monitor code for handling interrupts. You can find it in the Apple][Reference manual. The old monitor (pp155-171) is probably a bit easier for a beginner to analyze than the Autostart monitor (pp136-154).

The RESET form of interrupt transfers control to the location specified in $FFFC,$FFFD. Just start tracing the monitor's code there. The Applesoft interpreter initiates '&' processing with a machine-language instruction (BRK); your analysis must begin with control transferred to the address in $FFFE,$FFFF.

11.6 Interrupts for the Quasi-BASIC Programmer

As a programmer who wants to program in machine language only as an adjunct to BASIC programming, you probably don't want to get deeply into interrupt programming, but you might like to intercept and change what happens when certain kinds of interrupts occur.

The vectored pointers built into the Apple's microprocessor architecture in $FFFA through $FFFF are located in the ROM area of memory. Since this is Read Only Memory you can't change them, unless you have a language card or equivalent and are willing to live with a non-standard, modified monitor.

What is a BASIC programmer to do? Fear not, Wozniak simplified procedures when he wrote the Apple monitor. All the important interrupt-related activities are available to you in one convenient and well-documented area of memory — the Monitor Special Locations on page 3 of RAM memory.

Chapter XII
The Apple Keyboard Input Buffer Memory Page Two ($0200-$02FF) & The Getln System Of Input Associated With It

12.1
Introduction to the Keyboard Input Buffer and Getln

12.1.1 Keyboard Input in the Default Case

The default method of entering information into the Apple is via the keyboard on a line-by-line basis, with new information being entered below previous inputs and outputs. As it is entered, each character is echoed back via COUT, and becomes visible on the display screen at the current cursor position. This mode of input continues until the end-of-line 'RETURN' key is pressed. This signifies that the line of input has been completed and terminates the current input operation.

Certain special keys (such as the arrow and escape keys) are handled differently. These special keys allow you to edit. They enable you to: erase or modify the characters that have been input, recover characters that have previously been deleted but are still held in a special buffer, move the cursor to another area of the screen, read characters from the screen, or perform other functions that temporarily preempt control over the system.

When entry works its way down to the extreme bottom of the display screen, new data is still entered at the bottom of the screen but all information above it is shifted upward, line by line, until the oldest information disappears off the top of the screen.

The heart of the system that permits this kind of operation is the use of memory page two ($0200-$02FF = decimal 512-767) as a 256-byte input buffer. This buffer is called the keyboard input buffer and is given the symbolic name 'KEYIN', under the control of the monitor 'GETLN' subroutine.

KEYIN is used primarily by a monitor subroutine 'GETLN'. Its main function is, as the name implies, to GET date in LINE-sized chunks. When GETLN (and its ancillary routines in the Apple system monitor) get information from the keyboard they put it character-by-character into the KEYIN buffer.

When this method of input is used, the computer does not accept each keystroke as a complete entry to be acted upon at entry. Instead, it accepts the data provisionally, stores it, displays it, and gives the user an opportunity to examine it. Then the user can correct the entry before giving it final stamp of approval by pressing 'RETURN'.

12.1.2 Concept of Operation

The skeleton of 'GETLN's concept of operation is quite simple:

1. At 'GETLN' the X-register is initialized as an index, i.e. its value is set to 0.

2. A character is read from the keyboard into the single-byte hardware keyboard data input location $C000.

3. The index (X-register) value is not changed when the initial (zero-th) character is entered. But as each additional character is received, the number in the X-register is increased by 1. Thus it becomes a counter of the number of characters received in the current line of input (less one).

4. At 'ADDINP' the character read from the keyboard is stored into page 2 at location $0200, indexed by the contents of the X-register. When X is 0, the character goes to $200; when X is 1, the character goes to $201, etc. Thus the individual characters of the input line are stored sequentially in $200, $201, $202, $203, etc.

5. A 'RETURN' ends this process. The ASCII code for 'RETURN' ($8D or decimal 141) is entered into the buffer and becomes a flag to signify the end of the line and of the character string that represents it in the buffer. Putting this character into the buffer also jacks the value of the X-register up to a full count of the characters in the line (less the 'RETURN', which is considered as an end-of-line flag).

6. The information is used and the process starts over with X being reinitialized to zero.

7. The size of the KEYIN buffer (one memory page = 256 memory locations) restricts the length of text-line-oriented inputs to 255 characters. This is seldom much of a practical restriction. Other considerations make it a useful rule of thumb to keep a single input terminated by a 'RETURN' down to six full 40-character screen-lines of text.

This standard system of handling input is used

for all normal inputs to the scrolling screen display, whether the inputs are made at the monitor level, in Applesoft BASIC, in Integer BASIC or in machine- or assembly-language programming.

In practice, a number of embellishments (see section 12.2) are added to the skeleton process described above.

12.1.3 Variations on the Theme of Keyboard Input

Though the system just described is the default case, it is perfectly possible to get information into the Apple, even from the keyboard, without going through this process.

The information that goes into the KEYIN buffer is thoroughly analyzed and tested by GETLN and associated subroutines. Various alternative courses of action, most of them associated with error detection and/or editing the input, may be triggered by commas or other delimiters, or by special key combinations such as CTRL-X, or ESC followed by some other key.

These services are usually valuable, but the advanced programmer may want to bypass them in special cases.

One way to bypass this method of input is to use the Applesoft BASIC 'GET' statement. It GETs characters one at a time without using either GETLN or KEYIN.

The popularity of the plethora of 'input-almost-anything' routines is a sure indication that even when you are doing line-oriented input some of the GETLN-associated services can get in the way.

You can use routines written by others, or you can do your own programming.

The full services provided by the monitor and the GETLN/KEYIN pattern of operation may be considered as one end of a spectrum of possibilities. The other end of the spectrum goes directly to the hardware of the computer with absolutely no monitor firmware or other software support. In the middle of the spectrum you can use some of the features and services of the GETLN/KEYIN family, bypassing only those that get in the way.

At the direct-contact-with-the-hardware end of the spectrum you may use a machine-language routine, or PEEK directly at the single-byte hardware keyboard input register (address $C000 = decimal 49152 or -16384). This technique can be particularly useful in situations where you wish to check for user-input without interrupting ongoing processing.

Warning: If you get input this way don't forget to strobe the keyboard at $C010 = decimal 49168 or - 16368. Strobing is accomplished by any memory access to that location and hence can be accomplished by a PEEK, a POKE, or a machine-language instruction that refers to the location. Strobing $C010 changes the sign bit of the byte in $C000, thus giving you the opportunity to test whether you have already read the byte currently in that location.

12.2.1 Survey of Services Provided by GETLN

GETLN offers many useful services routinely to the Apple II programmer. Its main characteristics are:

1. When called, GETLN first prints a prompting character. This character identifies the system (software) that is awaiting input and notifies the user that the system is indeed asking for data. The major prompts used by the Apple II include:

   '*' indicates that the system monitor is awaiting user input

   '>' indicates that the Integer BASIC interpreter is awaiting your command input

   ']' indicates that the Applesoft BASIC interpreter is awaiting your command input

   '!' indicates that the mini-assembler built into the system monitor (but not into the autostart version of the system monitor) is awaiting your command input, and

   '?' indicates that a running program is awaiting data input needed to continue its current problem-solution

2. As you type in characters, they are printed on the screen and the cursor moves accordingly. The characters will even scroll upward if you reach the end of the screen without issuing a carriage return. Finally, when you press the RETURN key the entire line is sent off to the system that requested it, and the system reacts accordingly. It might issue another prompt indicating that further input may be required.

3. After the prompt is printed, GETLN drops into a nested set of subroutines that include NXTCHAR, RDCHAR, RDKEY, and either KEYIN or ESC1. Each of these routines has its own information analysis and processing responsibilities.

4. Initially the system drops down as far as the RDKEY subroutine, which changes the character at the current input-output location on the display screen to a blinking condition.

5. RDKEY calls upon its subordinate KEYIN to continuously scan the hardware keyboard data input location $C000, testing for the presence of a one bit in its highest-order position. This is the hardware's method of making a positive indication that a key has been depressed and that its coded value is available in $C000.

6. When KEYIN detects a one bit in the highest-order position it feeds the new character into a processing cycle and issues a keyboard strobe. This strobe changes the high-order bit to a zero to indicate that that particular character input is being processed and is not a new character to be entered into the processing cycle.

7. KEYIN also performs two ancillary actions. It restores to an unblinking condition the character modified by the RDKEY routine to create a blinking cursor. It also counts up the random number field during the time that it is repeatedly testing for the presence of a new keystroke. It thereby increments a number that can later be used by other parts of the program or by user programs as a pseudo-random number.

8. When KEYIN is finished it bounces control back up the hierarchy past RDKEY to RDCHAR with the newly accepted character in the A-Register. RDCHAR tests it to see if the character was an ESCAPE character. If so, it passes processing to the escape processor ESC1. If not, it bounces control back upward to NXTCHAR.

9. ESC1 actually gets control after an ESCAPE is detected. Depending upon what that character is, ESC1 calls the appropriate scroll window service routine. The RTS at the end of the scroll window service routine returns control to RDCHAR at its normal entry point.

10. ESC1 recognizes eleven escape codes, eight of which are pure cursor moves, which simply move the cursor without altering the screen or input line, and three of which are screen clear codes, which simply blank part or all of the screen. Thus ESC-A, ESC-B, ESC-C and ESC-D merely move the cursor right, left, downward or upward respectively for one position. ESC-E, ESC-F, and ESC-@ (or ESC-SHIFT-P) clear the screen from the current cursor position to the end of line, end of page and clear the screen respectively (the latter putting the cursor at the 'home' position in the top left corner of the screen). Each of these codes has a scope of effect that lasts for just one character past the escape. The Autostart ROM only has four very valuable additional escape modes, which remain in force as long as one of their keys is depressed, no matter how often. These codes are ESC-K, ESC-J, ESC-M and ESC-I which move the cursor right, left, down, and up respectively. They are arranged in a directional keypad on the keyboard so that their directions of movement will be obvious and natural.

11. With these functions performed, control passes back up from RDCHAR to NXTCHAR, the routine responsible for moving input from the accumulator into the input buffer and the top point in the character input loop. NXTCHAR does some checking itself for special conditions and some additional service processing.

12. For routine characters, NXTCHAR does little but move the character from the accumulator to a spot in the input buffer designated by the buffer pointer, which is the hardware X-Register. However, it does this only after checking the value of the character in the A-Register. If that value is $E0 or higher, the character is a lower-case character and NXTCHAR converts it to upper-case.

13. If the character tested by NXTCHAR is the retype key (the right arrow), it causes the X-Register to increment indicating an additional item moved into the buffer. However, what is moved is not the value of the key that was pressed, but the value of the character on the video screen (and hence in the screen output buffer), over which the cursor was moved in performing this activity. If the character is a backspace (the left arrow) the X-Register is decremented. The character code is not physically removed from either the input buffer or the screen display, it is just hidden and therefore unavailable.

12.2.2 The Routines

The GETLN family of routines are, for the most part, one long routine with many alternate

entry points that are given different names. Entry at any particular routine means that you will automatically drop through and execute routines further down the list and receive the services associated with them.

Notice that GETLN itself is the second routine name in this list. It is preceded by GETLNZ, which performs a carriage return before dropping into the GETLN procedure.

Also notice that near the end of the list we get into the ESCAPE processing portions of the family, which are activated only when the 'ESC' key has been pressed to initiate special editing procedures.

1. Routine Name: GETLNZ
Start Locn : $FD67(decimal 64871 or -665)

Register Conditioning at Entry:
A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:
BASL,H : Don't care
CH : Don't care
CV : Line where input is to occur

Conditions on Return:
A-Register : Contains ASCII code for 'RETURN' ($8D)
X-Register : Contains number of characters read before 'RETURN'
Y-Register : Contains contents of WNDWDTH
CH : Contains 0
CV : Contains current line number
BASL,H : Contains memory address for CV, WNDWTH

Window line is blank to the right of the end of the echoed input

Results:
CR is written, scroll takes place if appropriate.
Prompt character is written through COUT
Keyboard is read character by character. Each character is placed at $0200,X and X is incremented by 1.
Each character is "echoed" to the screen at cursor position and then cursor is advanced.
On reading a 'RETURN', control is returned to calling program

Description:
Output a carriage return and execute GETLN

2. Routine Name: GETLN
Start Locn : $FD6A(decimal 64874 or -662)

Register Conditioning at Entry:
A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:
BASL,H : Line address at which input to begin (in scroll window)
CV : Line where input is to occur (compatible with BASL,H)
CH : Where on line prompt is to be placed

Conditions on Return:
Same as for GETLNZ

Results:
Same as for GETLNZ except for initial CR.

Description:
Print the prompt character and initialize X-reg for indexed storage of the input characters into the input area. Execute NXTCHR.

3. Routine Name: NXTCHAR
Start Locn : $FD75 (decimal 64885 or -651)

Register Conditioning at Entry:
A-Register : Don't Care
X-Register : 0 for data to start at $200
Y-Register : Don't care

Monitor Parameter Conditioning:
BASL,H : Compatible with CV, pointing in window
CH : Where echoing of keyboard input is to start
CV : Compatible with BASL,H; pointing in window

Conditions on Return:
Sames as for GETLNZ

Results:
Same as for GETLN

Description: Top point in character input loop. Call RDCHAR to get character into A-reg. On return A-reg tested for presence of CTRL-U (Right Arrow). If found, A-reg loaded from ((BASL),Y), a location in low-res screen refresh memory, assuming Y-reg holds same value as CH.

If A-Reg value > = $E0, convert lower case letter to upper case by AND with $DF, and store from A-reg to KEYIN buffer.

If character is 'RETURN', call monitor routine CLREOL to clear to end of line with blanks. Then conditional branch transfers control to COUT so RTS xit of COUT will return control to the calling program w/ X-reg, indicating input character count ± 1.

If character is not 'RETURN', transfer control to NOTCR for display on output device, and/or for interpretation with regards to control character affecting input line.

3a. Routine Name: CAPTST
(A portion of NXTCHAR which you may wish to

deactivate)

Start Location: $FD7E (decimal 64894 or -642)

Registers and Parameter Conditioning at Entry: As for NXTCHAR

Description: This is the notorious capitalizer for Apple keyboard input. It tests to see if contents of A-Reg > $DF and if so ANDs it against $DF to make the character upper case. Can be replaced by NOPs to defeat this action.

Note: If you treat this as a subroutine in its own right and enter at this point without making change you get same effect as NXTCHR except ability to get input by scanning cross screen with right arrow key has been bypassed.

4. Routine Name: NOTCR

Start Locn : $FD3D(decimal 64829 or -707)

Register Conditioning at Entry:

A-Register : Character to be outputted via COUT
X-Register : IN,X points to character of interest
Y-Register : Don't care

Monitor Parameter Conditioning:

BASL,H : Don't care
CH : Don't care
CV : Don't care

Description: IN,X points to character of interest. Save current setting of INVFLG on stack and set INVFLG to $FF so character echoed to screen will be white on black. Send character in A-reg to COUT. On return from COUT restore INVFLG from stack. If character pointed to by IN,X is backspace go to BCKSPC. If character pointed to by IN,X is CTRL-X, goto CANCEL. Otherwise test X-Reg to see if KEYIN buffer full or almost full. If value of X-Reg > 247 call BELL to signal user KEYIN is almost full. Whether or not bell is sounded go to NOTCR1.

5. Routine Name: NOTCR1

Start Locn : $FD5F (decimal 64863 or -673)
Not recommended for use as a separate subroutine

Description: Increment X-reg. If this results in overflow to 0, then go to CANCEL; otherwise go back to NXTCHR.

6. Routine Name: CANCEL

Start Locn : $FD62 (decimal 64866 or -670)

Register Conditioning at Entry:

A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:

BASL,H : Don't care
CH : Don't care
CV : Line where input was to occur

Description: Print back-slash through COUT to indicate cancellation of line being inputted. Start new line and throw away inputted data by going to GETLNZ for reinitialization w/o using data.

7. Routine Name: BACKSPC

Start Locn : $FD71 (decimal 64866 or -655)

Register Conditioning at Entry:
As for NEXTCHR

Monitor Parameter Conditioning:
As for NEXTCHR with X-Reg pointing to deleted character

Description: On entry backspace character has already been printed through COUT and cursor moved back. If X-Reg is zero goto GETLNZ, otherwise decrement X-Reg and go to NEXTCHR

8. Routine Name: RDCHAR

Start Locn : $FD35 (decimal 64821 or -715)

Register Conditioning at Entry:

A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:

BASL,H : Line where input is to occur; in window; compatible with CV
CV : Line where input is to occur; in window; compatible w/BASL,H
CH : Horizontal posn in scroll window where cursor will be indicated

Conditions on Return:

A-Register: Contains value of key pressed
X-Register: No change
Y-Register: Contains contents of CH
BASL,H CV CH: Changed only if Escape Key function utilized

Results:

Screen character at cursor position (BASL),(CH) will be set to blinking until a key is pressed.
If the ESCape key is detected, appropriate escape routine will be called.
Cursor right arrow (control-U) will be returned to the calling program, not the contents of the screen at the cursor.
Cursor left arrow (control-H) will be returned to the calling program.
Cancel line input (control-X) generates no special action; service is not defined.
'RETURN' generates no special action because rest of KEYIN is not called.
Characters read from the keyboard will not be stored in memory page 2.
After the character is read, the blink will be turned off at the cursor position, but the key

just read will not be echoed to the screen, nor will the cursor be advanced.

Description: Call RDKEY to get next character placed into A-Reg. If on return escape key has been pressed, go to escape function relevant to monitor in use. For Autostart monitor, go to ESC and thence through ESCNEW to ESC1. For old monitor, go to ESC and thence to ESC1. After any request escape funtions performed control returns to REDCHAR as if there had been no interruption.

9. Routine Name: RDKEY

Start Locn : $FD0C (decimal 64780 or -756)

Register Conditioning at Entry:

A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:

BASL,H : Line where input is to occur; in window; compatible with CV
CV : Line where input is to occur; in window; compatible w/ BASL,H
CH : Horizontal position where cursor will be shown by blinking

Conditions on Return:

A-Register : Contains character read from keyboard
X-Register : Not changed
Y-Register : Contains contents of CH
CV : Is used to calculate the new line
BASL,H : Reflects the recalculated address
CH : Not changed

Results:

The character on the screen at the cursor position is set to blinking.

KEYIN routine is given control via (KSWL) for physical reading of the keyboard.

Return (RTS) in KEYIN returns to the caller of RDKEY, not to RDKEY.

Description: Gets next input character into A-Reg by doing indirect jump via KSWL,H, which normally points at KEYIN. (SPecifically at location specified by BASL,H and CH). Change that character in memory to blinking to indicate current cursor position. Return is to caller of RDKEY not to RDKEY itself.

10. Routine Name: KEYIN

Start Locn : $FD1B (decimal 64795 or -741)

Register Conditioning at Entry:

A-Register : value to be stored in screen area at (BASL),Y to remove blink after key press. (Normally last previous character entered.)
X-Register : Don't care
Y-Register : used to store A-reg into screen area to remove blink at (BASL),Y.

Monitor Parameter Conditioning:

BASL,H : Used as described with A and Y registers above
CH : Don't care
CV : Line where input is to occur

Conditions on Return:

A-Register: Contains input from keyboard register
Other Registers & Monitor Parameters: Unchanged

Results:

Input from keyboard register appears in A-Register.

Description: Gets next input key from keyboard hardware. Reads keyboard input buffer over and over again until presence of $80 bit shows that a character has been read. In this case, keyboard input buffer refers to memory page 2 buffer (screen display) rather than $C000. The sign flag is set or not set by checking the status of sign at $C000, which tells whether a key has been pressed. If sign is positive, loop back to KEYIN: if negative, pick up value at $C000 and strobe $C010 to reset sign of $C000 back to positive. Ancillary actions: Count up random number field (ignoring overflow). Restore blinking cursor value modified by RDKEY by storing A-reg at (BASL),Y before $C000 read into A-Reg.

11. Routine Name: ESC

Start Locn : $FD2F (decimal 64815 or -721)

Register Conditioning at Entry:

A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:

BASL,H : Don't care
CH : Don't care
CV : Line where input is to occur

Description: Enter from RDCHAR when ESC keypress detected. Calls RDKEY to get entry after ESC to A-Reg then calls ESC1 (old monitor) or ESCNEW (Autostart monitor) to perform requested function and return RDCHAR.

12. Routine Name: ESCNEW (Autostart Monitor Only)

Start Locn : $FBA5 (decimal 64421 or -1115)

Register Conditioning at Entry:
A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:
BASL,H : Don't care
CH : Don't care
CV : Line where input is to occur

Description: Supports cursor movement without data transfer ESC I,J,K or M. If next key pressed is one of them, do ESC A,B,C or D, which is relevant by calling ESC1. On return to ESCNEW, call RDKEY again and repeat process. If key is not I,J,K or M, then JMP rather than JSR to ESC1 so return is to caller of ESCNEW rather than ESCNEW.

13. Routine Name: ESC1
Start Locn : $FC2C (decimal 64556 or -980)

Register Conditioning at Entry:
A-Register : Don't care
X-Register : Don't care
Y-Register : Don't care

Monitor Parameter Conditioning:
BASL,H : Don't care
CH : Don't care
CV : Line where input is to occur

Description: Supports cursor movement without data transfer ESC A,B,C or D (and in autostart monitor with aid of ESCNEW also ESC I,J,K or M). Also ESC E (clear to end of line) ESC F (clear to end of window) and ESC @ (home). When called, contents of A-reg (and the condition that carry is 'set') indicate action to be taken. If one of the above ESC characters, conditional branch to appropriate scroll window service routine to take appropriate action. Otherwise ignore and RTS.

12.2.3 Replacement of KEYIN

One useful means of modifying the input system, yet keeping the GETLN services, is to write and use a replacement for KEYIN, then substitute its calling location for that of KEYIN at KSWL,H so that it will be excuted whenever KEYIN would be under normal circumstances.

Preferences to input from an external device in a particular slot may already have altered the address of KSWL,H, and you probably want to return to the condition the system was in. Therefore it is a good idea to save the current contents of KSWL,H before replacing them by your KEYIN replacement and restore them after it has been used. If you use DOS while the replacement is in use, expect confusion. DOS uses KSWL,H for its own purposes and periodically restores them to appear the way it thinks they should be, regardless of their current contents.

If you write a replacement for KEYIN it sould meet the following requirements:

1. A-Register:
   Store the A-Reg at (BASL),Y, then load from whatever source is to be used.
2. X-Register:
   Must be same on exit as on entrance.
3. Y-Register:
   Must be same on exit as on entry (unless you are protected against escape key processing, in which case not required). Note use of Y-Reg with A-Reg above.
4. CH, CV and BASL,H:
   Used for echoing keyboard replacement routine input, so either leave them alone if echoing is not required, or manipulate them in an appropriate manner to reflect your echoing requirements.

12.2.4 Automatic Capitalization in GETLN

Keyboard input of lower case letters is automatically converted to capitals by CAPTST as described in 12.2.2 3A . It may be defeated in a variety of ways.

Chapter XIII
The Monitor and DOS Vector Page

Page 3 is the last page in the first K of Apple memory, an area devoted to system support activities. Its main function, which takes up only 3/16ths of the available space, is to provide convenient interfaces to system firmware in the Monitor and DOS. The remainder of its space is available for user programming.

13.1
The Monitor Special Locations in Memory Page 3

The top $10 (decimal 16) memory locations are used by the monitor as special locations for the newer Autostart version of the monitor, and five less for the older non-autostart version. Figure 13.1A shows the allocation of locations used with the old monitor; figure 13.1B shows those for the new Autostart version.

Figure 13.1A

Memory Page 3 - Monitor Special Locations Used with (Non-Autostart) Monitor DOS not Activated

Hex	Decimal	Use
$3F5 $3F6 $3F7	1013 1014 1015	Holds a 'JuMP' instruction to the subroutine at $FF65 that handles '&' commands. This default is often reset by sophisticated users.
$3F8 $3F9 $3FA	1016 1017 1018	Holds a 'JuMP' instruction to the subroutine that handles 'USER' (CTRL-Y) commands. Default is set for $FF65 (MON) This is normal re-entry to top of monitor.
$3FB $3FC $3FD	1019 1020 1021	Holds a 'JuMP' instruction to the subroutine that handles Non-Maskable Interrupts (NMI's).Default set to $FF65 (MON) This is normal re-entry to top of monitor.
$3FE $3FF	1022 1023	Holds the address of the subroutine that handles Interrupt Requests(IRQ's).Same default(FF65) to MON(top of monitor).

Figure 13.1B
Memory Page 3-Monitor Special Locations Used with Autostart Monitor DOS not Activated

Hex	Decimal	Use
$3F0 $3F1	1008 1009	Holds the address of subroutine that handles machine-lan- 'BRK' requests.Default:$FF58(Same as pressing 'RESET key).
$3F2 $3F3	1010 1011	Soft Entry Vector. Points to $9DBF.
$3F4	1012	Power-up Byte. Value: $38.
$3F5 $3F6 $3F7	1013 1014 1015	Holds a 'JuMP' instruction to the subroutine at $FF65 that handles '&' commands. This default is often reset by sophisticated users.
$3F8 $3F9 $3FA	1016 1017 1018	Holds a 'JuMP' instruction to the subroutine that handles 'USER' (CTRL-Y) commands. Default is set for $FF65 (MON). This is normal re-entry to top of monitor.
$3FB $3FC $3FD	1019 1020 1021	Holds a 'JuMP' instruction to the subroutine that handles Non-Maskable Interrupts(NMI's).Default set to $FF65 (MON). This is normal re-entry to top of monitor.
$3FE $3FF	1022 1023	Holds the address of the subroutine that handles Interrupt ReQuests (IRQ's).Same default($FF65)to MON-top of monitor.

13.2
The DOS Vector Table in Memory Page 3 ($3D0-$3FF) (Includes Monitor Special Locations)

When the DOS is active, as it is in most Apple systems most of the time, the Monitor Special Locations block on memory page 3 is expanded to include jumps and subroutines that are important to interface user programs with DOS. Figure 13.2A is a detailed guide to the use of this expanded block of memory locations.

Figure 13.2A
Memory Page 3 - DOS & Monitor Vector Table ($3D0-$3FF)
(DOS Activated)

Hex	Decimal	Use
$3D0 $3D1 $3D2	976 977 978	Holds a JuMP to DOS Warmstart Routine at $9DBF. Re-enters DOS without discarding current BASIC program and without resetting MAXFILES or other DOS Environmental Variables.
$3D3 $3D4 $3D5	979 980 981	Holds a JuMP to the DOS Coldstart Routine at $9DBF. Re-initializes DOS as if it were re-booted, clearing the current BASIC file and resetting HIMEM.
$3D6 $3D7 $3D8	982 983 984	Holds a JuMP to the DOS file manager subroutine at $AAFD to allow a user-written assembly-language program to call it.
$3D9 $3DA $3DB	985 986 987	Holds a JuMP to the DOS Read/Write/Track/Sector (RWTS) routine at $B7B5 to allow user-written assembly-language programs to call it.
$3DC through $3E2	988 994	A short subroutine that locates the input parameter list for the file manager to allow a user-written program to set up input parameters before calling RWTS.
$3E3 through $3E9	995 1001	A short subroutine that locates the input parameter list for RWTS to allow a user-written program to set up input parameters before calling RWTS.
$3EA $3EB $3EC	1002 1003 1004	Holds a JuMP to the DOS subroutine at $AA51 that reconnects the DOS intercepts to the keyboard and screen data streams.
$3EF $3F0 $3F1	1007 1008 1009	Holds a JuMP to the routine at $FF59 that handles machine-language 'BRK' requests. Overall effect is the same as pressing the 'RESET' (or 'CTRL'RESET') key.
$3F2 $3F3	1010 1011	Soft Entry Vector. Points to $9DBF.
$3F4	1012	Power-up Byte. Value: $38.
$3F5 $3F6 $3F7	1013 1014 1015	Holds a 'JuMP' instruction to the subroutine at $FF65 that handles '&' commands. This default is often reset by sophisticated users.
$3F8 $3F9 $3FA	1016 1017 1018	Holds a 'JuMP' instruction to the subroutine that handles 'USER' (CTRL-Y) commands. Default is set for $FF65 (MON) This is normal re-entry to top of monitor.
$3FB $3FC $3FD	1019 1020 1021	Holds a 'JuMP' instruction to the subroutine that handles Non-Maskable Interrupts (NMI's). Default is set to $FF65. (MON) This is normal re-entry to top of monitor.
$3FE $3FF	1022 1023	Holds the address of the subroutine that handles Interrupt ReQuests(IRQ's).Same default($FF65) to MON-top of monitor

13.3
Page 3 Space Available to Users ($300-$3CF) and How It Is Typically Used for Machine-Language Programming

The remainder of memory page 3, $C0 (decimal 192) bytes, is not a trivial amount of memory. It is not needed for general support of Apple system hardware or firmware, but it is too large to be ignored and wasted. Yet it constitutes

only about 2% of the RAM memory space in a 48K Apple and it is located in the first K of memory with other memory that supports the Apple firmware. It is isolated from the large block of memory space set aside for BASIC programs by the 2nd K of memory: the text and low-res graphics area, another area set aside for system and firmware support.

Unfortunately this separation makes it impractical for the Apple system to make this relatively small and isolated block of memory space a part of the general-usage space allocated by the Applesoft or Integer BASIC interpreters. Thus it cannot be used as freely as one might like as part of BASIC programs.

However, $C0 (decimal 192) bytes is a convenient-sized block for small blocks of machine-language utility code that is often needed in conjunction with BASIC programs. This space is quite frequently used as a home for machine-language code used in a BASIC environment.

For example, it might be used to hold a printer-driver program used to supplement or modify the standard printer driver built into a printer interface card. Or it might be used to contain a keyboard filter program used for some special modification of keyboard input procedures. It could hold a special &-interpreter to enable the & key to be used for a special purpose.

Memory space used in this area does not reduce the amount of space available to BASIC programs and their variables, and does not require changes to either HIMEM or LOMEM, the limits of BASIC program memory allocation.

Chapter XIV
Test and Low-Resolution Graphics Display Memory Pages 4-7 and 8-11 ($400-$7FF and $800-$0BFF)

14.1
Text Output to the Screen — Introductory Frame of Reference

In the text mode the Apple can display 24 characters of lines with up to 40 characters on each line. The lines are numbered from 0 (top of page) to 23 (bottom of page). The positions within a line vary from 0 (left edge) to 39 (right edge).

Each character on the screen represents the contents of one memory location. The area of memory used for the primary (default) text page extends $400 bytes (1024 decimal) from location $0400 (1024 decimal) to location $07FF (2047 decimal). A secondary text display page of the same size is also available. The secondary text page extends from location $0800 (2048 decimal) to the location $0BFF (3071 decimal).

In most BASIC programs, you use and display text only from the memory locations associated with text page 1. All normal BASIC and monitor commands which generate printed screen output print that output to text page 1.

However, you can arrange to pass output to text page 2 by various indirect means, such as moving information there from text page 1 or POKEing data there directly. This may be done if page 2 is displayed or not. If data is put into page 2 while it is being displayed, it appears character-by-character as it is added, just like normal BASIC output. If data is put into page 2 while it is not being displayed, there is no visible change in the display screen while the data is being put into the display memory. However, if you give the command POKE −16299,0, the display changes instantly from that of the data in the text display buffer of page 1 to that of the data written into the memory locations associated with the previously invisible page 2. Thus a whole page of text may be placed onto the screen in the blink of an eye, providing that you are willing to do the appropriate set-up work in advance. POKE −16300,0 may be used to change the display instantly back to text page 1.

The hardware and the monitor software of the Apple II are organized so that the text page 1 buffer [memory locations $0400 (1024 decimal) to $07FF (2047 decimal)] normally operate as a 'scrolling' display area. This means that routinely each new character of text enters at the bottom line of the screen. When you reach the end of entry of that line of information, the carriage return entered at the end of the line causes all lines on the screen to shift upward by one line to provide space for the entry of a new line at the bottom of the screen. The cursor, which shows where on the display the next character will be entered, moves to the left of the now-blank bottom line.

The area of the screen where scrolling takes place during input may be limited to only a portion of the total display. This may be done by establishing 'window' boundaries — left, right, top and bottom limits for the scrolling area and normal input and output associated with it. These boundaries may be set up by POKEing operations.

In addition to normal entries at the bottom of the screen (or the bottom of the window) the Apple also makes provision for moving the current output or input to any desired random location within the scrolling area through use of vertical and horizontal tabbing functions.

In Applesoft BASIC, the system-specific commands VTAB and HTAB can be used to move the current printing location to any desired spot on the page within the current scrolling limits. For example, VTAB 5: HTAB 7: PRINT "+" will position the current printing position to line 5 (one quarter of the way down the screen from the top) and to horizontal printing position 7 (one fifth of the way across the screen from the left edge), then at that location print the symbol '+'.

Output may be inserted into any screen location, regardless of whether that screen location is inside or outside the scrolling window, by directly storing the information. This is most frequently done by means of POKE operations.

To use this method of entry you must first learn something about how the characters are stored in the computer memory and what locations in memory correspond to what locations on the screen.

14.2
Representation of Text-Characters Inside the Apple — The ASCII Code

Each text-character on the display-screen, when the Apple II is in text mode, is determined by the contents of one memory location. The text-character to be displayed is determined by the

ASCII (American Standard Code for Information Interchange) symbol associated with the bits in that memory location.

Actually the Apple II does not use the full United States national standard ASCII characters, but instead, a modified subset and superset of ASCII. This set is built around 64 characters: 26 upper-case letters, 10 digits and 28 special characters (punctuation, etc.). These characters would require only six bits to represent ($2^6 = 64$). However, the Apple also uses a block of 32 control characters. These characters can be entered from the keyboard by depressing the special control or CTRL key at the same time you depress the key of the relevant alphabetic character. The use of these control characters varies widely. For example, CTRL-G will ring a bell or sound a beeper. A CTRL-D in a PRINT statement will route the output associated with it, not to the screen, but to disk storage. If the Apple II is in its monitor mode a CTRL-B will cause the system to enter BASIC. BASIC;, a CTRL-C, will allow reentry to a BASIC program which has previously been interrupted without loss of data values, intermediate results and/or variable names which were current at the time the system exited BASIC;.

The U.S. national standard ASCII is a seven-bit code with provision for all these characters plus provision for a lower-case alphabet and some extra less frequently used special punctuation characters.

The Apple II as supplied by its manufacturer does not support the use of lower-case letters. The Apple II has neither provision for entering them from the keyboard nor of displaying them on the screen. Thus the 7-bit ASCII code supported by the Apple provides only a subset of the national standard ASCII characters.

The lack of lower-case characters and the related incompatibility with United States national standards is considered by many to be a significant weakness or fault in the Apple II system. Many users find the lack of a lower-case capability intolerable, especially if they are interested in applications which involve text and word processing. Thus many secondary vendors now provide means for modifying the Apple II to get the missing characters.

Since the Apple II stores information in 8-bit bytes it has an extra bit available to support up to $2^8 = 256$ code characters. There is no U.S. national standard which specifies how extra code combinations such as these should be used. Apple uses them to support FLASHING and INVERSE-VIDEO display versions of its standard alphanumeric characters.

Thus the Apple II character set and code shown in figure 14.2A is a modified subset of ASCII lacking lower-case letters at the same time that it is a superset containing FLASHING as well as INVERSE-VIDEO versions of the standard characters.

Figure 14.2A(1)—ASCII Subset/Superset used by Apple][(no lower case)

	Inverse				Flashing				(Control)		Normal				(LowrC)	
Dec	000	016	032	048	064	080	096	112	128	144	160	176	192	208	224	240
Hex	$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	-	/	

Figure 14.2A(2)
Keys and Their Associated ASCII Codes

Key	Alone	CTRL	SHIFT	Both	Key	Alone	CTRL	SHIFT	Both
					A	$C1	$81	$C1	$81
					B	$C2	$82	$C2	$82
					C	$C3	$83	$C3	$83
					D	$C4	$84	$C4	$84
0	$B0	$B0	$B0	$B0	E	$C5	$85	$C5	$85
1 !	$B1	$B1	$A1	$A1	F	$C6	$86	$C6	$86
2 "	$B2	$B2	$A2	$A2	G	$C7	$87	$C7	$87
3 #	$B3	$B3	$A3	$A3	H	$C8	$88	$C8	$88
4 $	$B4	$B4	$A4	$A4	I	$C9	$89	$C9	$99
5 %	$B5	$B5	$A5	$A5	J	$CA	$8A	$CA	$8A
6 &	$B6	$B6	$A6	$A6	K	$CB	$8B	$CB	$8B
7 '	$B7	$B7	$A7	$A7	L	$CC	$8C	$CC	$8C
8 (	$B8	$B8	$A8	$A8	M	$CD	$8D	$CD	$8D
9)	$B9	$B9	$A9	$A9	N ^	$CE	$8E	$CE	$8E
: *	$BA	$BA	$AA	$AA	O	$CF	$8F	$CF	$8F
; +	$BB	$BB	$AB	$AB	P @	$D0	$90	$D0	$90
, <	$AC	$AC	$BC	$BC	Q	$D1	$91	$D1	$91
- =	$AD	$AD	$BD	$BD	R	$D2	$92	$D2	$92
. >	$AE	$AE	$BE	$BE	S	$D3	$93	$D3	$93
/	$AF	$AF	$BF	$AF	T	$D4	$94	$D4	$94
->	$88	$88	$88	$88	U	$D5	$95	$D5	$95
<-	$95	$95	$95	$95	V	$D6	$96	$D6	$96
					W	$D7	$97	$D7	$97
space	$A0	$A0	$A0	$A0	X	$D8	$98	$D8	$98
return	$8D	$8D	$8D	$8D	Y	$D9	$99	$D9	$99
escape	$9B	$9B	$9B	$9B	Z	$DA	$9A	$DA	$9A

14.3 How Screen Locations of Text-Characters Map Into Memory Locations and Vice-Versa

A complete page of text requires 24 x 40 = 960 characters or 960 bytes of memory. Since the Apple hardware memory-pages hold 256 bytes each, it requires slightly less than four memory-pages to hold a full screen of characters. Four

memory-pages are assigned to one text/low resolution screen page. (The few extra locations not needed for screen-display are allocated as scratch-pad memory space for use by peripheral I-O devices, which may be plugged into the eight 'slots' inside the Apple.)

The Apple system has two text/low-resolution-graphics pages. Page 1 occupies memory locations $0400-$07FF (1024-2047 decimal); Page 2 occupies $0800-$0BFF (2048-3071 decimal). The BASIC interpreter *always* routes its output to text page 1. Page 2 is not available unless you start your program with LOMEM: 3072 (or higher). Otherwise Applesoft will allocate the text/low-res page 2 as program storage space. Even if LOMEM is changed, it displays text-page 1 unless it has been overtly instructed to display information from page 2.

· Each text-page may be divided into eight text sub-pages organized as 'macro-lines'. (See figure 14.3A for the orgainzation of a single macro-line.

Figure 14.3A — Memory Layout of a Single Text/Low-Resolution 'Macro-Line'

<-First 40 charctrs-><-Second 40 charctrs-><-Third 40 charctrs-><-8->
(Top 1/3 screen) (Middle 1/3 screen) (Bottom 1/3 screen) 'slot' bytes

Note that the display output from a single macro-line will appear partially on the top ⅓ of the screen, partialy on the middle ⅓ of the screen, and partially on the bottom ⅓ of the screen as shown in figure 14.3B.) Each of these eight macro-lines occupies 128 bytes of memory (half a memory-page). 120 of those bytes are used to represent 120 displayable text-characters. The remaining eight are not displayable on the display-screen.

Each of these non-displayable bytes is assigned to a different one of the eight peripheral 'slots' in the Apple II to serve as a byte of 'scratchpad memory' for the I-O peripheral which may be plugged into that slot. Since there are eight sub-pages, each slot gets 8 bytes scattered, one byte each, at eight locations in what is otherwise a screen-display memory area.

The eight macro-lines, taken together as a group, may be thought of as a logical display eight characters (or lines) high by 120 characters (or print-columns) wide. You cannot view this logical display directly on a TV screen — it is too long and thin. To view it on a TV screen we break the 120-character macro-lines down into three 40-character lines. (See figures 14.3C and 14.3D) This gives us 3 times 8, or 24, lines of 40 characters each — a convenient layout for a TV display-screen. The three lines whose 40-byte

Figure 14.3B ----- Screen Display Layout of Low-Resolution 'Macro-Line' (Example shown: Macro-Line 0)

Line	
00	<----ML00 First 40 characters -------->
01	.....
02	.....
03	.....
04	.....
05	.....
06	.....
07	.....
08	<----ML00Second 40 characters -------->
09	.....
10	.....
11	.....
12	.....
13	.....
14	.....
15	.....
16	<----ML00 Third 40 characters -------->
17	.....
18	.....
19	.....
20	.....
21	.....
22	.....
23	.....

Figure 14.3C--Internal Structure of the Text/Low-Resoultion Graphics Display Area
— Eight Macro-Lines of 120 Characters Each (ML00-ML07)+8 bytes—
— Each Macro-Line contains 3 Text-Display Lines (DL00-DL23) —

ML00	<1st 40 chars =DL00>	<2nd 40 chars =DL08>	<3rd 40 chars =DL16><8 bytes>
ML01	<1st 40 chars =DL01>	<2nd 40 chars =DL09>	<3rd 40 chars =DL17><8 bytes>
ML02	<1st 40 chars =DL02>	<2nd 40 chars =DL10>	<3rd 40 chars =DL18><8 bytes>
ML03	<1st 40 chars =DL03>	<2nd 40 chars =DL11>	<3rd 40 chars =DL19><8 bytes>
ML04	<1st 40 chars =DL04>	<2nd 40 chars =DL12>	<3rd 40 chars =DL20><8 bytes>
ML05	<1st 40 chars =DL05>	<2nd 40 chars =DL13>	<3rd 40 chars =DL21><8 bytes>
ML06	<1st 40 chars =DL06>	<2nd 40 chars =DL14>	<3rd 40 chars =DL22><8 bytes>
ML07	<1st 40 chars =DL07>	<2nd 40 chars =DL15>	<3rd 40 chars =DL23><8 bytes>

representations are together in memory in a single text sub-page do not create display-lines which are adjacent to one another on the display screen.

Instead, the three 40-character packets in a text/low-resolution graphics macro-line display at locations on the screen that are eight lines apart; one in the top third of the screen, one in the middle third, and one in the bottom third of the screen. (See figure 14.3D.)

The conversion to 40-character-wide display format is accomplished by a 'wrap-around' process. However, instead of a single macro-line wrapping around itself twice (first at the 40th character and a second time at the 80th character) to form 3 adjacent lines, the *whole* 8 macro-line logical display wraps around as a single entity.

At the wrap-around point at the 40th character, macro-lines 0 through 7 wrap as a unit to lines 8

through 15. (Line 0 wraps around to line 8; line 1 wraps around to line 9; and so on through line 7 wrapping around to line 15.)

At the wrap-around point at the 80th character, lines 0 through 7 (which have already wrapped once as a unit to lines 8 through 15) wrap around again as a unit to lines 16 through 23. (Line 0/8 wraps to line 16; line 1/9 to line 17; and so on through line 7/15 which wraps around to line 23.) (See figure 14.3E)

```
|-----------------------------------------|
|Li |  Figure 14.3D -----  Screen Display  |
|in |  Layout of Low-Resolution 'Macro-Line'|
|e  |     (Example shown: Macro-Line 0)     |
|   |-----------------------------------------|
|00|<----ML00 First 40 characters  -------->|
|01|.....................................|
|02|.....................................|
|03|.....................................|
|04|.....................................|
|05|.....................................|
|06|.....................................|
|07|.....................................|
|08|<----ML00Second 40 characters  -------->|
|09|.....................................|
|10|.....................................|
|11|.....................................|
|12|.....................................|
|13|.....................................|
|14|.....................................|
|15|.....................................|
|16|<----ML00 Third 40 characters  -------->|
|17|.....................................|
|18|.....................................|
|19|.....................................|
|20|.....................................|
|21|.....................................|
|22|.....................................|
|23|.....................................|
|-----------------------------------------|
```

```
|---|-----------------------------------------------------------------|
|D L|                          Figure 14.3E                           |
|i i|              Screen Display Wrap-Around of Macro-Lines          |
|s n|   DL = Display Line: A physical location on TV screen           |
|p e|   ML = Macro-Line: A logical entity of 128 consecutive memory locations|
|---|-----------------------------------------------------------------|
|DL00|<- - - First 40 characters of ML00- - - - - - - - - - - - - - - ->|
|DL01|<- - - First 40 characters of ML01- - - - - - - - - - - - - - - ->|
|DL02|<- - - First 40 characters of ML02- - - - - - - - - - - - - - - ->|
|DL03|<- - - First 40 characters of ML03- - - - - - - - - - - - - - - ->|
|DL04|<- - - First 40 characters of ML04- - - - - - - - - - - - - - - ->|
|DL05|<- - - First 40 characters of ML05- - - - - - - - - - - - - - - ->|
|DL06|<- - - First 40 characters of ML06- - - - - - - - - - - - - - - ->|
|DL07|<- - - First 40 characters of ML07- - - - - - - - - - - - - - - ->|
|DL08|<- - - Second 40 characters of ML00- - - - - - - - - - - - - - - >|
|DL09|<- - - Second 40 characters of ML01- - - - - - - - - - - - - - - >|
|DL10|<- - - Second 40 characters of ML02- - - - - - - - - - - - - - - >|
|DL11|<- - - Second 40 characters of ML03- - - - - - - - - - - - - - - >|
|DL12|<- - - Second 40 characters of ML04- - - - - - - - - - - - - - - >|
|DL13|<- - - Second 40 characters of ML05- - - - - - - - - - - - - - - >|
|DL14|<- - - Second 40 characters of ML06- - - - - - - - - - - - - - - >|
|DL15|<- - - Second 40 characters of ML07- - - - - - - - - - - - - - - >|
|DL16|<- - - Third 40 characters of ML00- - - - - - - - - - - - - - - ->|
|DL17|<- - - Third 40 characters of ML01- - - - - - - - - - - - - - - ->|
|DL18|<- - - Third 40 characters of ML02- - - - - - - - - - - - - - - ->|
|DL19|<- - - Third 40 characters of ML03- - - - - - - - - - - - - - - ->|
|DL20|<- - - Third 40 characters of ML04- - - - - - - - - - - - - - - ->|
|DL21|<- - - Third 40 characters of ML05- - - - - - - - - - - - - - - ->|
|DL22|<- - - Third 40 characters of ML06- - - - - - - - - - - - - - - ->|
|DL23|<- - - Third 40 characters of ML07- - - - - - - - - - - - - - - ->|
|---|-----------------------------------------------------------------|
```

14.4
Controlling What Appears Where on the Display-Screen

You may make a particular character display at a particular location on the screen by putting the correct combination of bits into the particular byte of memory which represents that part of the screen. This can be done by using conventional BASIC output routines or by bypassing those routines and injecting the desired output directly into the desired memory location.

If information goes into the area currently switched-on to display, the symbols appear as soon as the bits are placed in memory. If the information goes into the text page not currently selected for display, it (and the entire remainder of the page) will remain invisible until that display-page is switched on to replace the current page.

The ability to put information into display-page memory at any time or at any rate while that page is switched off and thus invisible. But to make it appear instantly when a single switch is thrown, can be the basis of some interesting and valuable visual effects. With graphics, it can be the basis of animation.

Since the Apple monitor and interpreter software do *not* support printing onto text page 2, you can get information to the page 2 display area by either of two methods:

1. Use conventional output techniques such as BASIC Print statements to feed the display information to text page 1 (whether or not text page 1 is currently being used to display information), then move the information to text page 2; or
2. POKE or use the Apple system monitor to put the desired display bits directly into the location in page 2.

   Once the information to be displayed is in the desired screen buffer area the appropriate soft-switches may be set to switch on and thus make visible the contents of any of the four display buffer pages:

   a. Text/Low-Resolution Graphics Page 1
   b. Text/Low-Resolution Graphics Page 2
   c. High-Resolution Graphics Page 1
   d. High-Resolution Graphics Page 2

This switching is automatically accomplished if you access the appropriate soft-switch or soft-switches in a way which causes that location in memory to be addressed. When programming in BASIC, a POKE to the relevant soft-switch location (expressed as a decimal address) will do the job quite conveniently.

14.5
The Low-Resolution Graphics Mode

In the low-resolution graphics mode the Apple II uses the same 1024 byte screen buffer areas ($0400-$07FF or $0800-$0BFF) as in the text mode. Each of these buffers can store *either* low-resolution pictorial information or text, but not both at the same time.

The Apple II does, however, provide a means of splitting the screen so that the top 5/6ths of the screen (lines 0-19) are displayed as low-resolution graphics and the bottom 1/6th of the screen (lines 20-23) are displayed as text.

The mixed-mode (5/6ths low-resolution graphics and 1/6th text) can be implemented by making sure the following combination of soft-switch states are activated:

$C050 (-16304) — Display graphics on at least part (and perhaps all) of the screen.

$C053 (-16301) — Mix text (bottom 4 lines only) and graphics display (rather than an all-graphics display).

$C057 (-16297) — Display graphics in Lo-Res mode (rather than Hi-Res mode).

This 5/6ths and 1/6th split is fixed and immutable. It cannot be moved up or down the screen. The same split is available for mixed display of text and hi-resolution graphics even though the two displays use widely-separated areas of display-memory.

In the low-resolution graphics mode each of the 960 character positions is displayed not as an ASCII character, but as two colored blocks stacked one on top of the other. Thus the screen display becomes 48 (24 × 2) blocks high by 40 blocks wide. If you use the graphics-text split mentioned above, the graphics portion is 40 blocks high by 40 blocks wide and there is space for 4 lines of text as well.

Each block can be any of sixteen colors. On a black-and-white television set the colors appear as slightly different gray-tones made up of distinct patterns of gray and white dots.

Since each byte in the text/low-resolution graphics display buffer represents two blocks on the screen stacked one above the other, each 8-bit byte is divided into two 4-bit parts called nibbles. Each nibble can be represented by a single hexadecimal digit. Since there are 2^4 or 16 bit combinations, each bit combination in a particular nybble represents a different color. The colors are as follows:

Figure 14.5A — Low Resolution Graphics Colors

Color on TV	Bit Pattern	Hexadecimal Representation	Decimal Representation
Black	0000	0	0
Magenta	0001	1	1
Dark Blue	0010	2	2
Purple	0011	3	3
Dark Green	0100	4	4
Gray 1	0101	5	5
Medium Blue	0110	6	6
Light Blue	0111	7	7
Brown	1000	8	8
Orange	1001	9	9
Gray 2	1010	A	10
Pink	1011	B	11
Light Green	1100	C	12
Yellow	1101	D	13
Aquamarine	1110	E	14
White	1111	F	15

Figure 14.5B - Low Resolution Graphics Colors

Decimal Code	Hex Code	Bit Code	Color	Distinguishable from (in B&W) Color identified by hex code
0	0	0000	Black	1 2 3 4 5 6 7 8 9 A B C D E F
1	1	0001	Magenta	0 3 5 6 7 9 A B C D E F
2	2	0010	Dark Blue	0 3 5 6 7 9 A B C D E F
3	3	0011	Purple	0 1 2 4 7 8 B D E F
4	4	0100	Dark Green	0 3 5 6 7 9 A B C D E F
5	5	0100	Gray 1	0 1 2 4 7 8 B E F
6	6	0110	Medium Blue	0 1 2 4 7 8 B D E F
7	7	0111	Light Blue	0 1 2 3 4 5 6 8 9 A C F
8	8	1000	Brown	0 3 5 6 7 9 A B C D E F
9	9	1001	Orange	0 1 2 4 7 8 B D E F
10	A	1010	Gray 2	0 1 2 4 7 8 B D E F
11	B	1011	Pink	0 1 2 3 4 5 6 8 9 A C F
12	C	1100	Light Green	0 1 2 4 5 8 B D E F
13	D	1101	Yellow	0 1 2 3 4 5 6 8 9 A C F
14	E	1110	Aquamarine	0 1 2 3 4 5 6 8 9 A C F
15	F	1111	White	0 1 2 3 4 5 6 7 8 9 A B C D E

The actual color displayed by your television set may vary from these standard values because you set the color and hue controls. They can also be adjusted by the COLOR TRIM control at the right rear of the Apple II main circuit board.

The value in the low-order (rightmost) nibble of the byte determines the color of the upper block of the display-pair; the one in the high-order (leftmost) nibble determines the color of the lower block. Thus a byte containing the binary bit pattern 11001000 (hexadecimal C8 — usually written $C8) would cause display of a brown block over a light green block.

When colors are displayed on a black-and-white TV set or monitor, they appear as black-and-white bit patterns rather like the conventional zip-tone black-and-white methods of coding and representing colors in printed books. Each pattern of shading represents a different color. Figure 14.5B shows which zip-tones are visually distinguishable from one another on a black-and-white display.

Follow these steps to put this information into decimal form for POKEing into memory: 1. get the decimal values of the colors from Figure 14.5B; 2.

add 16 times the decimal value for the block to be displayed on the bottom to the decimal value of the block to be displayed on the top to get the value to POKE into memory.

To obtain the colors from a decimal number obtained by PEEKing at the memory location, perform an integer division of the decimal number by 16. The color of the upper block is determined by the quotient; the color of the lower block by the remainder. For example, if the result of PEEKing is the number 208 (hexadecimal D8) then the quotient of 208/16 is 13, the remainder is 8, and the colors are brown and yellow.

Since the same block of memory is used for the text screen and for Low-Resolution graphics, interesting things happen if you put text into that block of memory and display it as Low-Resolution graphics or vice versa.

Each text character will become two blocks of hues determined by the ASCII code for the character. Because of the consistency of the high-order byte used in most text the display will often tend to show long horizontal gray, pink, green, or yellow bars separated by randomly colored blocks.

Conversely each block-pair will become an Apple text character from Table 14.2A. With a reasonably normal selection of colors many of these characters will be inverse or flashing characters and the screen will be a dazzling, flashing mess.

You can play interesting tricks on the computer by entering data in one mode and using it in another. Often in text mode certain characters are considered illegal, i.e. a comma in an input text string or a particular token representing an illegal command in BASIC. If you can grab such a string and treat it as Low-Resolution graphics data, you can bypass the checking which results in rejection of the data as illegal. With these tricks you can make errors that Apple tried to protect you against, but which smart programmers can sometimes use to their advantage.

Chapter XV
'User Memory' for BASIC Programmers
Typically Pages 9-149 ($0800-$95FF) But Highly Variable

15.1
Overview of 'User Memory' Space Available to BASIC Programmers

15.1.1 The Default Case

In the default case (where a user has a 48K Apple and is using ROM Applesoft) the RAM memory available is about $8DFC (decimal 36,348) bytes of memory.

Specifically, it consists of all of RAM between LOMEM (which is automatically set for new programs just beyond the end of Text Page 1 at $803) and HIMEM (which is automatically set to the beginning of DOS, normally $95FF).

15.1.2 Rationale for the Default Case

The memory available to BASIC programmers begins at memory page 8 because all lower pages are assigned to system firmware support uses.

As we have seen the bottom 1K of Apple memory, memory pages 0-3 ($0000-$03FF) is allocated to system functions such as systems worksheet, stack, input (keyboard buffer), and monitor special locations. One page is allocated to each of these functions, leaving some unused space on page 3 of this block available to the user. The second K of memory, pages 4-7 ($0400-$07FF), is allocated to system use as the primary text/low-res display output buffer (the scrolling output buffer associated with the keyboard input buffer). A few scattered memory locations in this area are also available for use by the peripheral expansion slots. Thus there is no usable space available below $0800.

The third K of memory, pages 8-11 ($0800-$11FF), is sometimes used as a secondary text/low-res display output buffer. Thus you might think that the automatically set LOMEM should be beyond this graphics page also, at about memory location $1200 or $1201.

However, the secondary text/low-resolution capability is used by only a small proportion of Apple programs. Thus both Apple BASIC interpreters, while allowing the user to set LOMEM at $1200, automatically set LOMEM at $803. This procedure makes an additional K of freely usable 'User Memory' available for BASIC programming.

The idea that LOMEM is automatically set to $803 is not a hard-and-fast rule. If you are using the older version of Applesoft (rather than the now more commonly used firmware, ROM, Applesoft Card, language-card RAM, or FP BASIC versions of Applesoft), the Applesoft interpreter itself will occupy memory pages 8-47 (memory locations $0800-$3000) and user memory will not begin until page 48 (memory location $3001).

At the HIMEM end, if you have an Apple that is not using DOS, user memory extends all the way to the top of RAM memory. This makes available an additional $2A00 (10752) bytes of memory. (The actual amount used will change somewhat if the default MAXFILES = 3 condition of the DOS is altered.)

If you have a 32K instead of a 48K Apple, you lose 16K = decimal 64 pages = $4000 = decimal 16384 bytes of memory. If you have a 16K Apple, you lose yet another 16K = $4000 = 16384 bytes of space.

In summary, the most common default condition, User Memory for Applesoft or Integer BASIC, runs from pages 8 through 149 (memory locations $0801-$95FF).

15.2
Variations in User Memory Availability In Different Hardware/Software Environments

Variations will occur from the default case as a result of common variations in the hardware/software environment.

If MAXFILES is used to change the amount of buffer memory space reserved by DOS, approximately $200 bytes of memory will be lost for each additional DOS buffer required above the default (MAXFILES = 3). However, if MAXFILES is reduced below the default value of 3, an equal amount of extra memory will become available for each buffer released. Thus if you can get by with MAXFILES = 1, you can get approximately $400 (about decimal 1000) extra bytes of memory for your Applesoft programs and data, but you will be severely limited in your flexibility in performing disk operating system activities.

If you choose to totally disable the disk operating system, or if you have removed it from its normal location into language card RAM, then you get a huge bonus of available memory. HIMEM, the top boundary of user-available memory, can be moved upwards to $BFFF. This

adds $2A00 (decimal 10752) bytes to the memory available to you.

However, if you use the version of Applesoft that occupies user RAM memory space, you receive a comparable penalty. RAM Applesoft occupies locations $800-$2FFF (decimal 2048 to $12287). Thus its use decreases the memory available for users by $2800 (decimal 10240) bytes.

If you have an Apple that has less than the full normal complement of 48K of RAM (exclusive of language card or equivalents), then HIMEM will move downward by the amount of memory missing. For example, if you have a 32K Apple, the amount of available memory would be reduced by 16K (decimal 16384) bytes.

15.3 Memory Allocation: Theory

15.3.1 Some Terminology, Fundamentals, and A Pictorial Overview

When Applesoft is set up to begin entry of a new program, the lowest memory address available for user program and data is called LOMEM; the highest available is called HIMEM. The as yet unused space between is called "user free space." This is the space into which user programs and program data (constants, variables, arrays, character strings, etc.) are automatically put by the Applesoft interpreter during the set-up and running of Applesoft programs.

When you create a BASIC program using the Applesoft interpreter, the interpreter automatically allocates space out of the free space area to meet four major needs:

1. Space for your BASIC program:
   The Applesoft interpreter puts a tokenized (specially abbreviated) copy of your source (BASIC) program immediately above LOMEM.
2. Space for Simple Variables:
   The Applesoft interpreter assigns space above the program to simple variables; i.e., variables that are not part of an array. There are three types of these: real number variables, integer number variables, and string pointers. String pointers are associated with string variables, but they do not contain the alphanumeric text of the string variable. They merely point to the location of the start of the string of characters and specify its length.
3. Space for Arrays:
   The Applesoft interpreter assigns space above that assigned to simple variables to arrays. As with simple variables, there are three types of arrays: real number arrays, integer number arrays, and string pointer arrays. As was the case for string variables, the actual alphanumeric characters of string arrays do not appear in the string pointer arrays, only pointers that specify where the alphanumeric characters are located.
4. Space for Character Strings:
   The actual alphanumeric characters associated with string variables and string arrays, as well as quoted charcter strings, are put into memory in the order of receipt working *downward* from HIMEM.

Notice that with this scheme of allocation, as a program increases in size, it eats away at the originally available free space from both the original LOMEM upward and HIMEM downward, leaving an ever-decreasing residue of the original free space somewhere in the middle.

A pictorial overview of this situation is provided in the *Applesoft II BASIC Programming Reference Manual*, provided with your Apple Computer. The diagram of Applesoft program memory map located on page 127 shows the allocation pattern. On page 137 the same source also provides a diagram of how individual variables and arrays are stored.

15.3.2 HIMEM and the Top End of User-Available Free Space

If the disk operating system is not in use, HIMEM is set to the highest location in RAM memory space. For a 48K Apple, this is address $BFFF (unsigned decimal 49151 or signed decimal −16383). For a 32K Apple, this is address $7FFF (decimal 32767). (Warning: If you have a language card or other RAM that overlays ROM and special I/O memory space $C000 through $FFFF, that space is *not* directly available to your Applesoft programs.)

If DOS is in use, it is located at the top of RAM memory and HIMEM is automatically reduced to the first unused location below DOS.

DOS occupies the space downward from the top of RAM memory ($BFFF for a 48K Apple) to the bottom of its last buffer. In the default case, three buffers are provided (MAXFILES = 3) and DOS occupies $2900 (decimal 10496) bytes of memory. Thus it extends downward to $9600 (unsigned decimal address 38400, signed decimal address −27136). HIMEM is automatically set to

this value by system software without human intervention.

With a 32K Apple in the same situation, the DOS would extend down to $5600 (decimal address 22016) and HIMEM is automatically set at that point.

Many commonly used Applesoft utilities also hide at the top of memory and push HIMEM down further. These include the Applesoft RENUMBER program on the DOS 3.3 System Master Diskette, the Applesoft Programmers' Assistant (APA) in the Applesoft Tool Kit, and the *CALL —A.P.P.L.E.* Program Line Editor (PLE). In most cases these utilities automatically set HIMEM at their bottom limit, usually without specifically notifying the user of its new value.

You may reduce the value of HIMEM even further by use of a HIMEM: command. (An alternate way of doing this is to POKE the desired new HIMEM value to memory locations $73,$74 (decimal 115,116)). You might, for instance, want to hide your own special machine-language utility program or a high-resolution graphics shape table above the area accessible to Applesoft. Obviously if you reduce HIMEM too much you risk the dreaded OUT OF MEMORY error condition.

At any time you wish, either during the preparation or the running of a BASIC program, you can inquire about the current value of HIMEM by examining memory locations $73, $74. A convenient means of doing this is the PRINT statement:

```
PRINT PEEK(115) + 256*PEEK(116)
```

There is no special error checking of the numeric values associated with the HIMEM: command. The computer will not give an error indication if a value of HIMEM is specified that is outside the range of available RAM memory — the only memory the contents of which can be successfully changed. (It will, however, give an "ILLEGAL QUANTITY ERROR" if the value specified is outside the range −65535 to +65535). Thus you can get yourself into big trouble if you are not careful!

The trouble is of a particularly insidious kind. Often a simple program will seem to work, even if you specify a HIMEM well up into ROM memory space and then start using character strings which obviously cannot be successfully put into the ROM memory immediately below HIMEM. The Applesoft interpreter is able to bypass some of the obvious traps you set for it when you tell it to allocate memory in an impossible address area, but it is not able to get around them all. Thus many programs do not give a hard failure, but instead will not execute reliably unless there is directly accessible RAM memory at all locations specified in the Applesoft program up to and including HIMEM.

Incidentally, an invalidly set value of HIMEM is a ticking time bomb, because it remains set, ready to do in your program(s) long after you may think it is gone. HIMEM: is not automatically reset by CLEAR, RUN, NEW, DEL, changing, or adding a program line. It is not even reset (under most circumstances) when you press 'RESET'! It is reset when you change language (INT or FP commands) or when you do a RESET CTRL-B RETURN (non-autostart ROM).

15.3.3 LOMEM and the Bottom End of User-Available Memory Space

LOMEM is the address of the lowest memory location available to a BASIC program. Unless you thoroughly understand how Applesoft handles the allocation of memory to programs and variables, it is very easy to misinterpret this statement.

When Applesoft is ready for entry of a new program, LOMEM, the bottommost memory location available for allocation to the user program and program data is automatically set to $803 if you are using ROM Applesoft.

You can, of course, decide to hide additional memory from Applesoft's memory allocation algorithms by using a LOMEM: command to set LOMEM deliberately to an even higher value.

Once you have these options firmly in mind you may think you have a handle on where LOMEM is. You do, but only for a while. If you put in your program, type run, and then query the system about the location of LOMEM, it won't be where you expected it! Instead it will be at a higher location, on occasions with large programs as much as 1000 locations higher, or even more!

How come? Applesoft moves it. Why? Look at figure 9.3A. The Applesoft interpreter moves your Applesoft Program in under LOMEM and pushes LOMEM upward by the amount of memory taken up by the program.

Each time you put a new statement in an Applesoft program it pushes up the location of all program statements after it. Strange as it may seem, every time you do this you push up the location of LOMEM, and with it the space for variables and arrays!

LOMEM may be examined at any time by studying the contents of locations $69,$6A

(decimal 105,106). This is conveniently done by the following PRINT statement:

```
PRINT PEEK(105) + 256* PEEK(106)
```

LOMEM may be set or reset by means of a LOMEM: statement. (An alternate method is by POKEing a new value to locations $69,$6A (decimal 105,106.) There is limited error checking at the time that LOMEM: is entered so the computer will accept values in the range −65535 to +65535. However, if LOMEM is set lower than the highest memory location occupied by the current operating system (plus any current stored program), or if it is set to a value higher than HIMEM, the system will produce an OUT OF MEMORY error as soon you attempt to run the program.

LOMEM is altered by any change in program length. It is reset to default values for the Applesoft interpreter you are using (ROM or RAM) by anything which deletes the current program. Thus it is reset by a NEW and by RESET CTRL-B.

LOMEM is not changed by commands that RESET (or its equivalent if you don't have an autostart ROM. Either RESET CTRL-C RETURN or RESET 3D0G RETURN).

Once set, unless it is first reset by one of the above commands, LOMEM: can be set to a new value *only* if the new value is *higher* in memory than the old. An attempt to set LOMEM: lower than the value still in effect would clobber the end of the program that had pushed LOMEM upward. Applesoft refuses to allow that to happen with the LOMEM: command. A lower value can be POKEd in.

It is perfectly possible and legitimate to change LOMEM during the execution of a program. However, it must be done with great care and sensitivity to current program functions and memory allocations, for the change may cause certain stacks or portions of a program to disappear or the linkages to them to become confused so that the program may no longer function properly.

15.3.4 Finding Out the Current Allocation Of User Memory in Your Applesoft BASIC Program

Before any BASIC program or data are entered (and/or the BASIC programmer initiates action to alter the normal allocation of memory) we have a particularly simple situation. The pointers that tell us the starting or ending addresses of all the major areas of our program all point to either LOMEM or HIMEM. Thereafter they begin to separate by amounts depending upon the nature and the size of the program which has been entered.

You can examine the allocation boundaries easily with the PRINT statements indicated below. These PRINT statements may be entered as immediate-execution statements when the program is not running or they can be given line numbers and imbedded in the running program. (In the latter case the space they themselves take will alter the results slightly.)

Figure 15.3A
Memory Allocations in Your Current Applesoft BASIC Program

```
The start-of-program address ($67,$68):
    PRINT PEEK(103)+256*PEEK(104)

The end-of-program address ($AF,$B0):
    PRINT PEEK(175)+256*PEEK(176)

The LOMEM/start-of-simple variables address ($69,$6A):
    PRINT PEEK(105)+256*PEEK(106)

The end-of-simple variables/start of arrays address($6B,$6C)
    PRINT PEEK(107)+256*PEEK(108)

The end-of-arrays/bottom-of-free-space address ($6D,$6E)
    PRINT PEEK(109)+256*PEEK(110)

The top-of-free space/next location for strings address
 ($6F,70)
    PRINT PEEK(111)+256*PEEK(112)

HIMEM ($73,$74):
    PRINT PEEK(115)+256*PEEK(116)
```

If you are familiar with machine language, you will probably prefer to get the hexadecimal form of these addresses. This is easily done by entering the Apple Monitor (CALL −151).

Type in the higher of the hexadecimal address pair (but don't include the $); press the spacebar; type in the lower address of the pair; press 'RETURN'. The monitor will print the two locations and the data in each; e.g.,

```
] CALL -151 <ret>
* 6E 6D <ret>

006E - 0A
006D - 01
```

The required memory allocation boundary address is four hex digits obtained by taking the two-digit contents of the *higher* pointer address followed by the two-digit contents of the *lower* pointer address; e.g., $0A01. Thus the location of the bottom of free space is $0A01 in the example.

15.4
How Memory Is Allocated for Program Code (BASIC Statement Structure)

15.4.1 Method of Allocation

The area between the start-of-program address (specified in $67,$68) and the end-of-program address (specified in $AF,B0) is occupied by BASIC statements. Each Applesoft BASIC line consists of the following modules in the order indicated:

Figure 15.4A
Layout of BASIC Statements in Program Memory Area

1. Next line address pointer (2 bytes)
2. BASIC line number (2 bytes)
3. BASIC token (1 byte)
4. Statement data or parameters
   - zero or more ASCII characters (0 or more bytes)
5. Delimiter(s)
   5L. End-of-line delimiter
   - ASCII `nul' or Hex `00' (1 byte)

   A compound statement consisting of a single line number, but multiple statements separated by colons will also have one or more of the following BEFORE the end-of-line delimiter

   5S. End-of-statement delimiter
   - ASCII `:' or Hex `3A' (one byte)
   After each end-of-statement delimiter the compound statement will restart WITHOUT a next line address or BASIC line number, i.e. it will restart with the token for the statement following the delimiter

   5P. End-of-program delimiter
   - ASCII `nul' nul' or Hex `00 00' (two bytes)

   The end-of-program delimiter follows an end-of-line delimiter. Every program terminates with three `nuls' i.e. `00 00 00'.

Applesoft programs are tokenized. Associated with each of these tokens is a subroutine that implements the activity described by that token.

The BASIC statement (and hence its token and subroutine) may neither need nor accept any additional information in the form of parameters or it may accept a considerable number of them. Sometimes the same BASIC statement-type has both options; e.g., 'PRINT' or 'PRINT A, B, C, D, ..., Z'. Regardless of the type of BASIC statement or its parameter-list options, the end of the parameter list is marked by an end-of-statement or end-of-statement/end-of-line marker.

Parameters are expressed as a series of ASCII characters. These represent whatever type of parameters are relevant whether they be variables, operators, functions, numeric literals, string literals, or some complex combination of all of these.

Whatever these parameters are, they must conform to some predefined rules of BASIC grammar that make it possible for the subroutine to determine the proper meaning. Failure to conform to these rules leads to the dreaded 'SYNTAX ERROR' when the subroutine is not able to decipher what action BASIC was supposed to take.

With this minimal background let's now examine some sample programs and see what we can learn from specific examples about how memory is allocated for program code in BASIC programs.

15.4.2 Sample Program and Analysis of How it Appears in 'User Memory'

You can examine how memory is allocated to it by the procedure of Figure 15.4B:

Figure 15.4B
Method for Analysis of Tokenized BASIC Statements in `User Memory'

(1) Enter program and list it
    For example, you might enter and list the following program:
```
10 LET A=5
20 LET B=3
30 LET C=A+B
40 PRINT "C=";C
50 END
```
(2) Enter system monitor (CALL -151)
(3) Get start-of-memory address from $67,68
(4) Get end-of-program address from $AF,$B0
(5) Dump memory from start-of-program address to end-of-program address
(6) Analyze information from dump using allocation pattern
    (See STATEMENT dump below)

The entries on the screen to steps (2) to (5) with the sample program appear on the screen as follows:

```
]CALL -151
*68 67

0068- 08
0067- 01   <Start-of-prog. address = 0801>

*B0 AF

00B0- 08
00AF- 32   <End-of-prog. address + 1 = $0832

           Dump memory from $0801 to $0831>

0801- 0A 08 0A 00 AA 41 D0
0808- 35 00 13 08 14 00 AA 42
0810- D0 33 00 1E 08 1E 00 AA
0818- 43 D0 41 C8 42 00 2A 08
0820- 28 00 BA 22 43 3D 22 3B
0828- 43 00 30 08 32 00 80 00
0830- 00 00
```

This dump may be analyzed as follows: (Note that ASCII characters can be represented in form shown in dump or with value $80 larger.)

STATEMENT: 10 LET A = 5

0801,0802:	Pointer to next line of BASIC program '0A 08' Next line starts at $080A.
0803,0804:	BASIC line number of statement '00 0A' Line number 10 ($0A = 10)
0805:	BASIC Token 'AA' Token for 'LET'
0806-0808:	Parameters: '41' ASCII character 'A' 'D0' Operator tag '=' '35' ASCII character '5'
0809:	End-of-line delimiter '00'

STATEMENT: 20 LET B = 3
- 080A,080B: Pointer to next line of BASIC program
  '13 08' Next line starts at $0813
- 080C,080D: BASIC line number of statement
  '14 00' Line number 20 ($14 = 20)
- 080E: BASIC Token
  'AA' Token for 'LET'
- 080F-0811: Parameters:
  '42' ASCII character 'B'
  'D0' Operator tag '='
  '33' ASCII character '3'
- 0812: End-of-line delimiter '00'

STATEMENT: 30 LET C = A + B
- 0813,0814: Pointer to next line of BASIC program
  '1E 08' Next line starts at $081E
- 0815,0816: BASIC line number of statement
  '00 AA' Line number 30
  ($AA = 30)
- 0817: BASIC Token
  'AA' Token for 'LET'
- 0818-081C: Parameters:
  '43' ASCII character 'C'
  'D0' Operator tag '='
  '41' ASCII character 'A'
  'C8' Operator tag '+'
  '42' ASCII character 'B'
- 081D: End-of-line delimiter '00'

STATEMENT: 40 PRINT''C='';C
- 081E,081F: Pointer to next line of BASIC program
  '2A 08' Next statement starts at $082A
- 0820,0821: BASIC line number of statement
  '28 00' Line number 40 ($28 = 40)
- 0822: BASIC Token
  'BA' Token for 'PRINT'
- 0823-0828: Parameters:
  '22' ASCII Character '"'
  '43' ASCII Character 'C'
  '3D' ASCII Character '='
  '22' ASCII Character '"'
  '3B' ASCII Character ';'
  '43' ASCII Character 'C'
- 0829: End-of-line delimiter '00'

STATEMENT: 50 END
- 082A,082B: Pointer to next line of BASIC program
  '30 08' Next statement starts at $0830
- 082C,082D: BASIC Line number
  '32 00' Line number = 50
  ($32 = 50)
- 082E: BASIC Token
  '80' Token for 'END'
- 082F: End of Line Indicator '00'

PROGRAM TERMINATION:
- 0830-0831: Pointer to next line of BASIC program
  '00 00' End-of-program indicator

15.4.3 Modified Sample Program and its Analysis

To see what difference it would make had the same program been written as a single compound statement:

```
10 LET A=5:LET B=3:LET C=A+B
   :PRINT "C=";C:END
```

we could undertake the same method of analysis with the modified program:

```
]CALL -151
*68 67

0068- 08
0067- 01 <Start-of-program address =
         $0801>

*B0 AF

00B0- 08
00AF- 22 <End-of-program address + 1
         = $0822>

*801.821

0801- 20 08 0A 00 AA 41 D0
0808- 35 3A AA 42 D0 33 3A AA
0810- 43 D0 41 C8 42 3A BA 22
0818- 43 3D 22 3B 43 3A 80 00
0820- 00 00
```

The detailed analysis of this version of the program follows:

STATEMENT: 10 LET A = 5: LET B = 3: LET C = A + B: PRINT "C = ";C:END
- 0801,0802: Pointer to next line of BASIC program
  '20 08' Next line starts at $0820
- 0803,0804: BASIC line number of statement
  '00 0A' Line number 10
  ($0A = 10)
- 0805: BASIC Token
  'AA' Token for 'LET'
- 0806-0808: Parameters:
  '41' ASCII character 'A'
  'D0' Operator tag '='
  '35' ASCII character '5'
- 0809: End-of-statement delimiter ':'
  '3A' ASCII character ':'

- - - - -

- 080A: BASIC Token
  'AA' Token for 'LET'

```
080B-080D:  Parameters:
            '42' ASCII character 'B'
            'D0' Operator tag '='
            '33' ASCII character '3'
080E:       End-of-statement delimiter ':'
            '3A' ASCII character ':'
- - - - -
080F:       BASIC Token
            'AA' Token for 'LET'
0810-0814:  Parameters:
            '43' ASCII character 'C'
            'D0' Operator tag '='
            '41' ASCII character 'A'
            'C8' Operator tag '+'
            '42' ASCII character 'B'
0815:       End-of-statement delimiter ':'
- - - - -
0816:       BASIC Token
            'BA' Token for 'PRINT'
0817-081C:  Parameters:
            '22' ASCII Character '"'
            '43' ASCII Character 'C'
            '3D' ASCII Character '='
            '22' ASCII Character '"'
            '3B' ASCII Character ';'
            '43' ASCII Character 'C'
081D:       End-of-statement delimiter ':'
            '3A' ASCII Character ':'
- - - - -
081E:       BASIC Token
            '80' Token for 'END'
081F:       End-of line delimiter  '<nul>'
            '00' ASCII Character   '<nul>'
PROGRAM TERMINATION:
0820-0821:  Pointer to next line of BASIC
            program
            '00 00' End-of-program indicator
```

15.4.4 Yet Another Sample Program for Analysis

Consider the following sample program:

```
00 REM SAMPLE PROGRAM 3
10 PRINT "HELLO, WHAT'S YOUR NAME";:
   INPUT NAME$
20 PRINT "GLAD TO MEET YOU,";NAME$
30 READ X1,X2
40 DATA 5,3
50 PRINT X1 + X2
60 END
```

As before we could get the beginning-of-program address ($0801) and the end-of-program address ($0887), do a hexadecimal dump using the monitor, and do an analysis from it.

Notice what we can see without in-depth analysis:

1. The comment portion of any REM statement is imbedded in the body of the program as ASCII characters, one byte per character of REM comment.
2. Variable names, such as NAME$, X1 and X2 are imbedded in their entirety in the body of the program as ASCII characters, also at the rate of one byte per character.
3. The string literals are also imbedded in the body of the program as ASCII characters, also at the rate of one byte per character of literal.
4. The numeric literals are imbedded in the body of the program as ASCII characters, also at the rate of one byte per character.

It seems unnecessary for this text to present the analysis of this third sample program in the same level of detail as that used before. Readers, however, are encouraged to undertake its analysis if you have not previously analyzed any program on your own.

15.4.5 Lessons to be Learned from Analysis of the Three Sample Programs

You should now be able to take any BASIC program and, if you are willing to undertake all of the detailed step-by-step analysis described in the previous sections, figure out exactly where and how every BASIC statement is represented in memory. This project is not something you will want to do often, but undertaking it these few times should have made several points obvious. For example:

1. You can save memory by leaving out REM statements.
   This is unfortunate because REM statements are valuable tools to help make programs more readable and more understandable. Fortunately there are various compacting utilities available in sources, such as Apple's 'Applesoft Tool Kit,' that enable you to have a fully documented master or developmental version of a program, then automatically delete REMs to save space and time in the version you use for everyday operations.
2. You can save memory by using shorter variable names.
   This is also unfortunate because self-explanatory variable names can be very helpful in reading and understanding a program. If you keep separate development and running versions of a program, it may be convenient to

use an editor, such as the *CALL A.P.P.L.E.* Program Global Editor, to replace the full names in the master copy of the program by only their first two letters in a run copy.

3. Strings are not necessarily located at the top of user memory.
   Although we often tend to think of string data as being allocated from the top of user memory downward, a major proportion of the character strings in most programs are not treated as string variables allocated in that fashion, but as string literals imbedded in the body of the program. This may be very significant if you are developing large high-resolution graphics programs and choose not to take the special protective measures for avoiding memory allocation conflicts recommended in the chapter on high-resolution graphics.

4. You can waste a great deal of machine time by unnecessary use of numeric literals.
   Numeric literals, e.g. 5 or 3.14159, become part of the body of a BASIC program. Every time the instruction containing the literal is executed, the computer must go through the process of number conversion to binary form. If the same number had been represented by a variable name that had once been set to that value, only one conversion would have been needed no matter how many times the statement is executed. You should be able to add several additional items to this list.

15.4.6 Using Memory Allocation Information to Create Self-Modifying BASIC Programs

Self-modification is an extremely powerful programming tool. It is also dangerous, potentially addictive, and the source of much self-indulgence in programming.

Self-modifying programs are the antithesis of structured programming. Whereas structured programming deliberately restricts the number and nature of programming structures that a programmer uses to make programs easier to follow and more modular in structure, self-modification allows limitless freedom and even permits you to change control structures on-the-fly while a program is being executed.

Self-modifying programs are often almost totally incomprehensible except to someone who knows the system hardware and firmware intimately. Often it is important to know exactly how the BASIC interpreter works and to be willing to sit down and analyze the self-modifying portions of the program step-by-step with great care.

Self-modifying programs are frequently interesting intellectual puzzles, but writing programs that are intellectual puzzles is seldom a sign of good programming. Like tobasco sauce or jalapeno peppers, self-modification must be used with care and moderation. It is not recommended as an every-day programming style.

So much for the disclaimers needed to put this technique into proper context. On the other side of the coin, there are occasions where self-modification leads to programs that are easier to use and/or understand.

Suppose, for example, that your analysis of a problem indicates that it involves 500 cases and the most straightforward way to solve it involves a 500-way branch. A 500-way branch is difficult to do simply in Applesoft BASIC.

The multi-way branch 'ON ... GOTO ...' will not accept that many options, and even if it did, it would create a difficult-to-understand mess in your program. It would like to have an ability to write GO TO A (or GOTO A%), where A or A% is a variable representing a line number that can be computed. However, Applesoft will not accept GOTOs in this form.

There are times when it would be useful to have a utility that modifies Applesoft to give it such a capability. Self-modification provides the power to trick Applesoft into providing this capability. Such a utility written in self-modifying Quasi-BASIC, with no hidden machine-language code, provides such a capability.

Figure 15.4C
Self-Modifying Applesoft BASIC Utility to Provide Capability to Perform GOTO <variable line number>

```
 1 GOTO 5
 2 FOR I=0 TO 4:POKE 2134+I,48:NEXT I: A$=STR$(A):
   FOR I=LEN(A$) TO 1 STEP -1:
   POKE 2138-LEN(A$)+I,ASC(MID$(A$,I,1)):NEXT I:GOTO 99999
 3 REM  WHENEVER GOTO 2 IS CALLED, 'GOTO 99999' IN LINE 2 WILL BE MODIFIED
        TO BECOME GOTO <VALUE OF VARIABLE A>; CONTROL WILL BE TRANSFERED TO
        LINE A.  FIRST LOOP IS NEEDED ONLY FOR SECOND AND SUBSEQUENT USES.
        ITS PURPOSE IS TO RESTORE THE MODIFIED GOTO TO ORIGINAL 'GOTO 9999

 5 REM BEGIN MAIN PROGRAM HERE
 .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
 .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
 .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
 100 A=1000:GOTO2: REM COMPUTE VALUE OF A BY ANY DESIRED MEANS.
                   GOTO 2 HAS THE EFFECT OF 'GOTO A'. A=1000, SO GOTO 1000
 .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
 1000 PRINT "THIS IS LINE 1000.  ARRIVE HERE BY GOTO 2 WHEN A=1000"
 .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
 1200 A=2000:GOTO2: REM COMPUTE A NEW VALUE OF A=2000. GOTO 2 WILL GOTO 2000
 .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
 2000 PRINT "THIS IS LINE 2000.  ARRIVE HERE BY GOTO2 WHEN A=2000"
 .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
```

The self modification in this program occurs in line 2. In that line a dummy statement GOTO

00000 is created and modified to take on the value of variable A. The part of the program to be self-modified is put at the beginning of the program so that changes in the program will not affect its location in memory. Knowledge of the location of the '00000' in the 'GOTO 00000' is essential in order to modify it to 'GOTO 01000' when A = 1000, 'GOTO 02000' when A = 2000, etc.

15.5 How 'User Memory' is Allocated for Simple Variables (Variables other than Arrays)

The pointer located in 105,106 (\$69,\$6A) indicates the start of that portion of user memory allocated to simple variables. The pointer located in 107,108 (6B,\$6C) indicates the end of the portion of user memory allocated to simple variables. This section deals with that area of memory. More specifically it deals with how that section of memory is allocated by the Applesoft BASIC interpreter.

We will pay particular attention to how you can determine where each variable in your BASIC program is located in the computer's memory. We will also provide a utility program you can use to get such information easily.

15.5.1 Information Layout for Individual Variables

Each simple variable, regardless of whether it is associated with a real number (figure 15.5A), an integer (figure 15.5B), or a character string (figure 15.5C), takes exactly seven bytes of space: two bytes for the variable and seven bytes for the data.

If you assign single-character names, those names are padded with a null character to become a two-character name in the table entries. If you assign names longer than two characters only two characters are retained in the table. Thus Applesoft BASIC is unable to distinguish between two supposedly different variables that share the same first two letters; e.g., variable 'YEAR1' is indistinguishable from variable 'YEAR2'.

The computer can distinguish between the variables AX (type real), AX% (type integer) and AX\$ (type string). It does not accomplish this by storing the type indicator explicitly, but by control of the high-order bit of the alphabetic name characters in the variable tables. The rule is simple:

> If a variable is of type real, both ASCII characters of the variable name in the table of variables will be positive ASCII. That is, both will have their high bits off (values less than \$80).
>
> If the variable is of type integer, both ASCII characters will be in negative ASCII (high bits on). That is, both ASCII characters will have values equal to or greater than \$80.
>
> If a variable is a string variable, the first ASCII character will be positive ASCII — it will have its high bit off; but the second will be negative ASCII — it will have its high bit on. The first character will have value less than \$80, the second equal to or greater than \$80.

Figure 15.5A
MEMORY LAYOUT: INDIVIDUAL
TYPE REAL SIMPLE VARIABLES

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Vbl Name Char 1	Vbl Name Char 2	Real No Exponent	Real No Mantissa1	Real No Mantissa2	Real No Mantissa3	Real No Mantissa4
<—Variable Name—>		<—Variable Value—>				
+ASCII	+ASCII					
Variable Name = A		Variable Value = 12.34				
65	0	132	69	112	163	215
+ASCII	+ASCII					

Figure 15.5B
MEMORY LAYOUT: INDIVIDUAL
TYPE INTEGER SIMPLE VARIABLES

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Vbl Name Char 1	Vbl Name Char 2	Integer Hi-Value	Integer Lo-Value	0	0	0
<—Variable Name—>		<—Variable Value—>				
-ASCII	-ASCII					
Variable Name = A%		Variable Value = 1234				
193	128	4	210	0	0	0
-ASCII	-ASCII					

Figure 15.5C
MEMORY LAYOUT: INDIVIDUAL
STRING POINTER SIMPLE VARIABLES

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Vbl Name Char Nol	Vbl Name Char No2	String Length	Start Lo-Address	Start Hi-Address	0	0
<—Variable Name—>		<—String Parameters—>			<—Two Zeros—>	
Variable Name = A\$		Variable Value = "THIS IS A STRING"				
65	128	16	30	9	0	0
+ASCII	-ASCII					

15.5.2 Analyzing Variable Allocation Information Using the System Monitor

Space is assigned to variables in the order that they are first mentioned in the program. Thus, if only one variable has been used, you will have only a single 7-byte entry; if two have been mentioned you will have two entries; if N have been mentioned you have a table of N entries.

The different types of simple variables are not

segregated. They appear in the order in which they were named, regardless of type.

You can investigate the allocation of variables without the aid of any software tools other than those in the monitor. The procedure is as follows:

```
  Figure 15.5D
Using Monitor to Find Applesoft Variables
```

1. Enter the monitor - CALL -151
2. Find the start of simple variables. (Use pointer in $69,$6A)
3. Find the end of simple variables . (Use pointer in $6B,$6C).
4. Dump memory between these limits
5. Use the layout patterns indicated in Figure 15.5A-C to make the analysis

For example:

```
10 LET A=5
20 LET B=3
30 LET C=A+B
40 PRINT "C=";C
50 LET D$="THAT'S ALL, FOLKS!":PRINT D$
60 LET E%=100
70 END

]RUN

C=8
THAT'S ALL FOLKS!

]CALL -151
*6A 69

006A-08
0069-5F  <Variables start at $085F>

*6C 6B

006C-08
006B-82  <Variables end at $0882 - 1>

085F.0881 <= Dump Simple Variables
085F- 41
0860- 00 83 20 00 00 00 42 00
0868- 82 40 00 00 00 43 00 84
0870- 00 00 00 00 44 80 11 33
0878- 08 00 00 C5 80 01 07 00
0880- 00 00 20
```

NOTE: As before a dump with alphanumeric formatting can be helpful, for it would highlight the positions of the variable names: A at $085F;B at $0867;C at $086D D at $0874 and E at $087B

The dump can be analyzed byte-by-byte as follows:

```
                Figure 15.5E
Byte-by-Byte Analysis of Monitor Printout of Variable Data
-------------------------------------------------------------
005F- 41 Positive ASCII `A' <= Real Variable A
0060- 00 Positive ASCII `nul'
0061- 83 Exponent= $83      <= Value of A = 5
0062- 20 Mantissa 1 = $20
0063- 00 Mantissa 2 = $00
0064- 00 Mantissa 3 = $00
0065- 00 Mantissa 4 = $00
------
0066- 42 Positive ASCII `B' <= Real Variable B
0067- 00 Positive ASCII `nul'
0068- 82 Exponent= $82      <= Value of B = 3

0069- 40 Mantissa 1 = $40
006A- 00 Mantissa 2 = $00
006B- 00 Mantissa 3 = $00
006C- 00 Mantissa 4 = $00
------
006D- 43 Positive ASCII `C' <= Real Variable C
006E- 00 Positive ASCII `nul'
006F- 84 Exponent= $84      <= Value of C = 8
0070- 00 Mantissa 1 = $00
0071- 00 Mantissa 2 = $00
0072- 00 Mantissa 3 = $00
0073- 00 Mantissa 4 = $00
------
0074- 44 Positive ASCII `D' <= Name of String Variable D
0075- 80 Negative ASCII `nul'
0076- 11 Length of String = $11 (decimal 17)
0077- 33 LSB of String Add  <= String starts at $0833
0078- 08 MSB of String Add
0079- 00 Zeros for String Vbl
007A- 00 Zeros for String Vbl
------
007B- C5 Negative ASCII `E' <= Integer Variable E
007C- 80 Negative ASCII `nul'
007D- 01 Hi-Byte of Integer <= Integer Value $107 = dec 263
007E- 07 Lo-Byte of Integer
007F- 00 Zeros for Integer Vbl
0080- 00 Zeros for Integer Vbl
0081- 00 Zeros for Integer Vbl
-------------------------------------------------------------
```

The information we wanted about where each variable was located in memory and where to find its value was definitely there, but it was difficult to work the information around into a form that was usable!

15.5.3 Locating Applesoft Variables Using a Utility Written in Applesoft

Once you know that variables are located in 7-byte long modules and that the type of variable is determined by the combination of +ASCII and -ASCII used in storing its name (first two characters only), then it is easy to write a utility to do the busy work of section 15.5.2 for you.

It is convenient to have this utility in two forms:

1. A fully-documented self-demonstrating version (figure 15.5F), and
2. A stripped-down version which takes minimum space in memory .

Figure 15.5F

```
2  REM   ======DEMONSTRATION ENVIRONMENT======
4  TEXT : HOME : PRINT  TAB( 13);"DEMONSTRATION": PRINT : PRINT  TAB( 11
);"FOR PRACTICAL USE": PRINT "   DELETE ALL LINE#'S L
    ESS THAN 60000,": PRINT " APPEND THIS SUBROUTINE TO YOUR PROGRAM":
PRINT "  AND CALL 60000 AT END OF YOUR PROGRAM"
5  PRINT : PRINT "PRESS ANY KEY TO CONTINUE...": GET ANSWER$
6 A$ = "A$":AA$ = "AA$":A = 1:AA = 2:A% = 3:AA% = 4:B$ = A$:B = A:C$ = B
$: GOSUB 60000
8  PRINT "END OF DEMONSTRATION": END
60000  REM   ======SUBROUTINE TO LOCATE SIMPLE VARIABLES======
60002  TEXT : HOME :HEX$ = "0123456789ABCDEF"
60004 STASVAR =  PEEK (105) +  PEEK (106) * 256: REM  LOCATE START-OF-SI
MPLE-VARIABLES
60006  REM
60008  PRINT "  SUBROUTINE TO FIND SIMPLE VARIABLES": PRINT : PRINT "  L
OCATIONS EXPRESSED AS OFFSETS FROM": PRINT "        VE
    CTOR $69,$6A (105,106)"
60010  PRINT : PRINT "CURRENT VECTOR VALUE = ";STASVAR;: GOSUB 60128
60012  PRINT : PRINT  TAB( 7);"ANY CHANGE IN YOUR PROGRAM": PRINT "  WIL
L CHANGE THE VALUE OF THIS VECTOR"
60014 FINSVAR =  PEEK (107) +  PEEK (108) * 256: REM  LOCATE FINISH-OF-S
IMPLE-VARIABLES
60016 OFFSET = 0: REM  SET TO ZERO FOR VARIABLE SEARCH
60018  PRINT : PRINT  TAB( 5);"TO GET DECIMAL VALUE OF VECTOR": PRINT  T
AB( 6);"PRINT PEEK(105)+256*PEEK(106)"
60020  PRINT : PRINT  TAB( 5);"TO GET HEX VALUE FROM MONITOR": PRINT  TA
B( 7);"CALL -151 <CR> 6A 69 <CR>": GOSUB 60136
60022  PRINT "TABLE BELOW SHOWS INTERNAL LAYOUT OF": PRINT "EACH TYPE OF
 INFORMATION": PRINT "WITHIN EACH TYPE OF SIMPLE VARI
    ABLE": PRINT
60024  PRINT "FORMATS ARE AS FOLLOWS:"
60026  PRINT "LOCATION   INTEGER    REAL      STRING": PRINT "          VA
RIABLE  VARIABLE  POINTER"
60028  PRINT "---------: -------- : --------: -------"
60030  PRINT "OFFSET+0: <----  1ST CHAR OF NAME ---->"
60032  PRINT "OFFSET+1: <--NULL OR 2ND CHAR OF NAME-->"
60034  PRINT "OFFSET+2: VALUE-HI: EXPONENT:LENGTH"
60036  PRINT "OFFSET+3: VALUE-LO: MANT1    :ADDRESS-LO"
60038  PRINT "OFFSET+4:   0      : MANT2    :ADDRESS-HI"
60040  PRINT "OFFSE+5:   0      : MAT3  :  0"
60042  PRINT "OFFSET+6:  0: MANT4   : 0"
60043 PRINT : PRINT  TAB( 14);"FOR EXAMPLE,": PRINT " ADDRESS OF INTEGER
 VARIABLE DATA VALUE": PRINT  TAB( 19);"IS": PRINT  TAB(
    6);"VECTOR (105,106) + OFFSET + 2": GOSUB 60136
60044  PRINT "THIS SUBROUTINE USES THE FOLLOWING VBLS"
60046  PRINT "   HEX$ = '0123456789ABCDF'"
60048  PRINT "   ZZ$    <DUMY RESPONSE>"
6000  PRINT "   STASVAR <START IMPLE VAIALES>"
60052  PRIN "   FINSVAR <FINIS SIPLE ARIABLES>"
60054  PRINT "   OFFSET  <OFFSET FROM STASVAR>"
60058  PRINT : PRINT "PLEASE VOIDCONFLICTS!": GOSUB 60132
60060  REM  *****EXMNE & PRINT OUT VARIABLE LOCATION INFORMATION****
60062  HOME
60064  REM
60066  EM   ETEMINE TPE OF NEX SIMLE-VARIABLE
60068  PRINT "CURRENT OFSET = 0 AT ";STASAR;: GSUB 608: PRINT
60070  PRINT "VBL  OFFSET     VARIABLE       TRAILING"
60072  PRINT "NAME DEC(HEX)   TYPE             ZEROS"
60074 PRINT "---- ---(---)   -------------    -----"
60076  IF  PEEK (STASVAR + OFFSET) < 128 AND  PEEK (STASVAR + OFFSET + 1
) > 128 THEN 60086: REM  DOUBLE-CHARACTER STRING VARI
    ABLE EXIT
60078  IF  PEEK (STASVAR + OFFSET) < 128 AND  PEEK (STASVAR + OFFSET + 1
) = 128 AND  PEEK (STASVAR + OFFSET + 4) <  > 0 THEN
     60086: REM  SINGLE CHARACTER STRING VARIABLE EXIT
60080  IF  PEEK (STASVAR + OFFSET) < 128 AND  PEEK (STASVAR + OFFSET + 1
) < 128 THEN 60104: REM  REAL VARIABLE EXIT
60082  GOTO 60094: REM  INTEGER VARIABLE EXIT
60084  REM
60086  PRINT "            STRING:LEN. LOCATION";"   00": REM   *****STR
ING POINTERS******
60088  GOSUB 60116: PRINT "$"; TAB( 5);: GOSUB 60118: PRINT "        ";
PEEK (STASVAR + OFFSET + 2);"    "; PEEK (STASVAR + O
    FFSET + 3) +  PEEK (STASVAR + OFFSET + 4) * 256: PRINT : GOTO 60144

60090  PRINT  PEEK (I);" ";: NEXT : PRINT : GOTO 60144
60092  REM
60094  PRINT "             INTEGER:VALUE";"        000": REM    ***** I
NTEGER VARIABLE *****
60096  GOSUB 60116: PRINT "%"; TAB( 5);: GOSUB 60118: PRINT "         ";

60098  PRINT  PEEK (STASVAR + OFFSET + 3) +  PEEK (STASVAR + OFFSET + 4)
 * 256: PRINT : GOTO 60144
60100  PRINT  PEEK (I);" ";: NEXT : PRINT : GOTO 60144
60102  REM
60104  PRINT "             REAL:EXP   M1  M2  M3  M4": REM   *****REAL V
ARIABLES *****
60106  GOSUB 60116: PRINT  TAB( 5);: GOSUB 60118: PRINT "      ";
60108  FOR I = STASVAR + OFFSET + 2 TO STASVAR + OFFSET + 6
60110  IF  PEEK (I) < 100 THEN  PRINT " ";: IF  PEEK (I) < 10 THEN  PRIN
T " ";
60112  PRINT  PEEK (I);" ";: NEXT : PRINT : PRINT : GOTO 60144
60114  REM  **S/R TO PRINT VARIABLE NAME***
60116  PRINT  CHR$ ( PEEK (STASVAR + OFFSET)); CHR$ ( PEEK (STASVAR + OF
FSET + 1));: RETURN
60118  IF OFFSET < 10 THEN  PRINT "0";
60120  IF OFFSET < 100 THEN  PRINT "0";
60122  PRINT OFFSET;"($";: GOSUB 60126: PRINT ")";: RETURN
60124  REM  *** S/R FOR DEC=>HEX CONVERSION***
60126  PRINT  MID$ (HEX$,1 + OFFSET / 16,1); MID$ (HEX$,1 + OFFSET - 16
*  INT (OFFSET / 16),1);: RETURN
60128  REM  *** S/R TO PRINT STASVAR IN HEX ***
60130  POKE 1007,OFFSET:OFFSET =  INT (STASVAR / 256): PRINT "($";: GOSU
B 60124:OFFSET = STASVAR - 256 * OFFSET: GOSUB 60124:
     PRINT ")":OFFSET =  PEEK (1007): RETURN
60132  REM  *** S/R TO WAIT FOR USER RESPONS[ ***
60134  PRINT : INVERSE : PRINT "HIT ANY KEY TO CONTINUE...";: NORMAL : G
ET ZZ$: PRINT : RETURN
60136  REM  *** S/R FOR GETTING OR BYPASSING MORE INFORMATION ***
60138  PRINT : INVERSE : PRINT "  IF YOU WANT MORE INFORMATION TYPE '?'"
: PRINT "  ANY OTHER KEY GETS VARIABLE LOCATIONS";: NORMAL
    : GET ZZ$: PRINT : IF ZZ$ <  > "?" THEN  POP : GOTO 60060
60140  HOME : RETURN
60142  REM= LOOP END - GO BACK FOR NEXT VARIABLE -
60144 OFFSET = OFFSET + 7: IF STASVAR + OFFSET > FINSVAR THEN  RETURN
60146  IF  PEEK (37) > 19 THEN  GOSUB 60134: HOME : GOTO 60068: REM  INT
ERRUPT AT BOTTOM OF PAGE; START FRESH PAGE
60148  GOTO 60076

]
```

The latter form of the program is less than half the size of the former. It can be obtained by stripping out self-documenting and self-demonstrating features, using only the first two characters of the variable names and removing all REMs.

The output produced by running version 1 of this utility is shown in figure 15.5G1. Version 2 eliminates the explanations and gives only the variable information (figure 15.5G2). Notice that locations are given in terms of OFFSET from the current value of the start-of-simple-variables.

Figure 15.5G1 DEMONSTRATION

```
FOR PRACTICAL USE
DELETE ALL LINE#'S LESS THAN 60000,
APPEND THIS SUBROUTINE TO YOUR PROGRAM
AND CALL 60000 AT END OF YOUR PROGRAM

PRESS ANY KEY TO CONTINUE...
SUBROUTINE TO FIND SIMPLE VARIABLES

LOCATIONS EXPRESSED AS OFFSETS FROM
VECTOR $69,$6A (105,106)

CURRENT VECTOR VALUE = 6316($18AC)

ANY CHANGE IN YOUR PROGRAM
WILL CHANGE THE VALUE OF THIS VECTOR

TO GET DECIMAL VALUE OF VECTOR
PRINT PEEK(105)+256*PEEK(106)

TO GET HEX VALUE FROM MONITOR
CALL -151 <CR> 6A 69 <CR>

IF YOU WANT MORE INFORMATION TYPE '?'
ANY OTHER KEY GETS VARIABLE LOCATIONS
```

Figure 15.5G2

```
CURRENT OFFSET = 0 AT 6316($18AC)

VBL  OFFSET     VARIABLE          TRAILING
NAME DEC(HEX)   TYPE                ZEROS
---- ---(---)   -------------      -----
                STRING:LEN. LOCATION    00
AN$    000($00)         1     38399

                STRING:LEN. LOCATION    00
A$     007($07)         2     2334

                STRING:LEN. LOCATION    00
AA$    014($0E)         3     2343

                REAL:EXP   M1  M2  M3  M4
A      021($15)       129   0   0   0   0

                REAL:EXP   M1  M2  M3  M4
AA     028($1C)       130   0   0   0   0

                INTEGER:VALUE           000
A%     035($23)         3

                INTEGER:VALUE           000
AA%    042($2A)         4

                STRING:LEN. LOCATION    00
B$     049($31)         2     2334

                REAL:EXP   M1  M2  M3  M4
B      056($38)       129   0   0   0   0

                STRING:LEN. LOCATION    00
C$     063($3F)         2     2334
```

```
                STRING:LEN.  LOCATION    00
HE$     070($46)         16      2492

                REAL:EXP    M1   M2   M3   M4
ST      077($4D)        141   69   96    0    0

                REAL:EXP    M1   M2   M3   M4
OF      084($54)        135   40    0    0    0

                REAL:EXP    M1   M2   M3   M4
FI      091($5B)        141   72  112    0    0

                STRING:LEN.  LOCATION    00
ZZ$     098($62)          1      38398

END OF DEMONSTRATION

]
```

The location of variables will change as you add to or delete lines from your program. (Unless specific instructions are given to the contrary, the simple variable table location moves around so that it is always immediately after the end of the program.) However, the OFFSET for a given variable from the vector specified in $69,$6A (105,106) will remain constant. (This is true, of course, only if you do not create new variables earlier in the modified program.)

The name and location of each variable are the most important items indicated.

If the variable is a string pointer, the length of string and location of the start of the string in memory is provided. (The string itself is not in the 7-bytes, only its length and a pointer to its start.) If the variable is of type integer, the value is given (as of the time of utility execution). If the variable is of type real, the five bytes of the floating-point form of its value are given (as of the time of utility execution).

15.5.4 Controlling the Locations Assigned to Variables

For Applesoft and machine-language programs to agree on memory locations that are to be shared in order to communicate with one another, it is convenient to be able to predict where certain variables will occur even without doing a detailed analysis.

A very easy way to do this is to mention these variables at the very beginning of the program in a fashion analagous to the way most programmers handle DIM statements to create arrays. The first variable mentioned in a program is allocated the first seven bytes in the variable table; the second, second seven bytes; the third, the third seven bytes, etc.

If the variables are real or integer the actual value of the variable can be found at OFFSET + 2. If the variables are string-pointers then OFFSET + 2 specifies the length of the string and OFFSET + 3,4 points to its physical location in memory.

15.6 How User Memory is Allocated for Arrays

The pointer located in 107,108 ($6B,$6C) indicates the start of the array area of user memory; the pointer located in 109,110 ($6D,$6E) indicates its end.

The method of allocation of space for REAL arrays is shown in figure 15.6A; that for INTEGER arrays is shown in figure 15.6B; and that for STRING POINTER arrays is shown in figure 15.6C.

There is a high degree of compatibility between the methods of representation of array and simple variables. For example, the method of distinguishing the type of variable is identical, depending upon the high bit of the two ASCII-character form of the variable name.

All simple variables took seven bytes (two bytes for name + five bytes for data/pointer). There is no such simple rule for array variables. Because arrays may have different numbers of elements, array variables do not have a fixed size. The size depends on the number of dimensions and the size of each dimension.

Since the length of arrays is variable, there is no great advantage in padding out the length of integer and string pointer data elements with zeros to make them the same size as real variable elements. In arrays each element takes only the amount of space actually needed for data representation.

Because of the different lengths for different arrays, each array includes a pointer to specify where the next array is to be found. This is a single-byte offset pointer rather than a two-byte, complete-address pointer. The offset stored in this byte is the difference in position between the start of the array in which the offset appears and the start of the next array.

An array specification requires one byte to specify the number of dimensions and then a two-byte size for each of the dimensions. The size of the last dimension in the DIM statement is always stored first. The numeric value of the size is one larger than the value in the DIM statement because each dimension of a BASIC array always contains a zero element. Thus an array with DIM(2,3) does not contain just $2 \times 3 = 6$ elements; it contains $3 \times 4 = 12$ elements.

These elements are not allocated to memory in the traditional order for mathematical arrays and matrices. Instead, elements are assigned with

the rightmost index changing slowly. Thus for a two-dimensional array elements are not assigned to memory in fashion that goes across each row from left to right, taking rows from top to bottom. Instead, BASIC uses the curiously non-mathematical procedure of storing elements in order from top to bottom of each column, taking the columns in order from left to right.

Figure 15.6A
Layout of Type Real Array in Memory

Byte #	Description
1	1st Char of Name (+ASCII)
2	2nd Char of Name (+ASCII)
3	OFFSET pointer to next array - low byte
4	OFFSET pointer to next array - high byte
5	Number of dimensions (K)
6	Size+1 of Kth dimension - high byte
7	Size+1 of Kth dimension - low byte
...	
2K+4	Size+1 of 1st dimension - high byte
2K+5	Size+1 of 1st dimension - low byte
2K+6	Array Elements starting with 0 element, e.g. A(0,0) for 2-D array. Arrays are stored with right-most index ascending slowest, e.g. for DIM A(1,1) the order of storage would be A(0,0),A(1,0),A(0,1),A(1,1) Each element is stored in 5-byte form as per simple type-real variables If the array is dimensioned DIM (K1,K2,K3...) then the number of elements is (K1+1)*(K2+1)*(K3+1)* ... and OFFSET=6+2K+5*(K1+1)*(K2+1)*(K3+1)...

In real arrays, five bytes are used per data element: an exponent byte and four mantissa bytes. In integer arrays, two bytes are used per data element. This means that if array variables can be defined and stored as integers rather than as type real numbers that three bytes can be saved per data element.

Figure 15.6B
Layout in Memory of Type Integer Array

Byte #	Description
1	1st Char of Name (-ASCII)
2	2nd Char of Name (-ASCII)
3	OFFSET pointer to next array - low byte
4	OFFSET pointer to next array - high byte
5	Number of dimensions (K)
6	Size+1 of Kth dimension - high byte
7	Size+1 of Kth dimension - low byte
...	
2K+4	Size+1 of 1st dimension - high byte
2K+5	Size+1 of 1st dimension - low byte
2K+6	Array Elements starting with 0 element, e.g. A%(0,0) for 2-D array. Arrays are stored with right-most index ascending slowest, e.g. for DIM A%(1,1) the order of storage would be A%(0,0),A%(1,0),A%(0,1),A%(1,1) Each element is stored in 2-byte form with the high-byte stored first If the array is dimensioned DIM (K1,K2,K3...) then the number of elements is (K1+1)*(K2+1)*(K3+1)* ... and OFFSET=6+2K+2*(K1+1)*(K2+1)*(K3+1)...

For string pointers, three bytes are needed per data element: one byte for the length of the string and a two-byte address pointing to its start. As with simple variables, the string itself is not part of the data element. It is in the special string storage area allocated downward from HIMEM.

Figure 15.6C
Layout in Memory of String Pointer Array

Byte #	Description
1	1st Char of Name (-ASCII)
2	2nd Char of Name (+ASCII)
3	OFFSET pointer to next array - low byte
4	OFFSET pointer to next array - high byte
5	Number of dimensions (K)
6	Size+1 of Kth dimension - high byte
7	Size|1 of Kth dimension - low byte
...	
2K+4	Size+1 of 1st dimension - high byte
2K+5	Size+1 of 1st dimension - low byte
2K+6	Array Elements starting with 0 element, e.g. A$(0,0) for 2-D array. Arrays are stored with right-most index ascending slowest, e.g. for DIM A$(1,1) the order of storage would be A$(0,0),A$(1,0),A$(0,1),A$(1,1) Each element is stored in 3-byte form, the same as simple string pointer variables w/o the final `00' i.e. length of string, then address low-byte first If the array is dimensioned DIM (K1,K2,K3...) then the number of elements is (K1+1)*(K2+1)*(K3+1)* ... and OFFSET=6+2K+3*(K1+1)*(K2+1)*(K3+1)...

These patterns of allocation look much more difficult than they really are. This is well-illustrated by analysis of a nonsense program that uses all three types of arrays:

Figure 15.6D
Program to illustrate Allocation of Memory to Arrays

```
 10 DIM A%(2,3):DIM A$(3):DIM A(2,2)
 20 K=0
 30 FOR I=0 TO 2
 40    FOR J=0 TO 3
 50       K=K+1
 60       A%(I,J)=K
 70       PRINT "A%(";I;",";J;")=";A%(I,J);"  ";
 80    NEXT J: PRINT
 90 NEXT I
100 A$(0)="ZEROTH":PRINT A$(0)
110 A$(1)="FIRST":PRINT A$(1)
120 A$(2)="SECOND":PRINT A$(2)
130 A$(3)="THIRD":PRINT A$(3)
140 K=0
150 FOR I=0 TO 2
160    FOR J = 0 TO 2
170       K=K+1
180       PRINT "A(";I;",";J;")=";A(I,J);"  ";
190    NEXT J:PRINT
200 NEXT I
210 END
```

You can check the start and end of array space using the monitor:

```
] CALL -151
* 6C 6B
006C- 09
006B- 70 <Array storage starts at $0970>*
         6E 6D
006E- 09
006D- DA <End of Array storage + 1 at $09DA
*0970.09DA <Dump area of memory
           including arrays
```

The memory dump may be interpreted as follows:

```
$0970,71  => -ASCII 'A' -ASCII 'nul': Name
             of array = A%
$0972     => $21 Offset to next array: Next
             array starts at $0991
$0973,74  => 2-dimensional array
$0975,76  => 2nd dimension + 1 = 4
$0977,78  => 1st dimension + 1 = 3:
             DIM(2,3)
$0979,7A  => A%(0,0) = 1
$097B,7C  => A%(1,0) = 5
$097D,7E  => A%(2,0) = 9
$097F,80  => A%(0,1) = 2
$0981,82  => A%(1,1) = 6
$0983,84  => A%(2,1) = $Adec10
$0985,86  => A%(0,2) = 3
$0987,88  => A%(1,2) = 7
$0989,8A  => A%(2,2) = $B = dec11
$098B,8C  => A%(0,3) = 4
$098D,8E  => A%(1,3) = 8
$098F,90  => A%(2,3) = $C = dec12
------- End of A% Array -------
$0991,92  => +ASCII 'A'; -ASCII 'nul':
             Name of array = A$
$0993     => $13 Offset to next array: Next
             array starts at $09A4
$0994,95  => 1-dimensional array
$0996,97  => Dimension + 1 = 4: DIM A$(4)
$0998-9A  => A$(0) string length 6 at $0894
$099B-9D  => A$(1) string length 5 at $08AE
$099E-A0  => A$(2) string length 6 at $08C7
$09A1-A3  => A$(3) string length 5 at $08E1
------- End of A$ Array -------
$09A4,A5  => +ASCII 'A'; +ASCII 'nul':
             Name of array = A
$09A6     => $36 Offset to next array: Next
             array starts at $09DA
$09A7,A8  => 2-dimensional array
$09A9,AA  => 2nd dimension + 1 = 3
$09AB,AC  => 1st dimension + 1 = 3: DIM
             A(2,2)
$09AD-B1  => A(0,0) = 1 (5-byte floating point
             representation)
$09B2-B6  => A(1,0) = 4 (      "      )
$09B7-BB  => A(2,0) = 7 (      "      )
$09BC-C0  => A(0,1) = 2 (      "      )
$09C1-C5  => A(1,1) = 5 (      "      )
$09C6-CA  => A(2,1) = 8 (      "      )
$09CB-CF  => A(0,2) = 3 (      "      )
$09D0-D4  => A(1,2) = 6 (      "      )
$09D5-D9  => A(2,2) = 9 (      "      )
------- End of A Array (Real) -------
$09DA - End-of-Arrays
```

15.7
How Memory is Allocated for Strings

The method that Applesoft uses for allocating memory to strings is widely misunderstood. Even an experienced Apple user is likely to know little more than the fact that strings are allocated from HIMEM downward. This statement is true, but superficial.

Almost all of the information you need to know about memory allocation for strings has already been covered. All we need here is to put this information together with a good example and a discussion of some implications that we did not discuss earlier.

Let's look at a sample program that uses the string T$ in several different contexts.

```
10 T$ = "T1. LITERAL": PRINT T$:T$ =
"T2.LITERAL": PRINT T$: FOR I = 3 TO 6:
READ T$: PRINT T$: NEXT: STOP: DATA T3.
FROM. KEYBOARD: DATA T4. FROM.
KEYBOARD: DATA T5.FROM.KEYBOARD:
DATA T6.FROM.KEYBOARD: END
```

The first thing we note is that the string area contains four strings:

T6.FROM.KEYBOARD
T5.FROM.KEYBOARD
T4.FROM.KEYBOARD
T3.FROM.KEYBOARD

It does *not* contain the strings:

T2.LITERAL
T1.LITERAL

Where are those strings? They are literals, so they are embedded in the body of the program.

Notice that every time a string was input to the program from the keyboard, a copy of that string was stored in the string area — in spite of the fact that in every case the string was destined to be assigned to the same variable T$.

The first keyboard input, T3.FROM.KEYBOARD, was $10, or decimal 16 characters long. The next location available for string assignment started out at HIMEM, which in this case happened to be at $73EF because the *CALL A.P.P.L.E.* Program Global Editor which I used to get the alphanumeric-formatted dump occupied memory down to that location. The string was pushed in tail-end-first and began to push the

next location available for strings downward. Finally, when all $10 characters of the string were in place, the process stopped and we were left with the beginning of the string at $73E0 and we were ready to put the next string into memory working downwards from that spot.

The string pointer for the variable was set to this point, the beginning of the string and it, together with the length of the string was recorded at the appropriate location in the table of variable values. Notice that the string is in memory in the correct order so that it can be read directly from an alphanumeric memory dump or by character-by-character decoding as you work your way upward in memory.

The second input, T4.FROM.KEYBOARD, was also $10 characters long. It fills in memory tail-first down to its start at $73D0. When T5.FROM KEYBOARD was received, it filled down to $73C0; when T6.FROM.KEYBOARD was received, it filled in down to $73B0.

Let's look at the program dump/simple variables area to see what references we can find to these strings and their locations.

The program contains none. It refers to variables by the ASCII characters of its name. Going to the simple variable table we notice that T$ is the first variable in the table (because it was the first variable mentioned in the program.)

Thus the information about it is at zero offset from the start-of-simple-variables pointer value; i.e., $0850. The first two bytes specify its name, the next two specify the length of the string, and the next two specify location of the start of the string. Clearly there is no room to hold multiple lengths and starting locations, so only the most recently used string location can be mentioned. It is: 10 B0 73. Length = $10 = decimal 16. Start-of-string location $73B0.

There are several interesting implications here. As long as we continue to input new values of T$ we will continue to assign new memory locations to the strings that are entered in response to the INPUT commands, using up more and more memory for each new string as it is input.

If the FOR-NEXT loop in this program were changed so that the program continued asking for more and more new values of T$, the new values would continue taking up more and more memory. If nothing else happened the next-location-for-strings would work its way downward and eventually reach the top of the program variables and the computer would run out of memory!

Our tiny program with only one string variable is filling up the computer's memory with strings, yet it can only make contact with one of them! Useless garbage is filling up most of what once was our 'user free space.'

Usually before you get into trouble the computer is able to sense the problem and automatically undertake a process of garbage collection. This process determines which strings are unattached to a variable name and gets rid of them, compacting those strings that remain back toward HIMEM and thus making more space available.

In a large program with many variables the garbage collection process may be very slow. If you don't happen to recognize what is going on, it can be very disconcerting indeed to have the normal operation of a program suddenly stop and the computer apparently doing nothing for as long as a minute or more! Such a minute can seem to be an eternity and you become totally convinced that your program has bombed.

There are also occasions when the computer runs out of memory in a way that does not trigger automatic garbage collection in time to avoid the dreaded 'OUT OF MEMORY' error. To avoid this and the possibility of a long wait at an inconvenient time, Applesoft gives you the capability to force garbage collection at a time of your choice.

You can keep track of the amount of free space you have between the top of variables/arrays and the bottom of the string area by means of the function FRE( ). For example, PRINT FRE(0) will print the amount of free space currently available. X = FRE(0) will assign the amount available to the variable X so that it can be used and/or tested by your program.

A nice thing about FRE( ) is that it forces garbage collection before it reports. It does this so that it can give you the true amount of space available for your use, not an amount artificially reduced by unattached strings.

If you have a program that you have reason to suspect might need garbage collection, why not use a FRE( ) just after a long printout and have the garbage collection go on while the user is reading the screen and before he gives a go-ahead signal?

15.8 What You Can Do If You Don't Have Enough Applesoft 'User Memory'

15.8.1 Memory Conservation

The best way to keep from running out of

memory is to conserve it rather than squeeze more into your computer than will fit.

The real key to memory conservation is not in programming tricks, but in careful analysis and planning of your programs. Careful planning and structuring can eliminate the need for unnecessary functions. Careful modularization and set-up of subroutines can allow you to use the same code over and over again without degrading the readability of your programs.

But don't overemphasize memory conservation at the expense of other desirable features unless it is absolutely necessary.

15.8.2 Making Unavailable Memory Available

It is often possible, if you are willing to live with a non-standard programming environment, to make special changes in the system that will free additional memory. However, it is a good idea to hold such techniques in reserve for use when really needed, rather than to operate routinely in a non-standard environment.

A very common technique, if you have a language card or equivalent RAM for the top 16K of Apple memory, is to move the DOS from its standard location to a high memory position. Another common technique is to strip unneeded modules out of the DOS so that the space can be used for your own machine-language programs.

15.8.3 Overlaying, Chaining, and Re-using the Same Memory

Sometimes programs just get too big to fit into memory all at once. Other times the programs themselves are not too big but you would like to use them with huge data files inside the computer that, together with the program, would exceed available memory capacity.

When you need a method to squeeze a program into your computer, which won't fit all at once, an obvious solution is to find some way to make the program work without all of the program in it at one time.

One way to do this is to split your program into independent modules that can share the same environment of variables and strings. Also provide some means for changing from module to module, and have only one of the modules in the computer at a time. The process is very simple conceptually.

Suppose you have a program that is too large to run on your computer but can be broken down into three modules '1','2' and '3', each of which will fit into the available memory together with all the variables and strings needed by the entire program. For the moment let's assume that module '1' is the largest.

If we set up and run '1' it will occupy an area of memory from the start-of-program pointer to the end-of-program pointer. Variables, arrays, free space, and strings will occupy the remaining area to HIMEM. What we want to do now is to change the program in the area from the start-of-program pointer to the end-of-program pointer to Module B, *without* changing the rest of the environment, then transfer control to it. Later we may want to go back to Module A or go on to Module C, still maintaining the same basic environment but with changes created during the running of module '2'.

LOADING '2' as an Applesoft program will not solve the problem because it will destroy the rest of the environment.

However, if we had previously LOADED '2' and then BSAVED only the program itself, i.e., that portion of memory from the start-of-program pointer to the end-of-program pointer, we could BLOAD the '2' into this area of memory without changing the environment. All we would have to do is to transfer to the correct location in '2' and make sure that the various pointers associated with the program didn't get mixed up in the process. The same procedure could be used to go on to '3' or to go back to '1'

There are a number of variations on this basic process. If B or C happen to be larger than '1' you will have to find out the end of the largest module, then set the start-of-simple-variables pointer to that point before running module '1' so that the variables will not be allocated into space, which will later be destroyed by having a larger module BLOADed on top of it.

With programs that run sequentially from '1' to '2' to '3', the process is very simple and no special utility is needed. However, for those who don't want to fiddle around with pointers, Apple has provided a CHAIN utility and appropriate directions for its use in the DOS 3.2.1 system master.

Suppose you have a two-part program stored on two files: 'PART.ONE' and 'PART.TWO'. If you wish to chain from 'PART.ONE' to 'PART.TWO', all you need to do is insert the following two lines to be executed in 'PART.ONE':

```
PRINT CHR$(4);"BLOAD CHAIN",A520
CALL 520"PART.TWO"
```

(NOTE: There must be no space or other character between the 520 and the quotation mark.)

You can chain back to 'PART.ONE' in the same way:

```
PRINT CHR$(4);"BLOAD CHAIN",A520
CALL 520"PART.ONE"
```

(NOTE: Don't depend upon the previous BLOAD to have set up the CHAIN. You cannot omit the BLOAD. The area you loaded to was in memory page 2, the character input buffer.)

In practical programs that use the overlay process, the problem of control is often handled by having a module '0' (command processor) that remains resident regardless of which module is in use. For example, '0' might be a menu-driven system to choose which of several graphics activities you wish to perform while '1','2', and '3' each contain one or more of these options.

If you select an option in a different module, then the part of the command processor associated with overlay control will undertake the loading of the correct module as well as transferring control to the correct position in it.

You can still use a CHAINing process here, but now you will have to include a copy of '0' with each of the modules and you will find yourself copying X on top of itself each time you make a change. If '0' is large this can involve a significant waste of both disk space and transfer time.

However, it is not difficult to write a utility that makes overlaying only a part of a program quite straightforward and tends to eliminate human errors in setting up pointers and transfer points. Here is such a utility adapted from one originally written by Dave Lingwood.

Figure 15.8A
Overlay Utility Subroutines

```
1
 ^...
 |... Module `0', the module not to be overlayed, goes here.
 v...
988
989 REM THE OVERLAY UTILITY SUBROUTINE OCCUPIES 990-998.
    990-992 CHOOSES DESIRED MODULE AND WHETHER TO BLOAD OR BSAVE
            YOU MAY WISH TO CHANGE THESE LINES OF CODE
    993-995 BLOADS DESIRED MODULE
    996-998 BSAVES DESIRED MODULE
        THE START-OF-MODULE SUBROUTINE IS LINE 999.
990 INPUT  "MODULE#";K:INPUT #"LOAD,SAVE OR NO CHANGE (S/L/N)?";A$:
    A$=LEFT(A$,1): IF A$="L" THEN 993
991 IF A$="S" THEN 996
992 RETURN: REM DEFAULT RESPONSE = NO CHANGE
993 GOSUB 999: PRINT CHR$(4);"BLOAD MODULE ";K;",A",A:REM LOAD THE MODULE
994 E=PEEK(-21920)+256*PEEK(-21919)+A+1:REM END-OF-PROG = BEGIN + LEN +1
995 POKE 175,E-256*INT(E/256):POKE 176,INT(E/256):RETURN
996 GOSUB 999: L=PEEK(175)+256*PEEK(176) + 1 - A :
    REM LEN = END-OF-PROG+1 - START-OF-PROG
997 PRINT CHR$(4);"BSAVE MODULE ";K;",A";A;",L";L: REM SAVE THE MODULE
998 PRINT "MODULE ";K;" SAVED (";L;" BYTES)": PRINT "END-OF-PROGRAM AT ";
    A+L REM NOTIFY USER OF ACTION     |
999 A = PEEK(121)+256*PEEK(122):RETURN:
      REM SUBROUTINE TO LOCATE BEGINNING OF MODULE WHICH IT BEGINS, I.E.
      ITSELF|
1000
  ^...
  |... Module `1' `2' or `3' (Modules to be overlayed)
  v...
64000
```

This utility depends upon having or being able to PEEK information about the location of the beginning of each module (from $79,7A = decimal 121,122) and the length of the module (from the BLOAD/BSAVE length information in the DOS at $AA60,61 = decimal −21920, −21919). It also depends upon the ability to PEEK the end-of-program pointer ($AF,$B0 = 175,176) and to reset it by POKEing.

Now lets examine how the utility located in lines 990-999 works. There are two subroutines in this package: One to do the overlaying, the other to mark the start point of a module to be overlayed.

The short subroutine in the package takes but a single line:

```
999  A = PEEK(121) + 256*PEEK(122):
     RETURN
```

It is put at the beginning of each program module. The memory locations PEEKed ($79,$7A (= decimal 121,122) are a zero-page pointer to the memory location of the next line number; i.e., to the next line of BASIC in the program after the single-line subroutine itself.

Thus this single line subroutine sets the variable A to the 'real' start of the overlay module, considering the single line subroutine as its pseudo-start.

The larger subroutine (lines 990-998) uses this information to determine where to BLOAD or BSAVE the modules which are to do the overlaying.

The large subroutine starts out by finding out what you want to do. Lines 990-992 determine which overlay module should be processed and whether you want that processing to be the saving or loading of that module.

If you want to BLOAD, control is transferred to the block of statements in 993-995. If you want to BSAVE, a module control is passed to the block of statements in 996-998. These are the lines of code that do the real work.

If you are going to BLOAD, line 993 first calls the small subroutine to find the start of the module to be BLOADED, then issues a command with the correct file name 'MODULE ';K and the correct starting address 'A' obtained from the small subroutine. It then PEEKs into the DOS to find the length of the module which it loaded and adds the starting address − 1 to compute the end-of-program address. Then it corrects the end-of-program pointer to reflect this corrected address and returns control to the location in module '0' from which it was called.

If you are going to BSAVE, line 996 calls the small subroutine to find the start of the module, then PEEKs the end-of-program pointer and computes the length of the module to be saved from the difference of the two. It then executes a BSAVE with the computed A (start) and L (length) values; prints out a report on what it has done and returns control to its point of call. (Usually you will want to stop once you have done this, but the subroutine format gives you the opton of continuing if you desire to do so.)

How do you set up a program to use this utility and system of overlaying?

In practice it is convenient to keep a program file of the entire program — Module '0', subroutines, and Modules '1', '2' and '3' as one long source program file.

When you are ready to set up the running version of the progam load up this whole file; delete the undesired modules, e.g. '2' and '3' when you want to set up '1'; then issue a GOSUB 990 to activate the utility and respond to the questions asked by saying that you want to SAVE module. '1'. The utility will set up and execute the saving of module '1'.

Repeat the process for modules '2' and '3'.

If you are going to set up overlayed programs, very often the housekeeping is simplified if you set up some conventions such as that module '1' uses BASIC line numbers 1001-1999; module '2', numbers 2001-2999, and module '3', numbers 3001-3999.

Then the process becomes totally mechanical:

1. Load the whole program,
   DEL 2000-4000,
   GOSUB 990,
   Respond module '1' and 'S'.
   The module '1' set-up will occur.
2. Load the whole program again,
   DEL 1000-1999 and 3000-4000,
   GOSUB 990,
   Respond module '2' and 'S'
   The module '2' set-up will occur.
3. Load the whole program again,
   DEL 1000, 2999
   GOSUB 990,
   Respond module '3' and 'S'
   The module '3' set-up will occur.

If you receive a report that the second or third module is larger than the first,

4. Take the larger end of program value,
   Add a little extra to allow for minor program changes, say 100 bytes.
   Then at the very beginning of module '0'
   set LOMEM: to this value.
   Restart the process at the beginning.

You may be surprised at this use of LOMEM: Remember a LOMEM: statement does not reset the start-of-program vector and hence the start-of-program location; it resets the start-of-simple-variables vector and hence the start-of-simple-variables location. The LOMEM: statement is functionally equivalent to, but easier and briefer than double-POKEing the numeric value specified into the start-of-simple-variables pointer ($69, $6A or decimal 105, 106).

You may find it reassuring to verify that this action has not moved the start-of-program pointer ($67, $68 or decimal 103,104) upward from its previous value (usually the default value $0801 or decimal 2817). Nothing is more reassuring than experimental verification of such assertions that may seem counter-intuitive.

With LOMEM reset, the program modules all begin at the same place as they did before, but the location for the program variables is overtly raised high enough so that there is adequate room for any of the overlays to fit in its entirety below the variables. Now no interference can occur if this is part of the set-up.

An EXEC file can easily be constructed to perform the whole process automatically, if desired. However I am not sure it is worth the bother.

Thus far we have talked about the utility and its set-up. Now let's discuss using the overlay system.

When running an overlayed program you may call the large subroutine of the overlay whenever you want to change overlays. You call the package in the same way and follow the same procedures (with prompting for human control of overlay changes), responding 'L' (for LOAD) rather than 'S' (for SAVE).

Later, when your program is stable and well debugged, you can bypass this level of direct human control. Just set up parameters K and A and do a GOSUB 993 rather than a GOSUB 990. The overlay will now occur automatically without any inquiry or human intervention.

Applesoft references functions by their RAM location, not by line number. Even if a function has the same line number in different modules (if there is any difference in the programs before the function(s) are defined) they will appear in different RAM memory locations. A somewhat different but comparable problem occurs with the ONERR GOTOs.

Chapter XVI
High-Resolution Graphics Display Memory Pages 32-63 & 64-95 ($2000-$3FFF & $4000-$5FFF)

16.1
Introduction

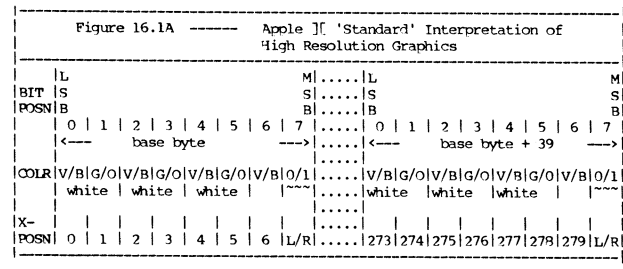
The Apple has a second type of graphic display, called high-resolution, or Hi-Res graphics. Like low-resolution (Lo-Res) graphics, two display-page buffer areas identified as Page 1 and Page 2 are assigned to this type of graphics. However, because the high-resolution graphics can display a great deal more fine-grained detail than can low-resolution, these display buffers must be a great deal larger — 8,192 bytes each rather than the 1,024 bytes, which sufficed for low-resolution graphics.

There are many similarities in organization and operation between low-resoluton and high-resolution graphics, but there are complications that make it harder to keep track of what is going on and to use it effectively.

Apple Computer Co. describes this capability as follows:

> "When your Apple is in the high-resolution mode, it can display 53,760 dots in a matrix 280 dots wide and 192 dots high. The screen can display black, white, violet, green, red, and blue dots, *although there are some limitations concerning the color of individual dots...*(emphasis added by author).
>
> "Each dot on the screen represents one bit from the picture buffer. Seven of the eight bits in each byte are displayed on the screen, with the remaining bit used to select the colors of the dots in that byte. Forty bytes are displayed on each line of the screen. The least significant bit (first bit) of the first byte in the line is displayed on the left edge of the screen, followed by the second bit, then the third, etc. The most significant (eighth) bit is not displayed. Then follows the first bit of the next byte, and so on. A total of 280 dots are displayed on each of the 192 lines of the screen.
>
> On a black-and-white monitor or TV set, the dots whose corresponding bits are "on" (or equal to 1) appear white; the dots whose corresponding bits are "off" (or equal to 0) appear black. On a color monitor or TV, it is not so simple. If a bit is "off" its corresponding bit will always be black. If a bit is "on" its color will depend upon the *position* of that dot on the screen. If the dot is in the leftmost column on the screen (called 'column 0', or in any even-numbered column, then it will appear violet. If the dot is in the rightmost column [column 279] or any other odd-numbered column, then it will appear green. If two dots are placed side-by-side, then will both appear white.
>
> If the undisplayed bit of a byte is turned on then the colors blue and red are substituted for violet and green respectively. Thus, there are six colors available in the high-resolution graphics mode, subject to the following limitations:
>
> 1) Dots in even columns must be black, violet or blue
>
> 2) Dots in odd columns must be black, green, or red
>
> 3) Each byte must be either a violet/green byte or a blue/red byte. It is not possible to mix green and blue, green and red, violet and blue, or violet and red in the same byte.
>
> 4) Two colored dots side by side always appear white, even if they are in different bytes."

This is the standard Apple interpretation of the Apple II system graphics. Its color/position /resolution characteristics are shown symbolically in figure 16.1A. All their software documentation uses this view. Its great advantage is that it makes the Apple II graphics look like bit-mapped graphics. Its greatest disadvantage is that it creates a great deal of confusion about high-resolution color and the fineness of resolution which can be achieved with an Apple.



When I discuss bit-mapped graphics, I mean that one bit somewhere in computer memory represents one distinguishable on-off dot position on the video display screen. A high-resolution display page of 8192 bytes contains 8 x 8192 = 65536 bits of memory. Deducting the eighth bit

of each byte, which is described as a color-control bit rather than a plotting bit, there are 53,760 bits available in display-page storage, one for each of the 280 by 192 display-points on the video screen. Turn that bit on, a dot appears at a particular location on the display-screen; turn it off, the dot disappears.

Unfortunately the dots are not all the same color, but that is no particular problem with a black-and-white TV set. Also, if you turn on the bit in an adjacent position in the same line, strange things happen — two adjacent dots can coalesce into a single white dot, even on a color TV. Nevertheless the concept remains fairly straightforward in spite of these complications.

16.2 Introduction to Use of High-Resolution Graphics

16.2.1 The Simple Way — Using the Applesoft BASIC Commands

The simplest and most convenient way to access high-resolution graphics is to use the graphics commands built into the Applesoft interpreter:

HGR To initialize in high-resolution graphics (page 1 - high resolution with 4 lines for text at bottom)

HGR2 Same as HGR except page 2

HCOLOR = (numeric value or expression) Sets high-resolution graphics color to that specified by the value of HCOLOR, which must be in the range 0 to 7 inclusive. Table 16.2A below gives the 'standard' color associated with each HCOLOR value and the HCOLOR values that are distinguishable from it on a black-and-white TV or monitor:

Figure 16.2A

H COLOR ON TV	Distinguishable shade on B/W TV
0 black1	1,2,3, 5,6,7
1 green	0, 3,4, 7
2 blue	0, 3,4, 7
3 white1	0,1,2, 4,5,6,
4 black2	1,2,3, 5,6,7
5 orange	0, 3,4, 7
6 violet	0, 3,4, 7
7 white2	0,1,2, 4,5,6,

Colors 0-3 have the MSB or color bit in each plot byte in the off condition; colors 4-7 have it in the on condition.

White1 (HCOLOR = 3) is created by a coalescence of green (HCOLOR = 1) and blue (HCOLOR = 2). White2 (HCOLOR = 7) is created by a coalescence of orange (HCOLOR = 5) and violet (HCOLOR = 6).

On a color TV or color monitor, since not all colors can appear at all bit-mapped positions, a single high-resolution dot will appear colored green or orange (depending upon MSB) if plotted at an odd x-coordinate. It will appear blue or violet if plotted at an even x-coordinate. If it is double-plotted at x and x + 1 the colors will coalesce to white.

Other than this, the two versions of white (and their corresponding two versions of black) are pretty much interchangeable until one gets into the fancy tricks of super high-resolution described in section 16.4.

HPLOT <value1>,<value2>

This version of HPLOT is used for point plotting. It plots a high-resolution dot at x-coordinate <value1> and y-coordinate <value2> using the current value of HCOLOR. If HCOLOR has not been assigned a value, the color is indeterminate and may even be the background color so that no plot seems to occur.

HPLOT <value1>,<value2> TO <value3>,<value4>

variants:

HPLOT <value1>,<value2> TO <value3>,<value4> TO <value5>,<value6> TO...

HPLOT TO <value3>,<value4>

These versions of HPLOT are used for line plotting.

The first version plots from x-coordinate <value1>, y-coordinate <value2> to x-coordinate <value3>, y-coordinate <value4>. The color is determined by the current HCOLOR. If no value has been assigned to HCOLOR, the color is indeterminate and may even be the background color so that no plotting appears to occur.

The first variant indicates that additional coordinate pairs preceded by the keyword 'TO' may be added at will, subject of course to the normal screen limits. and instruction length limits.

The second variant indicates that if the first coordinate pair is omitted, plotting will occur from the current cursor position. In this case the color of the line is determined by the color of

that last plotted point, even if the value of HCOLOR has subsequently been changed.

Warning: All versions of HPLOT must be preceded by HGR or HGR2, or equivalent machine-language initialization. Otherwise your whole program may be clobbered.

X-coordinates must be in the range 0 <=value<=279; Y-coordinate values must be in the range 0<=value<=191. If these rules are not followed an illegal quantity error message will be generated upon execution of the command.

Applesoft high-resolution graphics also has built in a group of high-resolution graphics commands that are quite different in concept and use than those thus far described. These include the following:

DRAW
XDRAW
ROT
SCALE
SHLOAD

These commands deal not with the plotting of individual points and lines, but with the manipulation of entire pictures (known as shapes) which are created and may be stored in memory as graphical data structures, then drawn with a single command. When they are drawn these special commands give one the option of specifying where they are to be drawn and whether they are to be changed in scale (drawn larger or smaller) or rotated from their original orientation. Discussion of these powerful, but not always easy to control, commands is deferred to section 16.5.

Note that if the system is in the mixed high-resolution graphics plus four lines of text, the plotting that occurs with y values in the range 160 through 191 will not be visible. It is also possible to plot on a different high-resolution screen-page than that which is being displayed. This will also result in invisible plotting. In either case, a single poke, which changes the area of memory being displayed, may cause this previously plotted, but invisible information to be displayed. This sort of thing is often done quite deliberately, for example to create animation effects without distracting the viewer's eyes. Changes are made in individual lines until the whole picture is drawn.

16.2.2 Information Useful in Pseudo-BASIC and Machine-Language Programming

The Apple II system has, in addition to the standard allocation of high-resolution memory pages, a built-in (software) allocation of many locations to the processing of high-resolution graphics information. In particular the Applesoft interpreter contains a number of high-resolution graphics-related subroutines. These do for high-resolution graphics the high-resolution analogs of the low-resolution graphics routines built into the monitor software.

These routines make heavy use of two groups of page zero memory locations, sometimes described respectively as containing external cursor data, and internal cursor data. The internal cursor data is derived from the external cursor data using selected monitor subroutines that are called automatically at appropriate times.

The external cursor data is not quite the same information specified by an Applesoft BASIC programmer to determine how the point is to be plotted, but it is easily derived from the Applesoft commands.

$00E0-$00E1 The x-coordinate of the point! (This requires 2 bytes, since the value can be greater than 255)

$00E2 The y-coordinate of the point
(This requires only 1 byte, since the value can be no greater than 189)

$00E4 The color masking byte.
(This is derived from HCOLOR and is a byte chosen from the color table ($F6F6-$F6FD) using HCOLOR as the look-up argument. The color masking table is documented in figure 16.2B:)

Figure 16.2B
Color Masking Table

```
$F6F6: $00 = 00000000 (hcolor=0) (black1)
$F6F7: $2A = 00101010 (hcolor=1) (green)
$F6F8: $55 = 01010101 (hcolor=2) (blue)
$F6F9: $7F = 01111111 (hcolor=3) (white1)
$F6FA: $80 = 10000000 (hcolor=4) (black2)
$F6FB: $AA = 10101010 (hcolor=5) (orange-red)
$F6FC: $D5 = 11010101 (hcolor=6) (violet)
$F6FD: $FF = 11111111 (hcolor=7) (white2)
```

Note MSB = 0 for hcolor = 1 to 3
MSB = 1 for hcolor = 4 to 7

Also note that one color in each group has (ignoring the MSB) only odd bits on and plots only for odd x-coordinates. The other has only even bits on and thus plots only for even x-coordinates. Both whites have all bits on. They plot one color within their group for odd x-coordinates and the other for even x-coordinates.

$00E6 The page indicator
(This indicates the page plotting will occur on, and is totally independent of the page which is currently being displayed. $20 indicates screen page 1 is to be plotted upon; $40 indicates screen page 2.)

Before actual plotting occurs this information must be processed further to find the exact memory byte and bit(s) that need to be changed to effect the desired plotting action.

Memory location $00E5, which is physically a part of the above external cursor grouping, differs from the others in that it is usually not provided by the user, but is computed by a monitor subroutine, and thus might better be considered as a part of the internal cursor data:

$00E5 Horizontal Byte Index

Way down at the end of the chain of graphical processing actions implemented by various subroutines, the final act of plotting is implemented by the following set of machine-language instructions:

```
LDA $1C
EOR ($26),Y
AND $30
EOR ($26),Y
STA ($26),Y
```

with the Y-register always set to the value of $00E5 before execution of these instructions.

These internal cursor locations have the following uses:

$001C On-the-fly color byte or 'running color mask' (The color masking byte shifted for odd addresses and none black and white, otherwise the unmodified color mask byte.)

$0026-$0027 On-the-fly base address
(The address of the left end of the screen display line upon which the desired point appears. These locations are documented in depth in section 16.3)

$00E5 Horizontal byte index
(The address offset from the base address in which the bit to be plotted may be found. Since there are seven plotting bits (plus one color bit) per plot byte, this is computed by an integer division of the horizontal screen coordinate (which is located in $00E0-$00E$) by 7.)

$0030 On-the-fly bit mask
(Specification of which of the seven bits in the selected byte corresponds to the point to be plotted. Computed from the remainder in the integer division which computed the byte index.)

The main subroutines in the monitor that do the processing of high resolution graphics are the following: (Note: The names are those used by the monitor. The commented code for each subroutine may be looked up in the monitor listing published as part of the *Apple II Reference Manual* furnished with each Apple II computer.

INITIALIZATION:

HGR ($F3E2) — High-resolution GRaphics
Displays page 1 in mixed mode ($C053, $C054). Sets page indicator ($00E6) to page 1 ($20). Sets on-the-fly color byte to zero (black1) and clears page 1 to black (all zeros).

HGR2 ($F3D8) — High-resolution GRaphics page 2
Displays page 2 in all graphics mode ($C052, $C055). Sets page indicator ($00E6) to page 2 ($40). Sets on-the-fly color byte to zero (black1) and clears page2 to black (all zeros)

COLOR:

HCOLOR ($F6F0) — High-resolution COLOR
With x-register containing the color index (0 to 7), this subroutine looks up the appropriate color mask from the table described by figure 16.2A and stores it in $E4.

BKGND ($F3F4) — BacKGrouND
With color mask from table 16.2 for relevant hcolor in accumulator, this subroutine puts this color into color mask byte parameter $1C and writes it to every location in the current memory page specified by $E6.

CURSOR POSITIONING

HPOSN ($F411) — Horizontal POSitioN
With the x-register containing the low-order bits of the horizontal screen coordinate, the y-register containing the high-order, and the a-register containing the vertical screen coordinate, this routine stores the registers in external cursor locations $E0, $E1 and $E2. Then using external cursor page indicator $E6, it computes and sets the internal cursor parameters $26, $27, $30 and $E5 to the same position and the internal cursor color mask $1C to correspond to the external cursor color mask $E4.

INTX ($F465) — step INTernal cursor X-coordinate
At entry, y-register is preset with current horizontal byte index. This routine then modifies the internal cursor ($1C, $E5, $30 and Y-register) to increment or decrement screen x-coordinate by 1. If the N-flag is positive (bit = 0), increment; if the N-flag is negative (bit = 1), decrement. Wrap-around occurs if you increment or decrement beyond end of visible screen.

INCRX ($F48A) — INCRement cursor X-coordinate
DECRX ($F467) — DECRement cursor X-coordinate
Perform the actual incrementing or decrementing for INTX.

INTY ($F4D3) — INTernal cursor Y-coordinate
Modifies the internal cursor base address (in

$0026-$0027) to increment or decrement screen y-coordinate by 1. If the N-flag is positive (bit = 0), decrement; if the N-flag is negative (bit = 1), increment. If increment or decrement moves you off the top or bottom of the screen, wrap-around occurs to other edge.

INCRY ($F504) — INCRement Y-coordinate
DECRY ($F4D5) — DECRement Y-coordinate
Perform the actual incrementing or decrementing for INTY.

IPOSN ($F5CB) — Internal POSitioN
Sets the external cursor coordinates in $E0-$E2 to values which correspond to the current internal cursor position.

PLOTTING SUBROUTINES

HPLOT ($F457) — High-resolution PLOT
With x- and y-coordinate positions in the x-, y-, and a-registers as per PHOSN, this subroutine plots a point by calling HPOSN and the PLOT.

PLOT ($F45A) — PLOT a point
Plot a point using the internal cursor data. This subroutine performs the five instructions specified earlier as the final step in point-plotting. Note that the Y-register as well as the internal cursor memory locations must be preset.

HLINE ($F53A) — High-resolution LINE
Preset with x-coordinate in the a- and x-registers and with y-coordinate in the y-register. This routine then draws a line from the internal cursor position to the location specified by the registers. On exit, it leaves the external cursor data corresponding to the input and the internal cursor data corresponding to the last point on the line. (Note: If internal and external cursor data were not the same when subroutine was called, an off-set occurs. If this results in plotting an off-screen point, wrap-around occurs.)

16.3 Similarities and Differences in Organization and Memory-Mapping Between High-Resolution Graphics and Text/Low-Resolution Graphics

We may envision that a high-resolution graphics display-page is divided into sub-pages or macro-lines that cover *exactly* the same screen display areas as the correspondingly numbered text/low-resolution graphics macro-lines, with each macro-line capable of holding the same number of high-resolution graphcis text characters in exactly the same positions as a text macro-line.

Thus figure 16.3A is essentially identical to the corresponding low-resolution diagram, figure 14.3C, except that the unqualified notation DL has been replaced by the more explicit notation CDL, for *Character* Display Line. The mention in the heading that a new term GDL for *Graphic* Display Line to represent a 'slice' taken out of the CDL.

Figure 16.3A ----- Internal Structure of the High-Resolution Graphics Display Area
Eight Macro-Lines of 120 Characters Each (ML00 - ML07) + 8 bytes
Each Macro-Line contains 3 Character-Display Lines (CDL00 - CDL23)
Each Character Display Line (CDL) is made up of 8 'slices
Each 'slice' is a one-dot-high Graphic Display line (GDL000 - GDL191)

ML00	<1st 40 chars=CDL00>	<2nd 40 chars=CDL08>	<3rd 40 chars=CDL16> <8bytes>
ML01	<1st 40 chars=CDL01>	<2nd 40 chars=CDL09>	<3rd 40 chars=CDL17> <8bytes>
ML02	<1st 40 chars=CDL02>	<2nd 40 chars=CDL10>	<3rd 40 chars=CDL18> <8bytes>
ML03	<1st 40 chars=CDL03>	<2nd 40 chars=CDL11>	<3rd 40 chars=CDL19> <8bytes>
ML04	<1st 40 chars=CDL04>	<2nd 40 chars=CDL12>	<3rd 40 chars=CDL20> <8bytes>
ML05	<1st 40 chars=CDL05>	<2nd 40 chars=CDL13>	<3rd 40 chars=CDL21> <8bytes>
ML06	<1st 40 chars=CDL06>	<2nd 40 chars=CDL14>	<3rd 40 chars=CDL22> <8bytes>
ML07	<1st 40 chars=CDL07>	<2nd 40 chars=CDL15>	<3rd 40 chars=CDL23> <8bytes>

In the high-resolution graphics environment, instead of a character being represented by a single 8-bit ASCII code character, the visible character is created by a 7 × 8 matrix of off-on dots made up of seven dots in each of eight macro-line slices (see figure 16.3B).

Figure 16.3B – Character 'A' in Top-Left Corner of Hi-Res Graphics Screen (Character 0 of Macro-line 0)

Slice 0				*				Byte $2000 (8192):Bit value=$8(8)
Slice 1			*		*			Byte $2400 (9216):Bit value=$10(16)
Slice 2		*				*		Byte $2800 (10240):Bit value=$22(34)
Slice 3		*				*		Byte $2C00 (11264):Bit value=$22(34)
Slice 4		*	*	*	*	*		Byte $3000 (12288):Bit value=$3E(52)
Slice 5		*				*		Byte $3400 (13312):Bit value=$22(34)
Slice 6		*				*		Byte $3800 (14336):Bit value=$22(34)
Slice 7								Byte $3C00 (15360):Bit value=$0(0)
bit values of dots	64	32	16	8	4	2	1	
	$40	$20	$10	$8	$4	$2	$1	

Each of the seven on-off dot positions is controlled by a single memory bit, all from the same byte of memory. The eighth bit in that byte is not displayed, but is used as a color selection bit. (We will also later describe it as the L/R, or Left/Right bit, when dealing with black-and-white pictures.)

Thus each of the eight slices requires its own byte of memory. Control of a 7 × 8 character block matrix requres eight bytes instead of the single byte used by the ASCII code when in the Text mode.

The method of bit assignment is totally straightforward, as may be seen in figures 16.3B and 16.3C. The individual bit values may be computed from the positions where bits are desired to be on and the values associated with the bit values as shown on the bottom of the figures.

Figure 16.3C -- Character 'B' in Position One Space to the Right of the Top-Left Corner of Hi-Res Graphics Screen (Character 1 of Macro-line 0)

Slice 0		*	*	*	*			Byte $2001 (8193):Bit value=$3C(60)
Slice 1		*				*		Byte $2401 (9217):Bit value=$22(34)
Slice 2		*				*		Byte $2801 (10241):Bit value=$22(34)
Slice 3		*	*	*	*			Byte $2C01 (11265):Bit value=$3C(60)
Slice 4		*				*		Byte $3001 (12289):Bit value=$22(34)
Slice 5		*				*		Byte $3401 (13313):Bit value=$22(34)
Slice 6		*	*	*	*			Byte $3801 (14337):Bit value=$3C(60)
Slice 7								Byte $3C01 (15361):Bit value=$00(00)
bit values of dots	64 $40	32 $20	16 $10	8 $8	4 $4	2 $2	1 $1	

Figuring out what memory locations to use for a particular screen location is considerably less straightforward. There are several viable approaches:

1. You may figure it out from studying figure 16.3A and the description which follows of the high-resolution version of the wrap-around process previously described.

2. You can figure it out from the high-resolution graphics addressing plan that creates this wrap-around process. It is described in figure 16.3D.

Figure 16.3D --- Addressing Plan for High-Resolution Graphics

SELECT WHICH HI-RES PG 001 OR 010	SELECT WHICH OF 8 SLICES OF MACRO-LINE	SELECT WHICH OF 8 MACRO-LINES	SELECT WHICH OF 120 TEXT-CHARACTER POSITIONS OR 740 DOT POSITIONS IN CHOSEN MACRO-LINE SLICE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<------M.S.B. OF ADDRESS------>								<------L.S.B. OF ADDRESS------->							

3. You can look it up in a table. Figure 16.3E is a table applicable to page 1 of high-resolution graphics, and figure 16.3F is a corresponding table for page 2 of high-resolution graphics.

4. However, for most routine activities, the simplest and most effective method is to let the computer do the work for you. The system monitor and the Applesoft interpreter contain subroutines that will do the appropriate calculations (or table look-ups) and allow you to specify only the screen-position coordinates.

The Apple Atlas will help you find this software in cases where it is not convenient to use the Applesoft BASIC commands directly. The available software ranges from subroutines that perform essentially the same functions as the Applesoft BASIC commands to ones that perform only the byte-addressing for you.

It is important to note that the use of 8 bytes per character for this type of character representation (as opposed to the 1 byte per character with ASCII code and text/low resolution means of representation) shows clearly why the high-resolution memory buffers must be eight times the size of the text/low resolution buffers $2000 (8192 decimal) bytes instead of $400 (1096 decimal) bytes. However the number of characters that can be represented in a character position is not limited to the ASCII set, but is limited primarily by the ability of the human eye to recognize differences in the 2^56 (many thousands of billions) possible bit combinations. You can create Cyrillic, Hebraic, Arabic, Chinese, or any arbitrary type of character representations using this technique.

Each of the eight slices that make up the horizontal divisions of a character becomes a separately-controllable unit for high-resolution graphics display purposes. Since there are eight macro-lines and eight slices per macro-line, there are 8 × 8 = 64 high-resolution graphics macro-line slices in a high-resolution graphics display buffer.

As with the text area three display-lines of 40 character-widths each macro-line (and each slice of each macro-line) contain three blocks of 40 characters. Since each character position (or character position slice) is seven dots wide, the total graphics screen must be 7 × 40 = 280 dots wide.

As with text/low-resolution graphics, these 120 displayable character positions are combined with eight non-displayable scratchpad positions, which are made available to the peripheral slots as dedicated memory space for their individual uses.

The wrap-around process is identical at the whole-character or macro-line level, whether the characters are created by low-resolution or high-resolution techniques. Therefore the slices wrap-around to create three graphic display lines (GDL's) per macro-line slice or three packets of eight GDLs per macro-line. Since there are eight macro-lines per screen display 3*8*8* = 192 graphic display lines (GDL's) cover the total display screen of 64 slices for the whole screen. The same macro-line slices appear physically one-third of a screen apart — one in the top ⅓ of the screen, one in the middle ⅓, and one in the bottom ⅓ — so they appear 64 graphic display lines apart.

A careful examination of the detailed map of correspondences between screen locations (figure 16.3E, page 1, and figure 16.3F, page 2) clearly shows the macro-line and ⅓ screen repetition patterns we have discussed.

Note that within a macro-line each slice is separated from the adjacent one by exactly $400 (1024 decimal) locations. This means that we could consider that the high-resolution graphics area was actually organized into a graphics macro-line concept based on a logical display eight units high — in this case eight slices by 1024 dot-display positions — and that this super macro-line is folded twice. The detailed analysis at this level is left to the reader.

The inverse mapping from memory location to text line is included in the detailed memory map in the Atlas.

Figure 16.3E1 (First Half - Hi-Res Page 1)

Mapping between Screen Display Line Position & High-Resolution Graphics Screen Buffer Addresses

<------TOP 1/3 OF SCREEN----->			<----MIDDLE 1/3 OF SCREEN---->			<----BOTTOM 1/3 OF SCREEN---->		
GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs
.....			macro-line 0					
000	$2000~$2027	(8192~8231)	064	$2028~$204F	(8232~8271)	128	$2050~$2077	(8272~8311)
001	$2400~$2427	(9216~9255)	065	$2428~$244F	(9256~9295)	129	$2450~$2477	(9296~9335)
002	$2800~$2827	(10240~10279)	066	$2828~$284F	(10280~10319)	130	$2850~$2877	(10320~10359)
003	$2C00~$2C27	(11264~11303)	067	$2C28~$2C4F	(11304~11343)	131	$2C50~$2C77	(11344~11383)
004	$3000~$3027	(12288~12327)	068	$3028~$304F	(12328~12367)	132	$3050~$3077	(12368~12407)
005	$3400~$3427	(13312~13351)	069	$3428~$344F	(13352~13391)	133	$3450~$3477	(13392~13431)
006	$3800~$3827	(14336~14375)	070	$3828~$384F	(14376~14415)	134	$3850~$3877	(14416~14455)
007	$3C00~$3C27	(15360~15399)	071	$3C28~$3C4F	(15400~15439)	135	$3C50~$3C77	(15440~15479)
.....			macro-line 1					
008	$2080~$20A7	(8320~8359)	072	$20A8~$20CF	(8360~8399)	136	$20D0~$20F7	(8400~8439)
009	$2480~$24A7	(9344~9383)	073	$24A8~$24CF	(9384~9423)	137	$24D0~$24F7	(9424~9463)
010	$2880~$28A7	(10368~10407)	074	$28A8~$28CF	(10408~10447)	138	$28D0~$28F7	(10448~10487)
011	$2C80~$2CA7	(11392~11431)	075	$2CA8~$2CCF	(11432~11471)	139	$2CD0~$2CF7	(11472~11511)
012	$3080~$30A7	(12416~12455)	076	$30A8~$30CF	(12456~12495)	140	$30D0~$30F7	(12496~12535)
013	$3480~$34A7	(13440~13479)	077	$34A8~$34CF	(13480~13519)	141	$34D0~$34F7	(13520~13559)
014	$3880~$38A7	(14464~14503)	078	$38A8~$38CF	(14504~14543)	142	$38D0~$38F7	(14544~14583)
015	$3C80~$3CA7	(15488~15527)	079	$3CA8~$3CCF	(15528~15567)	143	$3CD0~$3CF7	(15568~15607)
.....			macro-line 2					
016	$2100~$2127	(8448~8487)	080	$2128~$214F	(8488~8527)	144	$2150~$2177	(8528~8567)
017	$2500~$2527	(9472~9511)	081	$2528~$254F	(9512~9551)	145	$2550~$2577	(9552~9591)
018	$2900~$2927	(10496~10535)	082	$2928~$294F	(10536~10575)	146	$2950~$2977	(10576~10615)
019	$2D00~$2D27	(11520~11559)	083	$2D28~$2D4F	(11560~11599)	147	$2D50~$2D77	(11600~11639)
020	$3100~$3127	(12544~12583)	084	$3128~$314F	(12584~12623)	148	$3150~$3177	(12624~12663)
021	$3500~$3527	(13568~13607)	085	$3528~$354F	(13608~13647)	149	$3550~$3577	(13648~13687)
022	$3900~$3927	(14592~14631)	086	$3928~$394F	(14632~14671)	150	$3950~$3977	(14672~14711)
023	$3D00~$3D27	(15616~15655)	087	$3D28~$3D4F	(15656~15695)	151	$3D50~$3D77	(15696~15735)
.....			macro-line 3					
024	$2180~$21A7	(8576~8615)	088	$21A8~$21CF	(8616~8655)	152	$21D0~$21F7	(8656~8695)
025	$2580~$25A7	(9600~9639)	089	$25A8~$25CF	(9640~9679)	153	$25D0~$25F7	(9680~9719)
026	$2980~$29A7	(10624~10663)	090	$29A8~$29CF	(10664~10703)	154	$29D0~$29F7	(10704~10743)
027	$2D80~$2DA7	(11648~11687)	091	$2DA8~$2DCF	(11688~11727)	155	$2DD0~$2DF7	(11728~11767)
028	$3180~$31A7	(12672~12711)	092	$31A8~$31CF	(12712~12751)	156	$31D0~$31F7	(12752~12791)
029	$3580~$35A7	(13696~13735)	093	$35A8~$35CF	(13736~13775)	157	$35D0~$35F7	(13776~13815)
030	$3980~$39A7	(14720~14759)	094	$39A8~$39CF	(14760~14799)	158	$39D0~$39F7	(14800~14839)
031	$3D80~$3DA7	(15744~15783)	095	$3DA8~$3DCF	(15784~15823)	159	$3DD0~$3DF7	(15824~15863)

Figure 16.3E2 (Second Half - Hi-Res Page 1)

Mapping between Screen Display Line Position & High-Resolution Graphics Screen Buffer Addresses

<------TOP 1/3 OF SCREEN----->			<----MIDDLE 1/3 OF SCREEN---->			<----BOTTOM 1/3 OF SCREEN---->		
GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs
.....			macro-line 4					
032	$2200~$2227	(8704~8743)	096	$2228~$224F	(8744~8783)	160	$2250~$2277	(8784~8823)
033	$2600~$2627	(9728~9767)	097	$2628~$264F	(9768~9807)	161	$2650~$2677	(9808~9847)
034	$2A00~$2A27	(10752~10791)	098	$2A28~$2A4F	(10792~10831)	162	$2A50~$2A77	(10832~10871)
035	$2E00~$2E27	(11776~11815)	099	$2E28~$2E4F	(11816~11855)	163	$2E50~$2E77	(11856~11895)
036	$3200~$3227	(12800~12839)	100	$3228~$324F	(12840~12879)	164	$3250~$3277	(12880~12919)
037	$3600~$3627	(13824~13863)	101	$3628~$364F	(13864~13903)	165	$3650~$3677	(13904~13943)
038	$3A00~$3A27	(14848~14887)	102	$3A28~$3A4F	(14888~14927)	166	$3A50~$3A77	(14928~14967)
039	$3E00~$3E27	(15872~15911)	103	$3E28~$3E4F	(15912~15951)	167	$3E50~$3E77	(15952~15991)

.....macro-line 5.....								
040	$2280~$22A7	(8832~8871)	104	$22A8~$22CF	(8872~8911)	168	$22D0~$22F7	(8912~8951)
041	$2680~$26A7	(9856~9895)	105	$26A8~$26CF	(9896~9935)	169	$26D0~$26F7	(9936~9975)
042	$2A80~$2AA7	(10880~10919)	106	$2AA8~$2ACF	(10920~10959)	170	$2AD0~$2AF7	(10960~10999)
043	$2E80~$2EA7	(11904~11943)	107	$2EA8~$2ECF	(11944~11983)	171	$2ED0~$2EF7	(11984~12023)
044	$3280~$32A7	(12928~12967)	108	$32A8~$32CF	(12968~13007)	172	$32D0~$32F7	(13008~13047)
045	$3680~$36A7	(13952~13991)	109	$36A8~$36CF	(13992~14031)	173	$36D0~$36F7	(14032~14071)
046	$3A80~$3AA7	(14976~15015)	110	$3AA8~$3ACF	(15016~15055)	174	$3AD0~$3AF7	(15056~15095)
047	$3E80~$3EA7	(16000~16039)	111	$3EA8~$3ECF	(16040~16079)	175	$3ED0~$3EF7	(16080~16119)
.....macro-line 6.....								
048	$2300~$2327	(8960~8999)	112	$2328~$234F	(9000~9039)	176	$2350~$2377	(9040~9079)
049	$2700~$2727	(9984~10023)	113	$2728~$274F	(10024~10063)	177	$2750~$2777	(10064~11103)
050	$2B00~$2B27	(11008~11047)	114	$2B28~$362F	(11048~11087)	178	$2B50~$2B77	(11088~11127)
051	$2F00~$2F27	(12032~12071)	115	$2F28~$2F4F	(12072~12111)	179	$2F50~$2F77	(12112~12151)
052	$3300~$3327	(13056~13095)	116	$3328~$334F	(13096~13135)	180	$3350~$3377	(13136~13175)
053	$3700~$3727	(14080~14119)	117	$3728~$374F	(14120~14159)	181	$3750~$3777	(14160~14199)
054	$3B00~$3B27	(15104~15143)	118	$3B28~$362F	(15144~15183)	182	$3B50~$3B77	(15184~15223)
055	$3F00~$3F27	(16128~16167)	119	$3F28~$3F4F	(16168~16207)	183	$3F50~$3F77	(16208~16247)
.....macro-line 7.....								
056	$2380~$23A7	(9088~9127)	120	$23A8~$23CF	(9128~9167)	184	$23D0~$23F7	(9168~9207)
057	$2780~$27A7	(10112~10151)	121	$27A8~$27CF	(10152~10191)	185	$27D0~$47F7	(10192~18423)
058	$2B80~$2BA7	(11136~11175)	122	$2BA8~$2BCF	(11176~11215)	186	$2BD0~$2BF7	(11216~11255)
059	$2F80~$2FA7	(12160~12199)	123	$2FA8~$2FCF	(12200~12239)	187	$2FD0~$2FF7	(12240~12279)
060	$3380~$33A7	(13184~13223)	124	$33A8~$33CF	(13224~13263)	188	$33D0~$33F7	(13264~13303)
061	$3780~$37A7	(14208~14247)	125	$37A8~$37CF	(14248~14287)	189	$37D0~$37F7	(14288~14327)
062	$3B80~$3BA7	(15232~15271)	126	$3BA8~$3BCF	(15272~15311)	190	$3BD0~$3BF7	(15312~15351)
063	$3F80~$3FA7	(16256~16295)	127	$3FA8~$3FCF	(16296~16335)	191	$3FD0~$3FF7	(16336~16375)

Figure 16.3F1 (First Half - Hi-Res Page 2)

Mapping between Screen Display Line Position & High-Resolution Graphics Screen Buffer Addresses

<------TOP 1/3 OF SCREEN----->			<----MIDDLE 1/3 OF SCREEN---->			<----BOTTOM 1/3 OF SCREEN---->		
GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs
.....macro-line 0.....								
000	$4000~$4027	(16384~16423)	064	$4028~$404F	(16424~16463)	128	$4050~$4077	(16464~16503)
001	$4400~$4427	(17408~17447)	065	$4428~$444F	(17448~17487)	129	$4450~$4477	(17488~17527)
002	$4800~$4827	(18432~18471)	066	$4828~$484F	(18472~18511)	130	$4850~$4877	(18512~18551)
003	$4C00~$4C27	(19456~19495)	067	$4C28~$4C4F	(19496~19535)	131	$4C50~$4C77	(19536~19575)
004	$5000~$5027	(20480~20519)	068	$5028~$504F	(20520~20559)	132	$5050~$5077	(20560~20599)
005	$5400~$5427	(21504~21543)	069	$5428~$544F	(21544~21583)	133	$5450~$5477	(21584~21623)
006	$5800~$5827	(22528~22567)	070	$5828~$584F	(22568~22607)	134	$5850~$5877	(22608~22647)
007	$5C00~$5C27	(23552~23591)	071	$5C28~$5C4F	(23592~23631)	135	$5C50~$5C77	(23632~23671)
.....macro-line 1.....								
008	$4080~$40A7	(16512~16551)	072	$40A8~$40CF	(16552~16591)	136	$40D0~$40F7	(16592~16631)
009	$4480~$44A7	(17536~17575)	073	$44A8~$44CF	(17576~17615)	137	$44D0~$44F7	(17616~17655)
010	$4880~$48A7	(18560~18599)	074	$48A8~$48CF	(18600~18639)	138	$48D0~$48F7	(18640~18679)
011	$4C80~$4CA7	(19584~19623)	075	$4CA8~$4CCF	(19624~19663)	139	$4CD0~$4CF7	(19664~19703)
012	$5080~$50A7	(20608~20647)	076	$50A8~$50CF	(20648~20687)	140	$50D0~$50F7	(20688~20727)
013	$5480~$54A7	(21632~21671)	077	$54A8~$54CF	(21672~21711)	141	$54D0~$54F7	(21712~21751)
014	$5880~$58A7	(22656~22695)	078	$58A8~$58CF	(22696~22735)	142	$58D0~$58F7	(22736~22775)
015	$5C80~$5CA7	(23680~23719)	079	$5CA8~$5CCF	(23720~23759)	143	$5CD0~$5CF7	(23760~23799)
.....macro-line 2.....								
016	$4100~$4127	(16640~16679)	080	$4128~$414F	(16680~16719)	144	$4150~$4177	(16720~16759)
017	$4500~$4527	(17664~17703)	081	$4528~$454F	(17704~17743)	145	$4550~$4577	(17744~17783)
018	$4900~$4927	(18688~18727)	082	$4928~$494F	(18728~18767)	146	$4950~$4977	(18768~18807)
019	$4D00~$4D27	(19712~19751)	083	$4D28~$4D4F	(19752~19791)	147	$4D50~$4D77	(19792~19831)
020	$5100~$5127	(20736~20775)	084	$5128~$514F	(20776~20815)	148	$5150~$5177	(20816~20855)
021	$5500~$5527	(21760~21799)	085	$5528~$554F	(21800~21839)	149	$5550~$5577	(21840~21879)
022	$5900~$5927	(22784~22823)	086	$5928~$594F	(22824~22863)	150	$5950~$5977	(22864~22903)
023	$5D00~$5D27	(23808~23847)	087	$5D28~$5D4F	(23848~23887)	151	$5D50~$5D77	(23888~23927)
.....macro-line 3.....								
024	$4180~$41A7	(16768~16807)	088	$41A8~$41CF	(16808~16847)	152	$41D0~$41F7	(16848~16887)
025	$4580~$45A7	(17792~17831)	089	$45A8~$45CF	(17832~17871)	153	$45D0~$45F7	(17872~17911)
026	$4980~$49A7	(18816~18855)	090	$49A8~$49CF	(18856~18895)	154	$49D0~$49F7	(18896~18935)
027	$4D80~$4DA7	(19840~19879)	091	$4DA8~$4DCF	(19880~19919)	155	$4DD0~$4DF7	(19920~19959)
028	$5180~$51A7	(20864~20903)	092	$51A8~$51CF	(20904~20943)	156	$51D0~$51F7	(20944~20983)
029	$5580~$55A7	(21888~21927)	093	$55A8~$55CF	(21928~21967)	157	$55D0~$55F7	(21968~22007)
030	$5980~$59A7	(22912~22951)	094	$59A8~$59CF	(22952~22991)	158	$59D0~$59F7	(22992~23031)
031	$5D80~$5DA7	(23936~23975)	095	$5DA8~$5DCF	(23976~24015)	159	$5DD0~$5DF7	(24016~24055)

Figure 16.3F2 (Second Half - Hi-Res Page 2)

Mapping between Screen Display Line Position & High-Resolution Graphics Screen Buffer Addresses

<------TOP 1/3 OF SCREEN----->			<----MIDDLE 1/3 OF SCREEN---->			<----BOTTOM 1/3 OF SCREEN---->		
GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs	GDL	Hex Addrs	Decimal Addrs
.....macro-line 4.....								
032	$4200~$4227	(16896~16935)	096	$4228~$424F	(16936~16975)	160	$4250~$4277	(16976~17015)
033	$4600~$4627	(17920~17959)	097	$4628~$464F	(17960~17999)	161	$4650~$4677	(18000~18039)
034	$4A00~$4A27	(18944~18983)	098	$4A28~$4A4F	(18984~19023)	162	$4A50~$4A77	(19024~19063)
035	$4E00~$4E27	(19968~20007)	099	$4E28~$4E4F	(20008~20047)	163	$4E50~$4E77	(20048~20087)
036	$5200~$5227	(20992~21031)	100	$5228~$524F	(21032~21071)	164	$5250~$5277	(21072~21111)
037	$5600~$5627	(22016~22055)	101	$5628~$564F	(22056~22095)	165	$5650~$5677	(22096~22135)
038	$5A00~$5A27	(23040~23079)	102	$5A28~$5A4F	(23080~23119)	166	$5A50~$5A77	(23120~23159)
039	$5E00~$5E27	(24064~24103)	103	$5E28~$5E4F	(24104~24143)	167	$5E50~$5E77	(24144~24183)
.....macro-line 5.....								
040	$4280~$42A7	(17024~17063)	104	$42A8~$42CF	(17064~17103)	168	$42D0~$42F7	(17104~17143)
041	$4680~$46A7	(18048~18087)	105	$46A8~$46CF	(18088~18127)	169	$46D0~$46F7	(18128~18167)
042	$4A80~$4AA7	(19072~19111)	106	$4AA8~$4ACF	(19112~19151)	170	$4AD0~$4AF7	(19152~19191)
043	$4E80~$4EA7	(20096~20135)	107	$4EA8~$4ECF	(20136~20175)	171	$4ED0~$4EF7	(20176~20215)
044	$5280~$52A7	(21120~21159)	108	$52A8~$52CF	(21160~21199)	172	$52D0~$52F7	(21200~21239)
045	$5680~$56A7	(22144~22183)	109	$56A8~$56CF	(22184~22223)	173	$56D0~$56F7	(22224~22263)
046	$5A80~$5AA7	(23168~23207)	110	$5AA8~$5ACF	(23208~23247)	174	$5AD0~$5AF7	(23248~23287)
047	$5E80~$5EA7	(24192~24231)	111	$5EA8~$5ECF	(24232~24271)	175	$5ED0~$5EF7	(24272~24311)
.....macro-line 6.....								
048	$4300~$4327	(17152~17191)	112	$4328~$434F	(17192~17231)	176	$4350~$4377	(17232~17271)
049	$4700~$4727	(18176~18215)	113	$4728~$474F	(18216~18255)	177	$4750~$4777	(18256~18295)
050	$4B00~$4B27	(19200~19239)	114	$4B28~$562F	(19240~22063)	178	$4B50~$4B77	(19280~19319)
051	$4F00~$4F27	(20224~20263)	115	$4F28~$4F4F	(20264~20303)	179	$4F50~$4F77	(20304~20343)
052	$5300~$5327	(21248~21287)	116	$5328~$534F	(21288~21327)	180	$5350~$5377	(21328~21367)
053	$5700~$5727	(22272~22311)	117	$5728~$574F	(22312~22351)	181	$5750~$5777	(22352~22391)
054	$5B00~$5B27	(23296~23335)	118	$5B28~$562F	(23336~22063)	182	$5B50~$5B77	(23376~23415)
055	$5F00~$5F27	(24320~24359)	119	$5F28~$5F4F	(24360~24399)	183	$5F50~$5F77	(24400~24439)
.....macro-line 7.....								
056	$4380~$43A7	(17280~17319)	120	$43A8~$43CF	(17320~17359)	184	$43D0~$43F7	(17360~17399)
057	$4780~$47A7	(18304~18343)	121	$47A8~$47CF	(18344~18383)	185	$47D0~$67F7	(18384~26615)
058	$4B80~$4BA7	(19328~19367)	122	$4BA8~$4BCF	(19368~19407)	186	$4BD0~$4BF7	(19408~19447)
059	$4F80~$4FA7	(20352~20391)	123	$4FA8~$4FCF	(20392~20431)	187	$4FD0~$4FF7	(20432~20471)
060	$5380~$53A7	(21376~21415)	124	$53A8~$53CF	(21416~21455)	188	$53D0~$53F7	(21456~21495)
061	$5780~$57A7	(22400~22439)	125	$57A8~$57CF	(22440~22479)	189	$57D0~$57F7	(22480~22519)
062	$5B80~$5BA7	(23424~23463)	126	$5BA8~$5BCF	(23464~23503)	190	$5BD0~$5BF7	(23504~23543)
063	$5F80~$5FA7	(24448~24487)	127	$5FA8~$5FCF	(24488~24527)	191	$5FD0~$5FF7	(24528~24567)

16.4 Getting 560-Position Horizontal Resolution from the Apple

Unfortunately the 192 × 280 bit-mapped viewpoint, which is the standard Apple viewpoint of the capabilities of their machine, can be misleading and cause you to either overestimate or understimate the capabilities of their machine. The Apple II system is actually capable of plotting not at just 280, but at 560 different points along each line.

(*Warning:* Very early models of the Apple, which had only four colors (including black and white as colors) do *not* operate in the fashion described in this section, but can be made to do so with a very minor hardware modification which has been extensively documented in the literature as a means of expanding from four to six colors.)

If you have difficulty believing that the Apple II can plot at a resolution of 560 points, the following demonstration suggested by Bob Bishop (previously with Apple Computer Inc.) will make it uncontestable. This demonstration uses the system monitor to set bits in the correct locations and avoids the complexities of relating decimal addresses (required by BASIC POKE statements) to screen bit locations. First clear the display area of memory using the monitor move command and set the appropriate soft switches for high-resolution graphics with four lines of text at the bottom:

```
*2000:0
*2001<2000.3FFEM
*C050 C053 C057
```

(See the standard Apple documentation on the use of monitor commands from the keyboard if these typed-in monitor commands confuse you.)

Next, use monitor commands to turn on various bit-combinations in memory location $2000, which plots at the top left corner of the screen. (The low-order bit, value 1, displays in the leftmost of the 280 bit-mapped positions. The next bit, value 2, displays in the next of the 280 bit-mapped positions, etc.) The monitor command

```
*2000:1     turns on the first bit-mapped position.
*2000:2     turns on the second.
*2000:4
```

turns on the third, and so on. By following this pattern through the entire line of text, from $2000-$2027, you will get the 280 dot-positions that Apple standard documentation tells you you can get.

However, what about the color bit? Does it affect position as well as color? You can demonstrate this point by using a black-and-white monitor, or turning down the color controls on a color TV. The color bit is the most significant bit in the byte so its value is $80 (127 decimal). Let's try the same sequence of movements changing the sign bit from off to on at each bit position:

```
*2000:1
*2000:81
*2000:2
*2000:82
*2000:4
*2000:84
...
```

Note that for each of these monitor commands the dot shifted a tiny bit to the right. By following this pattern through the entire line of text, from $2000-$2027, you get 560 discretely and distinguishably different dot positions.

You can now conceive of the sign-bit as being a left/right bit for halves of the previous 280 bit-mapped locations. Now, the symbolic representation of the 280 position standard Apple bit-mapping pattern shown in figure 16.1A becomes the 560 position pattern shown in figure 16.4A.

Figure 16.4A ----- A 560-step interpretation of Apple][black and white High-Resolution Graphics

	L								M	L								M
BIT	S								S	S								S
POSN	B								B	B								B
	0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7	
	<--	base byte					-->			<--	base byte + 39					-->		
X-	0	0	0	0	0	0	0	0		5	5	5	5	5	5	5	5	
POSN	0	0	0	0	0	0	0	0	1	4	4	4	4	5	5	5	5	5
	0	1	2	3	4	5	6	7	8	6	7	8	9	0	1	2	3	4

Next, turn up the color controls if you are using a color display. If you cycle through the same monitor commands, the 560 line positions cycle through a color pattern:

```
(Line Position)
;0  ;1  ;2  ;3  ;4  ;5  ;6  ;7  ;8  ;...

(Color)
/Violet/Blue/Green/Orange/Violet/Blue/Green
      /Orange/Violet/ ...

(Intra-Cycle Count)
;0  ;1  ;2  ;3  ;0  ;1  ;2  ;3  ;0  ;...
(Count of Completed Cycles)
;0                ;1                ;2   ...
```

There are 140 violet points, 140 blue points, 140 green points, and 140 orange points in our 560 point line. Our color resolution in any of these colors can only be 140 positions across the screen! Thus, figure 16.1A, which shows the standard Apple pattern, can be reorganized for color as well as for black-and-white, this time into the pattern shown in figure 16.4B.

The simplicity and clarity of this viewpoint gives us a very easy-to-understand, albeit not too efficient or fast-plotting, method of writing a BASIC subroutine to permit 560-position resolution plotting as shown in figure 16.4B.

Figure 16.4B ----- A Simplified 140-step interpretation of Apple][High Resolution Color Graphics

	L								M	L								M
BIT	S								S	S								S
POSN	B								B	B								B
	0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7	
	<--	base byte					-->			<--	base byte + 39					-->		
COLR	V	B	G	O	V	B	G	O	V	V	B	G	O	V	B	G	O	V
X-								L/R										L/R
POSN	0		1		2		3	L/R		136+		137		138		139		L/R

Since graphics programs often get quite large and complicated for Applesoft graphics programmers, these amounts of space, even if you choose the best one for your particular situation, are often uncomfortably small.

Regardless of the situation, it can always be eased somewhat by careful planning and memory conservation. Don't forget 'dynamic conservation,' which occurs when a program is run, as well as 'static conservation,' when the program is written. Be sure to include garbage collection whenever needed to assure that space allocated for strings (which are no longer needed) is recovered. Otherwise you may be surprised from time to time by a totally unsuspected 'OUT OF MEMORY ERROR.'

You can also expand the area available for allocation. Obviously availability of a language card or equivalent and appropriate software to allow all of DOS to be moved out of the Applesoft allocation area and into ROM address space is a considerable boon.

Don't forget there are other ways, which require no new hardware, to expand the memory space available. Perhaps parts of the program can be performed by machine-language subroutines hidden away in areas not otherwise available to Applesoft (e.g., memory page 3 in the normally unused section $0300-$03CF, or in unused portions of DOS).

The 'Memory Allocation Patching' technique described in the next section can also recoup significant amounts of memory that might otherwise be wasted. It makes available program space that would be wasted because it is not contiguous to the remainder of the program.

A true escape from the absolute limits of the Applesoft allocation scheme is 'chaining,' because it allows you to use the same space over and over again for different segments of program. But you pay for the benefits of chaining with programming restrictions, an increase in planning and development time, and significant delays in program execution.

Obviously it is up to the program designer to estimate how much memory he is going to need and then make a choice of which technique or combinantion of techniques are most conveniently and effectively able to meet the requirement. Don't forget that programs ALWAYS take more space than originally planned and that your strategy should include plenty of opportunities for future expansion.

16.5 Procedure for Overcoming Memory Allocation Conflicts During Development of Applesoft High-Resolution Graphics Programs

16.5.1 Preamble

The Applesoft Interpreter only knows how to allocate memory to programs if that space is in a single, uninterrupted block. However, the high-resolution graphics pages are in the middle of the available block of memory. Instead of using HIMEM or Start-of-Program to force the interpreter to stay on only one side of the prohibited area, you can break your program into two modules, one goes below the prohibited area and the other will go above it. Then patch your program to fool the interpreter into believing that it has one big program with internal linkages.

This process places much less severe restrictions on your actions during further development or expansion of your program than do more drastic techniques such as CHAINing. However, you do not want to use this technique prematurely if there is no real need to do so. Although you can add onto the program, you must be very careful about internal modifications that might alter the validity of the module split.

In this section a procedure is presented that provides full protection and/or warnings to keep you out of memory allocation trouble. It starts with simple protective measures appropriate to moderate-size programs, switches you into the low-module 'memory allocation patching' procedure when it becomes necessary, and describes that procedure in step-by-step detail. At the end, when a program becomes so huge that it cannot fit into memory at one tie, it suggests the escape of chaining.

The procedure described below is nowhere near as long or complicated as it may seem on first glance. The length of the description is to make absolutely sure that you understand not just how to follow the procedure mechanically, but also how and why the procedure works. This will permit you to adapt the procedure and its concepts to other situations and environments.

16.5.2 The Procedure Step-By-Step

If you are developing a new program, start at #1. If you are planning a very large program where availability of sufficient memory space is likely to become a problem, be conscious of memory conservation techniques from the very beginning of program development. Don't forget the possibility of having two versions of your program: one with maximum readability and documentation and the other compacted by a utility program that squeezes out all unnecessary REMarks, line numbers, etc.

If you have an existing program that seems to be in trouble because of memory allocation conflicts between Applesoft and high-res graphics when it is working with default values of start-of-program and HIMEM, start at #8.

1. Evaluate your problem requirements and choose either the HIMEM or start-of-program changing strategy for protection against allocation conflicts. (figures 16.5A, B or C may be helpful.)

2. Write your program and use it normally until you run into an 'OUT OF MEMORY' condition. Save your program.

3. If program is protected against allocation conflict by a HIMEM change, go to step 4H; if it is protected by a start-of-program change, go to step 4L.

4H. (Protected by HIMEM Change). Set

HIMEM back to its original value (typically $9600). You are no longer protected against allocation conflicts. Reload the program. Check the end-of-program address [PRINT PEEK (175) + 256 *PEEK (176)]. If the program already extends into a graphics page, which you will be using for graphics, go to step 6. Otherwise go to step 5.

4L. [Protected by Start-of-Program Change]. Set start-of-program back to its original value (typically $0800). Reload the program. You are no longer protected against allocation conflicts.

5. The Applesoft interpreter is now operating in its default mode, allocating program and variables from the bottom upwards and strings from HIMEM downwards. You will no longer be OUT OF MEMORY. However, any code you write from now on is potentially subject to destruction by high-res drawing activities once the program grows large enough for conflict to occur again.

6. Continue developing your program and adding additional code as needed. Always save your program each time you make a change before running it.

7. If, as you continue to add to your program, you run into obvious conflicts, you will know the program has grown enough so that even with the extra space now provided it has grown into the conflict area. (Strong indications of conflict are strange characters on the screen, or destruction of part of your program. Unexplained computational malfunctions can also be signs of conflict). Go to 8.

If you do not run into obvious conflicts by the time you finish your program development, you may still have run into undetected conflicts. Check the end-of-arrays/beginning-of-free-space address [PRINT PEEK (109) + 256 * PEEK (110)]. If it is in the conflict area you will have to go to 9 to assure reliable operation of your program. If this test does not detect any hidden conflicts, relax! No further action is required — you need not do any memory allocation patching.

8. Verify that a true conflict exists by checking whether the end-of-arrays/bottom of free space [PRINT PEEK (109) + 256 * PEEK(110)] crosses the boundary into a high-res graphics area being used.

If conflict is verified, continue at 9.

If there is no allocation conflict, you have some other kind of bug. Clear it up, then start this procedure over at the beginning, if still appropriate.

9. Check the end-of-program address [PRINT PEEK (175) + 256 * PEEK (176)].

If the program is long enough to cross the boundary into the zone of conflict, or is within eight bytes of doing so, go to 10L. If it is not that long, go to 10S.

10S. (Arrive here if program does not need to be split before storing it as first module of program).

Add the statement GOTO *** to your program (*** is the line number you will use as the first line in the second module of the program. The eight bytes specified above are used to make sure there is space for the GOTO *** statement.)

Recheck the end-of-program address to verify that the program is still too short to cross over into the conflict area. If the recheck indicates a cross over, go to 10L.

Save the program to disk as the first module of final program.

You must have at least one statement in the second module of your program, even if it is just a REM or END statement. This will force the variables and arrays to be located above the graphics area and provide a starting point for further program development in the second module. Create a new program with at least one statement (having line number ***) and save it as the second module of the program.

Go to 11.

10L. (Arrive here if the program is too long to be used as the first module of the ultimate program.) You are now faced with the problem of figuring out where to cut off the program so that it won't cross into the graphics area. A straightforward way of doing this is to chop statements off at the end one at a time, checking the end-of-program address [PRINT PEEK (175) + 256 * PEEK (176)] until you are at least eight bytes below the boundary. Add to the program GOTO ***, where *** is the line number to be used for the first line of the second module. Recheck to verify that the program is still short enough and save as the first module of the program.

Reload the original (too long) version of the program. Delete the code saved in the first module. Save it as the second module (or at least that part of the second module that has thus far been completed. Go to 11.

11. If further program development causes you to run out of space after memory allocation patching procedure is performed, you may want to come back and recover the small amout of unallocated space between this module end and the bottom of the graphics space. Such refinement should be part of a sophisticated overall memory conservation program.

Program development in module 2 can proceed without special warnings, but any changes in module 1 should trigger a review and revalidation of the whole memory allocation status. Be especially careful that your program has not grown into the conflict area and that the GOTO *** address setting described in 13 is correct.

12. You're ready to patch the memory allocation addresses to link the first and second modules together around the graphics area(s).

A. Load the first module of the program.
B. Either: (1) CALL -151 to enter the monitor and make the investigations and changes indicated using hexadecimal addresses or (2) Make the investigations with PRINT PEEKs and changes with POKEs using the decimal form of the addresses.
C. Verify that the end-of-program address ($AF, $B0 or 175,176) is below any high-res area used by the program. If not, go back and follow the previously prescribed procedure properly.
D. Examine the beginning-of-program address for module 1 ($67,68 or 103,104). It should contain $0801 (decimal 2049), the starting address for the first module. Change it to $4001 (decimal 16385) or $6001 (decimal 24577), depending upon which high-res pages you wish to patch around.
E. Load the second module of the program.
F. Verify that the end-of-program address ($AF,$B0 or 175,176) does point to an address above the high-res page Write its value down! If not, go back and follow the previously prescribed procedure properly. Both modules are now verified as being in memory in their correct positions.
G. Change the beginning-of-program address ($67,$68 or 103,104) back to $0801 (decimal 2049). The beginning and ending pointers now contain both modules of the program and there is a big hole in the middle where the high-res graphics pages are located.

13. Now you must modify the GOTO *** instruction you put at the end of module 1. The end of this instruction is located at the position in memory you wrote down in instruction 12F.

The instruction is represented by eight bytes of memory organized as follows:

/1/ The first two bytes contain the address of the following line
/2/ The next two bytes contain the line number.
/3/ The next byte contains the GOTO token.
/4/ The next two bytes contain the GOTO line number.
/5/ The last byte contains all zeros ($00). As the very last statement of the program it differs from other statements by having an extra two bytes of zeros.

You must modify the first of these items, the address of the following line. Its value should be one greater than the original end-of-program value you wrote down, and its location should be seven less than that address in memory. Change it to point to the correct new position of that line: $4001 (decimal 16385) if the second module is located immediately over the primary high-res page, or $6001 (decimal 24577) if the second module is located immediately above the secondary high-res page.

15. You may continue developing and expanding your program subject to the warnings expressed in 11. If it grows so huge that you run out of memory you will have to resort to memory conservation techniques for short-term relief, or take the big plunge into chaining the program into parts that do not occupy memory at the same time.

16. Miscellaneous Comments: Your program size will be 32 or 64 sectors larger than the sum of the two modules, with the extra sectors representing the amount of high-res graphic space saved in the middle of the program.

If there is a particular high-res picture you want saved in that area, you can BLOAD it in before saving the program. However, remember that to use it you will have to initialize graphics without use of the automatic initialization features of the HGR or HGR2 commands; their use can destroy the picture before it could be shown.

Figure 16.5A
Memory Space Available to Applesoft Programmer Using HIGH-RES PRIMARY PAGE ONLY

1. Default: Neither HIMEM nor Start-of-Program changed
   $0800-$1FFF = $1800 (decimal 6144) bytes for program & variables
   $4000-$9600 = $5600 (decimal 22016) bytes for character strings
2. HIMEM changed to $1FFF (to protect program)
   $0800-$1FFF = $1800 (decimal 6144) for everything: program, variables & strings
3. Start-of-Program changed to $4000 (to protect program)
   $4000-$9600 = $5600 (decimal 22016) for everything: program, variables & strings
4. Start-of-Program changed and DOS disabled or moved to language card
   $4000-$BFFF = $8000 (decimal 32768) for everything: program, variables & strings
5. Memory Allocation Patching
   $0800-$1FFF patched to $4000-$9600 = $7E00 (decimal 32256) bytes
6. Memory Allocation Patching Combined with DOS Removal
   $0800-$1FFF patched to $4000-$BFFF = $9800 (decimal 38912) bytes

Figure 16.5B
Memory Space Available to Applesoft Programer using HI-RES SECONDARY PAGE ONLY

1. Default: Neither Start-of-Program nor HIMEM changed
   $0800-$3FFF = $3800 (decimal 14336) bytes for program & variables;
   $6000-$9600 = $3600 (decimal 13824) bytes for character strings
2. HIGHMEM changed to $3FFF (to protect program)
   $0800-$3FFF = $3800 (decimal 14336) for everything: program, variables & strings
3. Start-of-Program changed to $6000 (to protect program)
   $6000-$9600 = $3600 (decimal 13824) for everything: program, variables & strings
4. Start-of-Program changed and DOS disabled or moved to language card
   $6000-$BFFF = $8000 (decimal 32768) for everything
5. Memory Allocation Patching
   $0800-$3FFF patched to $6000-$9600 = $7E00 (decimal 32256) bytes
6. Memory Allocation Patching combined with DOS Removal
   $0800-$3FFF patched to $6000-$BFFF = $9800 (decimal 38912) bytes

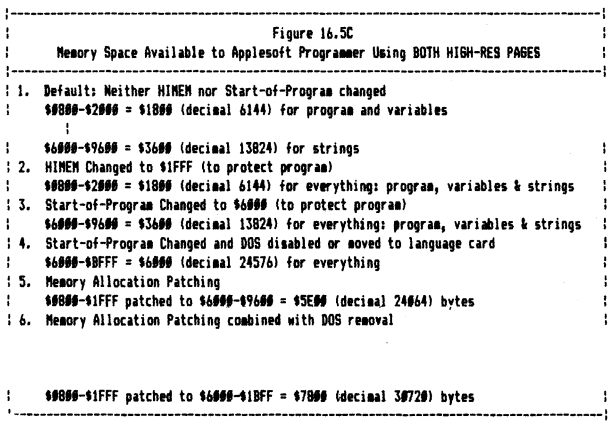


Figure 16.5C
Memory Space Available to Applesoft Programmer Using BOTH HIGH-RES PAGES

1. Default: Neither HIMEM nor Start-of-Program changed
   $0800-$2000 = $1800 (decimal 6144) for program and variables
   $6000-$9600 = $3600 (decimal 13824) for strings
2. HIMEM Changed to $1FFF (to protect program)
   $0800-$2000 = $1800 (decimal 6144) for everything: program, variables & strings
3. Start-of-Program Changed to $6000 (to protect program)
   $6000-$9600 = $3600 (decimal 13824) for everything: program, variables & strings
4. Start-of-Program Changed and DOS disabled or moved to language card
   $6000-$BFFF = $6000 (decimal 24576) for everything
5. Memory Allocation Patching
   $0800-$1FFF patched to $6000-$9600 = $5E00 (decimal 24064) bytes
6. Memory Allocation Patching combined with DOS removal

   $0800-$1FFF patched to $6000-$1BFF = $7800 (decimal 30720) bytes

16.6 Imbedding in Applesoft User Memory Space

The procedure described in 16.5.2 for allocating around the high-res graphics space can be applied to any arbitrary block of memory space within the region allocated by the Applesoft interpreter.

The non-Applesoft material will normally be BLOADed into place. The exact linkage into the Applesoft program depends upon the material and your program requirements, but usually can be accomplished using the BASIC PEEK, POKE, and CALL techniques described in earlier chapters.

Chapter XVII
The Disk Operating System
Default Location = Memory Pages 150-191 ($9600-$BFFF)

17.1
Introduction

The Apple disk drive is an electromechanical device capable of

1. writing information onto magnetic diskettes,
2. (a) permitting the physical removal of diskettes; (b) permitting the storage on an arbitrarily large number of diskettes for an arbitrary period of time; (c) permitting the return of any previously recorded diskette to the disk drive, and
3. reading the information back from the diskette for a specific use.

The diskette provides permanent storage; the information doesn't disappear when the computer is turned off.

The diskette also provides storage in large amounts. Each standard Apple diskette can hold about 140,000 characters of information; about 120,000 are available for holding user programs and data. You may find it convenient to think of that as about 40 pages of text.

The user sees this information as arranged in a file, a named collection of data on the diskette. It can contain text, programs, or data.

For example, if you have DOS active, have an Applesoft program in your computer and type in 'SAVE FISHBAIT', you copy the program onto the disk in the currently active disk drive and create a file of type Applesoft named 'FISHBAIT'. (If you already had a file with that name you would replace the older version.)

The DOS is a large and complicated package of software/firmware that facilitates the use of diskettes and disk drives. With the DOS you don't have to worry about how long the program is or which locations on the disk are occupied, which are available for storing a program, or exactly which ones actually to use when you store it.

DOS enables you to use such simple commands as LOAD FISHBAIT to retrieve the program FISHBAIT no matter where it was stored. It allows you to write text or numeric data to a disk file with little or no more trouble than printing it out on a printer.

The availability of such a combined hardware/software disk system makes a fundamental difference in the capabilities of a microcomputer like the Apple. Without them the Apple is a high quality toy. With them it is a real, practical tool for general-purpose computation, data processing, and word processing.

Disks were not available on the early Apple II's. The first bug-plagued version of DOS was not even released until the latter half of 1978. One reason is obvious. With the semiconductor memory technology available at the time the Apple was released, the available memory sizes were only 4K-16K and the disk operating system requires about 10K of memory. No matter how valuable a disk might be, that didn't leave much in the way of resources to do anything with a disk. Once larger capacity memory chips were available at reasonable prices and more Apples were expanded to 48K, that imbalance disappeared.

Yet a major legacy of earlier days remains in the DOS. The short-cuts and idiosyncrasies that were accepted in order to get around the early technology/cost problems have made the Apple DOS like no other. Not only is the method of issuance of commands (using PRINT statements with the CTRL-D) quite different from the industry norm, but there are annoying limitations (e.g., the special procedures needed for dealing with TEXT files).

Nevertheless, DOS has proven to be usable and practical. It allows the user to store and retrieve large amounts of data and programs quite conveniently *by name*, successfully insulating him from the details of how and where the information is physically recorded on the diskettes.

17.2
How Information Is Organized On Apple II Disks

17.2.1 Introduction

The disk drive works in many respects like a hybrid combination of record player and tape recorder/playback unit. Inside the black paper protective carrier of an Apple diskette is a 5 ¼" diameter disk of oxide-coated plastic (usually mylar) with a large hole in the middle, somewhat reminiscent of a 45RPM phonograph record.

The information is recorded on it circularly like on a 45RPM record too, except that instead of the information being recorded in one long circular spiral, it is recorded in 35 separate discrete circular tracks.

Like the pick-up of a record player, the pick-up (called the read head) can be moved inward or outward to playback (read) information in the outer tracks, the inner tracks, or anywhere between.

Physically, of course, the read heads are much more like those of a cassette tape player than they are like a phonograph pickup because the method of recording and playback is magnetic.

To read or write information, the disk drive physically moves the read head inward or outward on the disk, then waits for the rotation of the disk to bring the desired recording location to where it can be read electronically.

Because physical motion, no matter how fast, is slow compared to the electronic speeds of operation inside the computer, disk operations are quite slow compared to internal computer operations. Storage and retrieval information from a disk is just not competitive in speed or flexibility to storage of the same amount of information in the computer's internal electronic memory. (It doesn't have to be. It provides long-term storage like a book; the internal memory provides fast, short-term storage like a scratchpad.)

The retrieval process is fastest if you arrange to have the information you need coming under the read head just at the moment you ask for it. You can seldom achieve this kind of optimization without a great deal of work or just blind luck. If you are going to try to do so, you have to keep track of where everything is, how fast things are moving, and how long every relevant operation in the computer takes to perform. Except in very special and demanding cases, *forget it!*

We will soon be describing positions on a disk in terms of track and sector. If you arrange it so the information you need in a hurry is on the same track as the read head, then the only electromechanical time delay is that for the rotation of the diskette to bring the desired sector of information under the read head. You have eliminated the most time-consuming portion of the electromechanical process of seeking out the information, and you can speed up operations considerably.

The worst of all possible situations occurs when the read head starts out at the outermost track and must move all the way across the disk to the innermost track before it can begin its rotary search for the correct sector. If you are interested in rapid disk operations, you must avoid these situations.

17.2.2 Tracks

Apple diskettes have 35 tracks. Each consists of a circular recording path at a fixed distance from the center of the disk. Thus, each is like a very thin, flat ring, concentric with all the others. They are numbered from 0 (the outermost track) to 34 (the innermost track.)

To read (or write) information on a particular track, a read head (pickup) is physically moved inward towards the center of the disk, or outward towards its rim in discrete steps using a stepper-motor.

On occasion you may hear about phases or half-track positions, usually in conjunction with somebody's copy-protection scheme. When the disk read heads move in or out, it takes two steps of the stepper-motor to move from one track to the next. Thus, if you are willing to write a special program to perform this function, you can stop them at 70 different phase positions or phases, which include the 35 track positions and 35 positions halfway between tracks. Many copy protection schemes involve writing special items of information at such half-track positions.

17.2.3 Sectors

Each track is subdivided into sectors. The sector is the smallest unit of information that can be written to, or read from, a diskette at one time. Each sector contains one memory page (256 bytes) of usable information.

Each track contains the same number of sectors, so the physical length inches or centimeters of a sector on the outermost track is longer than that of a sector on the innermost track. However, sectors on the outermost track and the innermost track take the same amount of time to pass by the read head.

The actual format and method of encoding used in recording involves more pulses and bits than the 8 × 256 bits that eventually get to the memory page. The method of encoding used in earlier versions of the DOS (through version 3.2.1) permitted 13 sectors to be recorded on each track. An improved method of encoding introduced with DOS version 3.3 allows 16 sectors to be recorded on each track.

There is a small hole in the protective covering of your diskette through which you can see a hole in the plastic diskette itself. This hole, called an index hole, is used in many microcomputers to physically and electrically mark the first sector on each track. The Apple does not use this hole. Instead the sectors are coded as self-identifying to software in the DOS. This method of locating specific sectors using software is called software-sectoring or, more commonly, soft-sectoring.

17.2.4 Standard Overhead of Pre-Recorded Information on Apple Diskettes (Copy of DOS, VTOC, Directory)

Before we get down to details regarding the amount of space this makes available to us on a diskette, we must note that certain sectors on a diskette are reserved for specific uses:

1. All sectors on Tracks 0, 1, and 2 are reserved for a copy of the DOS. The presence of the DOS insures that you will be able to boot or re-boot the DOS with that particular diskette.

2. Sector 0 of Track 17 (the track which is equidistant from the innermost and outermost tracks) is reserved for the VTOC (Volume Table Of Contents).

3. The remainder of Track 17 is reserved for the disk directory.

We can now summarize the basic organization of, and available space on, Apple diskettes as shown in figure 17.2A.

Figure 17.2A
Diskette Organization Showing Standard Overhead Assignments

SECTOR =>	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Track 00	<-DOS IMAGE->															
Track 01	<-DOS IMAGE->															
Track 02	<-DOS IMAGE->															
Track 03																
.																
.																
Track 16																
Track 17	VTOC<-CATALOG->															
Track 18																
.																
.																
Track 34																

17.2.5 Summary of Diskette Track/Sector/Byte Capacity

We can now compute and summarize both the basic and effective capacity of Apple II diskettes as shown in figure 17.2B.

Figure 17.2B
Diskette Track/Sector/Byte Capacity

		DOS 3.3	Earlier
Tracks		35	35
Sectors	per track	16	13
	per diskette	560	455
Bytes	per sector	256	256
	per track	4096	3328
	per diskette	143360	116480

After exclusion of overhead for copy of DOS, Volume Table of Contents, and diskette directory, the following amounts of space remain for users:

User-Available Sectors	496	403
User Bytes	126976	103168

17.3 Diskette Organizational Concepts

This section discusses methods for organizing the disk to let users access information by file name rather than by track/sector.

17.3.1 The Volume Table of Contents

The Volume Table Of Contents (VTOC) is the kingpin of the diskette. If it is destroyed and cannot be reconstructed, you are in trouble. Subordinate to it, and equally important in finding your files on diskette, is the catalog. The VTOC and catalog must always be present. The copy of the DOS in Tracks 0, 1, and 2 is a convenience, but it is perfectly feasible to remove it and recoup the space for other uses.

As mentioned earlier, the VTOC and the catalog are placed on Track 17, midway between the inner- and outermost tracks on the disk. A little theoretical background is needed to explain why this is a smart place to put them.

To find a particular file of information you request, DOS starts at the VTOC. VTOC tells the computer where to find the first active catalog entry. The computer starts searching there for the name of the file you have specified. If it does not find it in the first sector of the catalog it searches, it chains its way to the second, then to the third, and so on, through all the entries in the catalog looking for the particular file you have specified by name. When it finds the name of the file you have specified in the catalog, DOS uses the catalog to look up the track and sector where the file begins.

We have learned that if the disk drive is to read a particular track and sector it must first move the read head inwards or outwards to the correct track (if it is not already on the correct track), then wait for the rotation of the diskette to bring the desired sector under the read head. Track-to-track movement using the stepper motor is relatively quite slow.

By putting the VTOC and the catalog on the same track we eliminate track-to-track movement until we have found the location of our data. This saves time. By having the VTOC and catalog (and hence the starting location for read head movement) at track 17 midway between the innermost and outermost tracks, the Apple disk subsystem minimizes the average amount of track-to-track movement needed to find the data.

Figure 17.3A shows the information included in the VTOC and the byte location of each element of information within it. Figure 17.3B shows the same information in lines of eight bytes, the way

the information would appear on a conventional hexadecimal dump.

Figure 17.3A
Internal Structure and Layout of the Volume Table Of Contents

BYTE	DESCRIPTION	DOS
$00	Not used	
$01	Track number of first catalog sector	$B3BC
$02	Sector number of first catalog sector	$B3BD
$03	Release # of DOS used to INIT the diskette VTOC appears on	$BDBE
$04-$05	Not used	
$06	Diskette volume number (1-254)	$B3C1
$07-$26	Not used	
$27	Maximum number of track/sector pairs that will fit in one file track/sector list sector (122 [$7A] for standard 256 byte sectors)	$B3E2
$28-$2F	Not used	
$30	Last track where sectors were allocated-track to allocate next	$B3EB
$31	Direction of track allocation (+1 or -1)	$B3EC
$32-$33	Not used	
$34	Number of tracks per diskette (35 [$23] - standard diskettes)	$B3EF
$35	Number of sectors per track (16 [$10]for DOS 3.3 or later; (13 for DOS 3.2.1 or earlier)	$B3F0
$36-$37	Number of bytes per sector (LO-HI) (256 [$100] for standard	$B3F1
$38-$3B	Bit map of free sectors in track 0 ($0)	$B3F3
$3C-$3F	Bit map of free sectors in track 1 ($1)	$B3F7
$40-$43	Bit map of free sectors in track 2 ($2)	
$44-47	Bit map of free sectors in track 3 ($3)	$B3FB
...	...	
...	...	
$BC-$BF	Bit map of free sectors in track 33 ($21)	$B478
$C0-$C3	Bit map of free sectors in track 34 ($34)	$B47B
$C4-$CF	Bit maps for additional tracks for non-standard diskettes with more than 35 tracks (expansion capability)	

NOTE:DOS Column indicates VTOC Sector buffer locn w/DOS at $9600 (48K Apple)

Notice in figure 17.3B how much of the space in the VTOC is blank and available for future expansion. Also note how many parameters, which remain fixed in the Apple disk drives as currently sold, are set up for possible change in the software of the VTOC.

Figure 17.3B
Volume of Table of Contents Layout: Hexadecimal Dump Format

	$0	$1	$2	$3	$4	$5	$6	$7
$00		<-TRK/SEC LINK->		DOS VER			VOL #	
$08								
$10								
$18								
$20								#T/S LE's
$28								
$30	LT ALLOC <-				TKS/DSK	SEC/TRK	<-SECTOR SIZE->	
$38	<- FREE SCTR BIT MAP, TRACK 0				-><	FREE SCTR BIT MAP, TRACK 1		->
$40	<-				2 -><			3 ->
$48	<-				4 -><			5 ->
$50	<-				6 -><			7 ->
$58	<-				8 -><			9 ->
$60	<-				$A (10)-><			$B (11)->
$68	<-				$C (12)-><			$D (13)->
$70	<-				$E (14)-><			$F (15)->
$78	<-				$10 (16)-><			$11 (17)->
$80	<-				$12 (18)-><			$13 (19)->
...	<-				... ...-><			... ... ->
...	<-				... ... -><			... ... ->
$B8	<-				$20 (32)-><			$21 (33)->
$C0	<-				$22 (34)-><	FUTURE EXPANSION		->
$C8	<- FUTURE EXPANSION				-><-	FUTURE EXPANSION		->
...	<- ...		...		-><-	...	...	->
...	<- ...		...		-><-	...	...	->
$F8	<- FUTURE EXPANSION				-><-	FUTURE EXPANSION		->

17.3.2 Bit Maps of Free Sectors in Each Track of the Diskette

A very obvious and important characteristic of the VTOC that we have not yet discussed is the existance of a Bit Map of Free Sectors for each track. Each track's Bit Map is a 4-byte long string of ones and zeros. A free sector is designated by a bit on (1) at the appropriate location in the map. If the sector is in use, the bit is off (0).

The pattern of mapping is as follows, using hexadecimal identification for sectors:

Byte	Sector On-Off Bit Positions
Base Address + 0:	F E D C B A 9 8
Base Address + 1:	7 6 5 4 3 2 1 0

In Case 1, if all sectors are free in these two bytes the bit pattern will be

Base Address + 0:	1 1 1 1 1 1 1 1 or ($FF)
Base Address + 1:	1 1 1 1 1 1 1 1 or ($FF)

In Case 2, if all are being used it will be

Base Address + 0:	0 0 0 0 0 0 0 0 or ($00)
Base Address + 1:	0 0 0 0 0 0 0 0 or ($00)

In Case 3, if only sectors 5 through A are free the pattern will be

Base Address + 0:	0 0 0 0 0 1 1 1 or ($07)
Base Address + 1:	1 1 1 0 0 0 0 0 or ($E0)

Notice that two bytes provide all the bits needed for DOS 3.3, 16-sector diskettes and more than enough for earlier 13-sector diskettes. The four bytes of space provided in the VTOC provides expansion space for up to 32 sectors per track. The bits in the extra bytes — Base Address + 2 and Base Address + 3 — are set to 0 and ignored. Thus the complete 4-byte hexadecimal representation of the three cases above becomes

Base Address + 0	$FF	$00	$07
Base Address + 1	$FF	$00	$E0
Base Address + 2	$00	$00	$00
Base Address + 3	$00	$00	$00

17.3.3 The Diskette Catalog (Directory)

The DOS organization to find a named file starts at the VTOC. Byte 1 of the VTOC points to the track of the first catalog sector; byte 2 points to its sector. In standard diskettes, the track number is 17 ($11) and the sector number is 15 ($0F).

Since a single sector does not contain enough space for a large catalog, each catalog sector uses its bytes 1 and 2 to point to succeeding catalog sector(s).

The organization of a catalog sector is shown in figure 17.3C.

Figure 17.3C
Structure and Layout of the Diskette Catalog (Directory)

BYTE	DESCRIPTION	DOS
$00	Not used	
$01	Track numbr of next catalog sector (17 or $11 for standard)	$B4BC
$02	Sector number of next catalog sector	$B4BD
$03-$0A	Not used	
$0B-$2D	First File: Catalog Description of File	
$2E-$50	Second File: Catalog Description of File	
$51-$73	Third File: Catalog Description of File	
$74-$96	Fourth File: Catalog Description of File	
$97-$B9	Fifth File: Catalog Description of File	
$BA-$DC	Sixth File: Catalog Description of File	
$DD-$FF	Seventh File: Catalog Description	

NOTE: DOS Column indicates location in Catalog/Directory sector buffer with DOS starting at $9600 (48K Apple). The catalog description of the current file will follow with the offsets indicated for the first file.

Figure 17.3D Sample Diskette Catalog & Directory

```
Catalog Printout:
   DISK VOLUME 176
    A 002 SIGN.ON
    A 002 SAMPLE.APPLESOFT
   *A 002 LOCKED.APPLESOFT
    B 020 SAMPLE.BINARY
    I 002 SAMPLE.INTEGER
   *I 002 LOCKED.INTEGER
    T 069 SAMPLE.TEXT
   *T 072 LOCKED.TEXT
.....................................................
Directory Dump:
 Slot:6 Drive:1
 Track: $11 (17)  Sector: $F (15)
     00    00 11 0E 00 00 00 00 00 00 00 00 12
     0C    0F 02 D3 C9 C7 CE AE CF CE A0 A0 A0
     18    A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
     24    A0 A0 A0 A0 A0 A0 A0 A0 02 00 13 0F
     30    02 D3 C1 CD D0 CC C5 AE C1 D0 D0 CC   SAMPLE.APPL
     3C    C5 D3 CF C6 D4 A0 A0 A0 A0 A0 A0 A0   ESOFT
     48    A0 A0 A0 A0 A0 A0 A0 02 00 14 0F 82
     54    CC CF C3 CB C5 C4 AE C1 D0 D0 CC C5   LOCKED.APPLE
     60    D3 CF C6 D4 A0 A0 A0 A0 A0 A0 A0 A0   SOFT
     6C    A0 A0 A0 A0 A0 A0 02 00 15 0F 04 D3            S
     78    C1 CD D0 CC C5 AE C2 C9 CE C1 D2 D9   AMPLE.BINARY
     84    A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
     90    A0 A0 A0 A0 A0 14 00 16 0F 01 D3 C1            SA
     9C    CD D0 CC C5 AE C9 CE D4 C5 C7 C5 D2   MPLE.INTEGER
     AB    A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
     B4    A0 A0 A0 A0 02 00 17 0F 81 CC CF C3            LOC
     C0    CB C5 C4 AE C9 CE D4 C5 C7 C5 D2 A0   KED.INTEGER
     CC    A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
     D8    A0 A0 A0 02 00 18 0F 00 D3 C1 CD D0            SAMP
     E4    CC C5 AE D4 C5 D8 D4 A0 A0 A0 A0 A0    LE.TEXT
     F0    A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
     FC    A0 A0 45 00                                     E
```

17.3.4 Catalog File Descriptions

There are catalog descriptions of seven files in a fully occupied catalog sector. This is where DOS learns what is important to it about any particular user file on the diskette. Figure 17.3E shows the internal structure of any particular catalog file description.

Notice that even at this level, DOS has not yet found the track(s) and sector(s) on which a particular named file is located, but the end is in sight. It finds out such basic information about the stored file as its name, its file type, its length, and whether or not it is locked, plus the track and sector of a track/sector list. This track/sector list is what actually contains the physical track/sector locations of the file itself.

Figure 17.3E
Structure and Layout of a Catalog File Description

BYTE	DESCRIPTION	DOS
Start+$00	Track of 1st track/sector list sector If this byte contains $00, the entry is assumed never to have been used and hence available for use. (NOTE: This means that track 0 is never available for user files.) If this byte contains $FF this is a deleted file and the original track number is copied to byte Start+$20	$B4C6
Start+$01	Sector of 1st 'track/sector list' sector	$B4C7
Start+$02	File type and flags TYPES: $00 - Text file $01 - Integer BASIC file $02 - Applesoft BASIC file $04 - Binary file $08 - S type (special) file $10 - Relocatable object module file $20 - A type file $40 - B type file FLAG: $80 - LOCKED file. (Add $80 to file type to lock file.)	$B4C8
Start+$03 through Start+20	File name (Maximum length = $1D or 30 characters)	$B4C9
Start+$21 through Start+22	Length of file in sectors (HI/LO format) (NOTE: Catalog command displays only low byte)	$B4E7

NOTE: DOS column indicates location in catalog/directory sector buffer assuming DOS starts at $9600 (48K Apple).

17.3.5 The Track/Sector List

We are finally to the point where DOS finds the track(s) and sector(s) at which a program or data file is physically stored. The file description points to the track/sector list that contains the information in the format shown in figure 17.3F.

Figure 17.3F
Structure and Layout of a Track/Sector List

BYTE	DESCRIPTION
$00	Not Used
$01	0 if no more track/sector lists are needed. Track number of next track/sector list if more list(s) needed.
$02	Not used if no more track/sector lists are needed Sector number of next track/sector list if more list(s) needed.
$03-$04	Not used
$05-$06	Sector offset in file (usually zero) of first sector described by this list. (Needed for random file if 1st sector not allocated.)
$07-$0B	Not used
$0C-$0D	Track and sector of 1st data sector (unless changed by offset)
$0E-$0F	Track and sector of 2nd data sector (or zeros = ignore or end)
$10-$11	Track and sector of 3rd data sector (or zeros = ignore or end)
... ...	... ...
... ..	... ...
$FC-$FD	Track and sector of 121st data sector (or zeros = ignore or end)
$FE-$FF	Track and sector of 122nd data sector (or zeros = ignore or end)

Notice that a single track/sector list has space in it for up to 122 track/sector pairs, each of which identifies a single sector of the file which is to be retrieved. Thus, only a single track/sector list is needed for files up to 122 sectors in length. However provision is made for chaining to additional track/sector lists for very long files.

All files (except perhaps a random TEXT file) are considered to be continuous streams of data, even though they must be broken up into 256-byte chunks to fit into diskette sectors. The use of the track/sector list allows an arbitrary number of these chunks to be strung together without ever bothering the programmer with such problems as:

1. keeping track of the boundaries between them,
2. keeping track of how many of them are needed,
3. keeping track of where they are, and
4. keeping track of the order in which they should be used.

Notice that with the use of the track/sector list chunks of information which contain adjacent information content need not be placed in physically adjacent sectors on the diskette. You don't need a single continuous chunk of disk space as large as the file you are trying to save. You need only as much space as the file will occupy, even if it is scattered here and there in one or two-sector chunks.

This is no mere theoretical advantage. Repeated saving and deleting of files can fragment the available space into a large number of small chunks. But this kind of information organization permits DOS to a system of memory allocation which uses the bit maps of free sectors (section

17.3.2) to find any sectors which are free, no matter where they are. Then the track/sector list allows every last isolated chunk to be used to fulfill the storage requirements of files, whether they be large or small.

It is worthwile to mention, however, that a file which is chopped up into many isolated pieces on different tracks will require significant head movement to seek out all its required sectors of information and thus will retrieve information more slowly than a file which is in contiguous locations. This reinforces our earlier rule-of-thumb that if you want faster-than-average retrieval of a particular file make it the first (or one of the first) files saved on a newly initialized diskette.

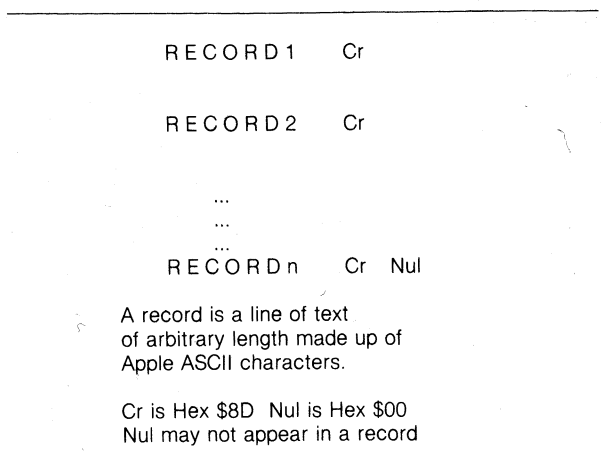
If a file is a sequential file the first appearance of a $00 track/sector signals the end of the file. However, random files can have areas within their logical structure which have never been used and which therefore have never had sectors allocated in their track/sector list. In such cases a $00 indicates a vacant area to be ignored, not the end of the file.

17.3.6 Text Files

A text file is nothing more than an arbitrary string of characters interspersed with occasional carriage returns to specify the end of a line.

A file of the TEXT data type consists of one or more records made up of ASCII characters separated from one another by ASCII Carriage Returns (Hex Code $8D and terminated with an ASCII Nul character (Hex Code $00). See figure 17.3H.

Figure 17.3H — Text File Structure



In sequential files, DOS detects the end of a text file either by finding a Nul character or by finding no more file sectors assigned in the file's track-sector list.

Random-access files set up a file structure but may fail to put anything in it; whatever garbage bits may exist in those locations might include Hex $00 combinations. Therefore DOS must depend on the structure assigned at the time of creation of the random file to find the end of a file, or the lack of any more sectors in the track-sector list. *NOTE:* The structure set up when a random access file is defined establishes fixed maximum lengths for each record.

17.3.7 Binary Files

Binary files save machine-language programs, binary data (which might be automatically gathered from sensors and generated by analog-to-digital converts), etc. Such material may be of arbitrary length and may include in its body any possible binary combination of bits.

A text file depends upon information in the form of carriage returns [binary 10001101] and nulls [binary 00000000] inside the body of the file to specify its division into records and its end. Since these bit combinations can appear in the body of the binary program or data that a binary file must save, a binary file must obviously have some different concept of organization. Form follows function, so we must look at how a binary file is used to see how it must be organized.

If we are to get binary information from inside a computer and store it on a file we must know two things: where to get the information and how much to get. An easy way to specify this is with a starting point and a length. Thus, a typical DOS command for saving a block of information from inside the computer as a binary file is

BSAVE FISHBAIT, A$300, L$B0.

(The B in front of SAVE indicates Binary. The A (for Address or 'At') indicates the starting point and the L (for 'Length') indicates length. The numbers may be in hexadecimal form with a dollar sign as shown or in decimal form without a dollar sign.)

Since DOS needs to know the actual length of the file to manipulate it properly, it can pick up the length at this time and store it as part of a header or prefix to the file — then it won't have to specify the end of the file by using a special combination of bits within the record end-of-file.

When we have a binary file stored on a diskette and we want to put that information back into a computer, the same things must be specified where it is to go and how much to put into the

machine. A command to put the same information back into the machine might be

BLOAD FISHBAIT, A$300, L$B0.

This is a perfectly legitimate DOS command. However, most of the time you will want to put back the same amount of information that you originally saved. We have stored that information in the file's prefix, so we can drop the L$B0. And most of the time we will want to put the information right back into the same location and context with other information in the computer, as before.

So why not store the BSAVE 'A' information (the $300 that went to the DOS at the time we stored the program) as another prefix to the binary data in the file? Then the user of DOS won't have to keep separate track of, and enter, this information. This reduces normal loading to

BLOAD FISHBAIT

using information stored in the prefix of the type Binary file to specify the load location and length unless explicitly overridden by the programmer.

Incidentally, it is worth mentioning that DOS does keep a record of the most recently BSAVEd or BLOADed start and length values:

Start At ('A' value) is stored in $AA72,73
[PRINT PEEK(-21902)+256*PEEK (-21901)]
Length ('L' value) is stored in $AA60,61
[PRINT PEEK(-21920)+256*PEEK(-21919)]

The concepts we have described lead to the design of binary files shown in figure 17.3I:

Figure 17.3I — File Ogranization of Binary Files

Low Byte 'A' Address	High Byte 'A' Address	Low Byte 'L' Parameter	High Byte 'L' Parameter	Arbitrary number of bytes of binary information (Length agrees with 'L'

17.3.8 Basic Program Files (Applesoft and Integer)

The concept of organization of BASIC program files is the same as that for binary files, but with a slight simplification.

Since these files are always used by one or the other of the BASIC interpreters, the interpreter always knows the start-of-program and end-of program locations so the user never needs to mention them; SAVE FISHBAIT suffices automatically. Going back from the diskette to the computer, the BASIC interpreter always knows where to start a program. In fact, due to a change in LOMEM:, it may be necessary to load the program in a different location from which it was stored. Thus it is neither necessary nor desirable to save the 'Start At' or 'A' address.

DOS, of course, still will find it convenient to have the length of the file stored and available with the program, for its internal processing and to tell the BASIC interpreter where the end of the program will be.

Figure 17.3J — BASIC Program File Organization (Applesoft of Integer)

Low Byte 'L' Parameter	High Byte 'L' Parameter	Arbitrary number of Bytes of binary information (Length agrees with 'L'

NOTE: Distinction between Integer & Applesoft files is not made explicitly in the file. It is saved on the disk in the directory of the file.

17.3.9 Recap of the DOS Method of Finding Information on the Diskette

1. Start at VTOC. It describes the way the diskette is structured, not just for standard Apple diskettes, but for many possible variants as well. This is not just a static structure; it changes as files are added or deleted by updating a bit map for each sector of the disk showing which sectors are occupied and which are not.

2. Bytes 1 and 2 of VTOC point to the CATALOG.

3. The catalog may occupy several sectors. Therefore bytes 1 and 2 of each catalog sector point to the next sector in the catalog (if any).

4. Individual entries in the catalog identify the name of each file, its type (text, binary, Applesoft, etc.), its length (in sectors) and whether or not it is locked.

5. Each entry in the catalog also contains a pointer in its bytes 0 and 1 to a track/sector list.

6. The track/sector list contains an ordered list of up to 122 track/sector pairs. The first points to the first sector which contains recorded information (program or data) from the specified file. The next specifies the sector for the next chunk of information, and so on.

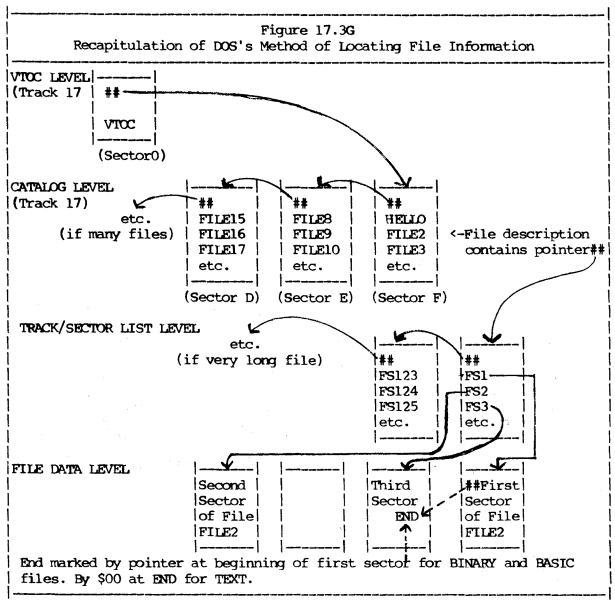
7. If a file is more than 122 sectors long, its track/sector list overflows one sector, but a pointer on the first t/s list points to a new sector containing a continuation of the list. (This happens as many times as necessary.)

8. DOS now knows where to start picking up a

file, but it doesn't know where the file ends unless it ends at the end of a sector (or unless it is a random access file that has an explicit description of its structure.)

9. The file itself specifies its own end except in the special cases above. If the file is a TEXT file the ASCII character 'Nul' ($00) specifies the end of file. BINARY and BASIC files (both Applesoft and Integer) specify their length explicitly by a two-byte length field in the header or preamble of their files.

A schematic diagram of this process is shown in figure 17.3G:



17.3.10 Deleting and Resurrecting Files

Figure 17.3G can be very useful in understanding aspects of the operation of the DOS that otherwise would be opaque. One of these aspects is the 'black magic' resurrection of deleted files.

If you were looking very closely, you may have noted in the description of the catalog that when a file is deleted its catalog entry remains physically in the catalog, but is marked in a special way to show that it has been deleted. It is also true that its track-sector list is used to re-mark the bit maps of space used and available. The re-marking indicates that the sectors identified in the track/sector list as previously used for the deleted file are now once again available for assignment. The actual data bits in the file sectors used to store the data are NOT overtly erased. That would be an unnecessary waste of time and effort.

The pointer chain shown in figure 17.3G that shows the path DOS follows to find the data, is broken. The sectors are 'un-linked' from the system that makes them exist as parts of working files. The file is dead. The sectors previously identified as being used as part of the file are now identified as part of the pool of unused sectors in the VTOC. (To be more precise, in VTOC's bit map of used and available sectors in each track.)

The individual bit patterns in the sectors at the file data level may still be there if they have not yet been allocated to some new use. If you are lucky you may even be able to find the un-linked track/sector list. To do this kind of work it is very valuable to have a Disk ZAP program. This is a program that makes it convenient for you to examine the contents of individual sectors on the diskette and change them.

With your knowledge and with a ZAP program (or even without it) you can resurrect a dead file if it is important enough to warrant the considerable amount of effort involved. Basically all you have to do is re-establish the broken linkages. If you really want to learn the DOS well, try creating a file named Lazarus with a distinctive, easily-recognized pattern of information in it. Delete it. Then find out how much you really know!

17.3.11 Disk Space Allocation to Files and Simple Rules of Thumb for Improving Disk Response

Given a freshly initialized diskette, the first sectors to be allocated to user files will be on track 18. This is immediately adjacent to the catalog track (17) so there is minimum head travel and hence minimum delay in seeking out the information.

After the space in track 18 is used, the system allocates space in track 19, track 20, and so on up to track 34. Then it goes back adjacent to the catalog track at track 16 and works its way backwards 16,15,14, etc., until it runs out space. (Last track is 4 if DOS is present; 1 if DOS has been removed.)

While it is true that track 16, a particularly favorable track, is allocated after track 34, a particularly unfavorable track, it is obvious that the average distance from the catalog track for sectors allocated quite early in the allocation history of a diskette is less than the average distance of those allocated later.

However, there is more of an advantage here for the files that are early onto the diskette. They are less likely to be fragmented into several pieces that require several head movements and hence additional mechanical movement delay time.

Before we describe how this happens let's talk about the dynamics of disk utilization. Many diskettes, particularly those used for program development, have an active history. You write several files, or the same file in several different versions, during the time you are developing a program. Some are deleted. This leaves gaps in the allocated areas.

Later, when the allocation process gets back to a deleted area, it may have small file that slips nicely into the available gap, perhaps creating a new smaller gap. Or the file may be too large to fit into the gap. DOS can cope with this, but it has to split the file into two or more pieces. These pieces are logically linked together by the track/sector list, but they may not be on the same track. Conceivably they can be on tracks far removed from one another.

The more deleting and saving you do, the more the fragmenting process continues. The to-and-fro motion of the read head becomes increasingly frenetic. Disk operations get slower and slower. Consequently, my program ran faster when I put it on a brand new disk!

Some good rules of thumb:

1. To get faster disk response with a minimum of technical effort put a file that you particularly want to speed up as the first file on a brand new disk.

2. Put the files whose disk response time is least important onto the disk last.

3. If you have been doing a lot of program development work on a particular disk, it is a good idea to periodically copy the files off of it onto another disk and reinitialize the disk. You may wish to think of this technically as a form of garbage collection.

17.4
How Diskette User Space is Allocated to User Files (PRograms, Text and Data)

If you are even slightly conscientious about wanting to save wasted time. a simple rule-of-thumb will allow you to insure better than average seek-find times:

Given a freshly initialized disk. the first files to be stored tend to be allocated on track 18, then 19 and so on up to 34, then back down to 15, 14 and eventually down to to DOS. This is not necessarily a smooth fill-in process. Some files will be too big to fit the available space remaining on a given track and there will be gaps. There will also be later fill-ins of gaps. The process can get messy. However you can be reasonably sure that the first few files to be stored will be stored in locations which are particularly favorable in terms of average access time

Chapter XVIII
The Specialized Input-Output Memory (Some of It Behaves Very Strangely and Some Isn't There At All) Memory Pages 192-207 ($C000-$CFFF)

18.1
Introduction to the 16 Pages of Specialized Input-Output Addresses

The 16 pages of memory that have addresses of the form $Cxxx (where xxx can be any three hexadecimal digits) are reserved for functions associated with either built-in hardware I-0 activities or with I-0 activities associated with the Apple slots.

They are not the only pages that deal with input-output activities. In Chapter 12 we dealt with the Input Buffer Page (Page $02xx); in Chapter 14 we dealt with the text and low-resolution graphics video display pages ($04xx-$07xx and $08xx-$11xx); and in Chapter 16 we dealt with the high-resolution Graphics video display pages ($2xxx-$3xxx and $4xxx-$5xxx).

The $Cxxx locations are different. Some of the locations do not exist as separate entities from others. Indeed many of the locations on this page are inactive or unimplemented phantoms.

Some of the locations just don't act like ordinary memory locations. They don't hold the right amount of information or change their values when accessed or change their values spontaneously as a result of external occurances.

This area is unique among memory areas in its degree of specialization. Parts of it may be implemented with RAM memory, other parts with ROM, and yet other parts will be implemented with logical elements such as flip flops. These elements are quite different from either the RAM or ROM memory elements used in conventional memory locations.

Even in those areas where implementation is by conventional ROM chips, the presence or absence of the chips depends on auxilliary equipment (cards plugged into the Apple slots).

Certain blocks of memory in this address range are allocated to specific slots and the memory implemented for these addresses will be located on peripheral cards plugged into those slots. Hence, this memory will remain inactive until the specific slot with which they are associated is activated.

Other blocks in a different part of this region may exist in multiple versions that share the same addresses but contain different information and perform different functions. These also are located physically off the main computer on peripheral cards.

Bits in some memory locations can be given new values in ways quite different from the bits in conventional memory locations. For example, they can be set by external conditions such as the position of a game paddle, by the striking of a key on the keyboard, or by access to different memory location.

This contrasts sharply with the situation for locations we have covered in earlier chapters. They have been conventional RAM locations assigned special additional duties by the hardware and by the firmware of the Apple system.

18.2
The Strange Page: Built-in I-O Locations ($C000=>$C07F) and Slot (Peripheral Card) I-O Space ($C080=>$C0FF)

This page is strange because special hardware intercepts these special addresses, partially decodes them, and handles them in special ways.

We will go through this area vary carefully, first identifying the anomalies you can expect to observe. In section 18.3 we go on to discuss the specific capabilities associated with each address (or group of addresses).

Even readers with little if any, specific experience with hardware analyses may find section 18.3.2 helpful in understanding the underlying order below the apparent chaos of different characteristics in this area. Those who are adept in understanding hardware analyses may want to skim the remainder of 18.2

18.2.1 The First Half ($C000=>$C07F) of the Strange Page in Context with the Rest of the Apple I-O System

This area contains the interfaces for keyboard input, the game controller inputs, the pushbutton inputs, and the cassette input. It contains the corresponding direct output capabilities: the speaker, cassette and annunciator outputs.

All the input-output capabilities associated with features standard to all Apple II systems fun-

nel through these few memory locations associated with this half-page (128 addresses): $C000=> $C07F (decimal 49152=>49279 or −16384=> −16257).

Input-output capabilities that use ancillary hardware plugged into the Apple slots, (e.g. diskettes, printers, and communications terminals) find their route for getting information into or out of the Apple via areas of the $Cxxx address space other than this very special half-page.

18.2.2 Anomalous Characteristics of Locations in the First Half of the Strange Page ($C000=>C07F)

These addressable locations are unlike any other group of memory locations in the Apple.

The key to the strange behavior of addresses in this area is that they are not really associated with conventional memory hardware and addressing techniques. There are three major areas of anomalies: one in the amount of data that can be stored in an addressable location; the second in possible non-unique addressing of data; and the third in what happens when you do access the locations. These are the key anomalies:

1. Not all of addressable locations contain a full byte (8 bits) of information; many can store only a single bit of information.

2. Not all of them are uniquely distinguishable from one another; as many as sixteen different addresses can all be used interchangeably to access some physical location within this address zone.

3. Some can be written to but not read from; some can be written to by PEEKing as well as by POKEing; while others should be written to only by PEEKing, not by POKEing. (Comparable anomalous behavior required for machine-language access as well.)

All in all, this is a very confusing area until you know something about what underlies this strange behavior.

18.2.3 Bit Versus Byte Anomaly in the Strange Page

First let's consider the bit-versus-byte data anomaly.

In the $C000=>$C07F area, addresses do not point to a conventional 8-bit memory cell. Most addresses point to specialized circuitry, such as flip-flops capable of storing only a single bit of information. Whenever only a single off-on state (a single bit) is associated with a given address, the circuitry is arranged so that it appears to be located in the MSB (Most Significant Bit) or sign bit position of the byte that would normally be expected in specified memory location. The remaining bits become inconsistent garbage.

Other addresses point to a byte of storage which appears to suffer from schizophrenia. Its MSB bit leads a semi-independent existence, a quite different life from that of its remaining seven bits, which may have a consistent and useful meaning in terms of the input-output system. (Discussion of these anomalous bytes is deferred to section 18.2.4)

Flip-flops are easily arranged to control other circuits internal to the computer, or they can be arranged as outputs to control circuits outside the computer. The single bit they are able to store is quite adequate for many input or output activites (e.g. for a pushbutton input, a speaker toggle output, or a soft switch to control some aspect of a video display).

Single bit inputs or outputs are always in the MSB (Most Significant Bit or sign bit) position where it is most easily manipulated and tested by either machine-language or BASIC commands.

In BASIC, if the PEEK of a particular address is greater then 127, the MSB or flag bit is on ('1'); if it is 127 or less the bit is off ('0').

In machine language an equivalent test can be made by testing the sign (N) bit of the status register for the '0' or '1' condition. (This should be done after referring to the address to be tested with the BIT command.)

18.2.4 Full-byte Input Locations in the First Half of the Strange Page Work Differently from Conventional Full-byte Memory Locations

In the $C000=> $C0FF area, even locations that can store a full 8-bit byte behave differently from normal memory locations. The bits in bytes of conventional memory can be set only by addressing them and writing information to that address. The bits in this area of memory can be set by conditions from the outside world, e.g. $C000's bits can be changed by a key press on the Apple keyboard. $C064 through $C067's bits can be changed by changing the position of game controller paddles 0 through 3 respectively.

The contents of these locations also can be affected by accesses to other memory locations than themselves, specifically to strobe locations associated with their input functions.

Moreover, the method of setting the contents of the MSB is separated from, though related to,

the method of setting the other seven bits. For example, for the keyboard data input byte ($C000) the MSB can be reset by accessing the Clear Keyboard Strobe ($C010). For the game controller inputs [$C065, $C066 and $C067], the MSB's can similarly be reset (and their timing loops restarted) by accessing the game controller strobe at $C070.

In effect the MSB and the other seven bits of these full-byte inputs constitute two separate, but closely related inputs.

18.2.5 The Incompletely Decoded Address Anomaly in the First Half of the Strange Page

Next let's consider the addressing anomalies in the $C000 => $C07F address range. As we shall later see addresses in this range are directed to special I-O selector circuitry that partially decodes them. In this range only addresses in the $C050 => $C05F sub-range go on to a second level of decoding, which decodes them completely.

This means that part of the address decoded is used in determining the location where the computer puts or takes information. The part of the address that is not decoded has no effect whatsoever.

Thus addresses in a partially decoded addressing area, which differ only in bits not decoded, are totally indistinguishable.

Addresses $C00x, $C01x, $C02x, $C03x, $C04x, and $C07x all have the last four bits (the last hexadecimal digit) undecoded. For example, in the case of the keyboard data input address $C000, the last hexadecimal digit of the address is not decoded. Thus $C00F decodes to the same byte of information as $C000. So does $C00x where x is any hexadecimal digit. They are totally interchangeable.

18.2.6 Data-Change on Read-Access Anomalies in the First Half of the Strange Page

Let's consider data-change-on-access anomalies in the $C000 =>$C07F address range.

Whenever you access a conventional RAM memory cell you destroy the information in it. Under normal circumstances you don't even know this is happening. The circuits associated with the memory immediately write back the original information and return the cell to its original state or its desired new state. However, the circuitry of a flip-flop is not designed to operate in this fashion.

Soft-switches and toggles in the $C000 = > $C07F address space are flip-flops with different methods of addressing inputs. Strobes may be considered as output-generating or controlling flip-flops.

A soft switch flip-flop has a pair of addresses from which it can be accessed. If you access one of them, the bit in at the address accessed goes on (i.e., becomes a '1' while the bit at the other address becomes a '0'). Whatever was there before is destroyed in the process and no means is provided to determine what it was.

A toggle flip-flop differs from a soft-switch in that it has but a single address. If you access it you change its state. That is, if it was in an 'off' or '0' state, it switches to an 'on' or '1' state. Conversely, if it was in an 'on' or '1' state and you access it, it switches to the 'off' or '0' state.

When you read or PEEK a particular address you access it once. When you write or POKE, you access it twice. The two accesses happen almost on top of one another. For example, if you POKE the utility strobe you get two pulses 1 microsecond long separated by a 1/40th microsecond.

It seems natural to POKE rather then PEEK when you want to change a value. It is, except in anomalous situations such as here.

However, since we get two pulses instead of one when we POKE or otherwise write, and since we usually desire only a single pulse, we must face up to awkward questions about the wisdom of POKEing flip-flops (toggles, soft switches or strobes):

1. Will two pulses have the same desired effect as one?

2. Does the second of the two accesses follow so close behind the first that it always seems to be a single double-duration pulse? If so, does a double-duration pulse provide an acceptable solution to the problem?

(The circuits in the Apple are fast enough so that a typical separation of about 24 nanoseconds between pulses is adequate to achieve functional separation into separate pulses under routine conditions. Thus the second question is moot. POKEing can still cause problems sometimes.)

3. Depending upon the circuits driven and the recovery time between pulses, can you be sure the two accesses associated with a write or POKE are far enough apart that they always can be treated as two separate accesses? If the separation is occasionally insufficient to retain the dinstinction, erratic operation could ensue if you used POKEs for deliberate double-pulsing.

(For reliable and consistent operation you don't want to have to depend upon a close call. If you

want two pulses you would have more margin of safety using two PEEKs than a single POKE.)

Net result: The natural or intuitive solution of using machine-language write operations or POKEs to change the state of flip-flops is sometimes acceptable, but not always.

A simple, safe rule of thumb is 'Never write or POKE to a flip-flop or a strobe; a PEEK or any machine-language access to the location will do the job.' This rule is easy to remember and follow, but somewhat more stringent than necessary.

More complicated rules that allow additional freedom in the use of write operations and POKEs are

1. Never write (POKE in BASIC) to a Toggle.
2. You can safely write to soft-switch (There, two pulses are as good as one.)
3. You can safely write to the Keyboard and Game controller strobes (a double pulse will be generated but it will have no adverse effect).
4. Don't write to the Utility Strobe unless you have positive proof that the double pulse won't cause trouble in the particular application you have chosen.

Properly implemented, these rules are as safe as the simpler, more stringent one presented first.

18.2.7 The Second Half ($C080 => $C0FF) of the Strange Page

The 128 locations in the second half of the strange page are quite conventional. They are, however, assigned to very specialized use as eight blocks of 16 memory locations. Each block of 16 locations is assigned to input-output uses in conjunction with one of the eight Apple slots.

Each block consists of addresses of the form $C0Sx, where S = 8 + the slot number with which the block is associated. X can have 16 values (0 through F) giving 16 bytes in the block. Thus $C080 = > $C08F are assigned to slot #0; $C090 = > $C09F are assigned to slot #1;... and so on, through $C0F0=>$C0FF being assigned to slot #7. Figure 18.3A portrays the assignment schematically and figure 18.B expands its base address/indexing implications.

18.3
The Strange Page In Depth

18.3.1 Tabular Summary/Overview

The $C0xx page of memory contains two remarkably dissimilar half pages. Figure 18.3A presents a graphical summary of the addressing pattern for the first half-page $C000= > $C07F.

Figure 18.3A

Memory Half-Page $C000=>$C07F: Addresses of Apple][System Built-in I-O Functions

	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F
$C000	<-Keyboard Data Input->															
$C010	<-Clear Keyboard Strobe->															
$C020	<-Cassette Output Toggle->															
$C030	<-Speaker Toggle->															
$C040	<-Utility Strobe->															
$C050	graph	text	nomix	mix	pg 1	pg 2	lo-res	hi-res	an0		an1		an2		an3	
$C060	cin	pb 1	pb 2	pb 3	gc 0	gc 1	gc 2	gc 3	<-repeat $C060-$C067->							
$C070	<-Game Controller Strobe->															

Abbreviations used:
graph => set graphics mode
text => set text mode
nomix => set all text or all graphics
mix => set for mix of text and graphics
pg 1 => display page 1 (primary pg)
pg 2 => display page 2 (secondary pg)
lo-res=> display low resolution graphics
hi-res=> display high resolution graphics
an * => use annunciator output *
pb * => use pushbutton input *
gc * => use game controller input *
cin => cassette input

Notice that 24 different Input-Output-related functions share 128 memory addresses. Six of the 24 occupy 16 addresss each; Twelve of the 24 occupy 2 addresss each; and Eight of the 24 occupy a single address each. This strange behaviour is due to the incomplete address decoding anomaly.

Figure 18.3B presents the corresponding breakdown for the second half-page $C080 = > $C0FF.

Figure 18.3B
Memory Half-Page $C080-$C0FF: I-O Space for Apple 'Slots'

	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F
$C080	<----- Input-Output (Peripheral Card) Space for Slot #0 ----->															
$C090	<----- Input-Output (Peripheral Card) Space for Slot #1 ----->															
$C0A0	<----- Input-Output (Peripheral Card) Space for Slot #2 ----->															
$C0B0	<----- Input-Output (Peripheral Card) Space for Slot #3 ----->															
$C0C0	<----- Input-Output (Peripheral Card) Space for Slot #4----->															
$C0D0	<----- Input-Output (Peripheral Card) Space for Slot #5 ----->															
$C0E0	<----- Input-Output (Peripheral Card) Space for Slot #6 ----->															
$C0F0	<----- Input-Output (Peripheral Card) Space for Slot #7 ----->															

18.3.2 Hardware Perspective of the Strange Page

Whenever the strange page is addressed, a 74LS138 located at position H12 on the Apple mother board detects that fact and enables another 74LS138, known as the I/O Selector. This chip is located at position F13 on the Apple mother board.

The I/O Selector ignores the second half-page ($C080 = > $C0FF) of the strange page and partially decodes the first half-page ($C000 = > $C07F) in eight areas of 16 bytes each:

$C00x, $C01x, ..., $C07x

The I/O Selector has eight output lines numbered 0 through 7. Each output line of the 74LS138 becomes active when the 16-byte range having the same digit in its third hexadecimal digit position is being referenced. For example, an address of the form $C05x will cause I/O Selector output line 5 to become active.

Thus the 74LS138 I/O Selector distinguishes between the addresses in figure 18.3A (the first half-page) and figure 18.3B (the second half-page). It ignores any address in figure 18.3B, but it partially processes any address in figure 18.3A.

The I/O Selector does a partial decoding of addresses in figure 18.3A. This partial decoding breaks the overall block of 128 addresses in figure 18.3A into eight modules, each of which is a horizontal row of 16 addresses and activates a different output line for each row.

The '0' line from the I/O Selector is activated when an address in the $C000 Keyboard Data Input row of figure 18.3A is specified. When activated, this line opens a gate that allows data to flow from the keyboard connector into the RAM data multiplexer. See section 18.3.3 for additional interpretation of what this means functionally. No additional decoding occurs so it is impossible to distinguish between addresses on this row of figure 18.3A.

The '1' line from the I/O Selector is activated when an address in the $C010 Clear Keyboard Strobe row of figure 18.3A is specified. When activated, this resets the 74LS74 flip-flop at B10, which is the keyboard (input) flag (MSB or flag bit of the keyboard input byte). See section 18.3.3 for additional interpretation of what this means functionally. No additional decoding ever occurs so it is impossible to distinguish between addresses on this row of figure 18.3A.

The '2' line from the I/O Selector is activated when an address in the $C020 Cassette Output Toggle row of figure 18.3A is specified. When activated, it toggles a flip-flop, which is one half of the 74LS74 at Apple mother board location K13. The output of this flip-flop is connected *via* a resistor network to the tip of the cassette output jack. See section 18.3.4 for additional interpretation. No further decoding occurs so no distinction is ever made between the addresses on this row of figure 18.3A.

The '3' line from the I/O Selector is activated when an address in the $C030 Speaker Toggle row of figure 18.3A is specified. When activated, it toggles a flip-flop, which is the other half of the 74LS74 at Apple mother board location K13. The output of this flip-flop is connected through a capacitor and Darlington amplifier circuit to the Apple's speaker connection at the right edge of the mother board under the keyboard. See section 18.3.4 for additional interpretation. No further decoding occurs so no distinction is made between the addresses on this row of figure 18.3a.

The '4' line from the I/O Selector is activated when an address in the $C040 Utility Strobe row of figure 18.3A is specified. It is directly connected to pin 5 of the Game I/O connection. See section 18.3.5 for additional interpretive information. No further decoding occurs so no distinction is made between the addresses on this row of figure 18.3A.

The '5' line from the I/O Selector is activated when an address in the $C050 row of figure 18.3A is specified. It is used to enable the 74LS259 integrated circuit at Apple mother board location F14. This IC contains the soft switches for the video display and the Game I/O connector annunciator outputs. Further decoding occurs using the last hexadecimal digit of the address. Bits (address lines) 3, 2, and 1 of this hex digit specify which soft-switch to access and address line 0 to specify the setting of the selected switch. See sections 18.3.5 and 18.3.6 for functional interpretations of what this means.

The '6' line from the I/O Selector is activated when an address in the $C060 row of figure 18.3A

is specified. It is used to enable a 74LS251 eight-bit multiplexer at Apple mother board location H14. This multiplexer, when enabled, connects one of its eight input lines to the MSB (Most Significant Bit) of the three-state system bus. Bits 2, 1, and 0 of the last hex digit of the address control the eight input lines the multiplexer uses. Bit 3 is unused so that the block of eight addresses that has this bit in the '1' condition is indistinguishable from the block of eight that has this bit in '0' condition.

Four of the multiplexer's inputs come from a 553 quad timer at location H13. The inputs to this timer are the game controller (paddle) pins on the Game I/O connector. How these are used to detect and react to the paddle position is covered in detail in section 18.3.7.

Three of the remaining inputs come from the single-bit (pushbutton) inputs on the Game I/O connector. The final multiplexer input comes from a 741 operational amplifier at Apple mother board location K13. The input to this operational amplifier comes from the cassette input jack.

The '7' line from the I/O Selector is activated whenever an address from the $C070 Game Controller Strobe row of figure 18.3A is specified. No further decoding occurs so the computer is unable to distinguish between different addresses in this row. This line is used to reset all four timers in the 553 quad timer at location H13, which are used in conjunction with the game controllers/paddles.

18.3.3 Keyboard Data Input ($C00x) and the Clear-Keyboard Strobe ($C01x)

The primary data input of the Apple II System is the keyboard input. It uses $C000 as the address of a one-byte hardware interface register.

It is not strictly true that address $C000 is the address of the keyboard input register. The last hexadecimal digit of the address is not decoded. If any address $C01x in the range $C000 through $C00F is specified, the results will be identical.

The seven low-order bits of the byte in $C000 represent the character ASCII code of the key that was most recently depressed, while the eighth bit is treated as a 'flag' bit.

Whenever a key on the keyboard is pressed, this 'flag' bit (in the position of $C000) is set 'on.' In addition, bits representing the ASCII code for the letter, number, or special symbol represented by the key are sent to the seven low-order bit positions of $C00x.

Thus a PEEK of $C00x (any PEEK using locations in the range PEEK(49152) to PEEK(49167)) has a value > 127 after a key is depressed.

The flag bit stays in that condition until the Clear Keyboard Strobe ($C010) is accessed. (Note: As with $C000, the last hex digit of $C01x is not decoded so any address from $C010 = > $C01F has identical effect.

Keyboard clear strobing is usually accomplished by doing a PEEK(– 16368). However, the strobing action occurs any time $C010 (decimal – 16368) is memory-accessed in any way. For example, the machine-language instruction LDA C010 would also strobe the keyboard.

When strobing occurs the 8th bit is reset, but the seven data bits are not erased or altered in any way. Use of the strobe when you access the data in $C000 makes it possible to tell whether another keystroke has occurred since the last time you processed the keyboard input. The 'standard' Apple convention is that no new input will be accepted from the keyboard until the MSB is reset by strobing.

Thus, once your program has begun processing the information received from one keypress, it should activate the clear keyboard strobe to release the keyboard and allow the keyboard to accept the next character. If you plan to go back for a second look you may defer strobing the keyboard at the cost of slowing down the input (and possibly even losing a character typed if the person at the keyboard continues to enter information while the keyboard is not ready to accept information).

The Apple II system monitor takes care of this, and many other housekeeping activities associated with routine BASIC inputs, by using the RDCHAR routine or the even higher-level GETCHAR routine.

However, there are times when it is useful for you to use the direct hardware inputs without the intermediary of systems software. When the standard input routines access $C000, if a keystroke has not yet arrived, they remain in a wait loop reaccessing $C000 over and over again until an input occurs. This means that no further computing can go on until input arrives.

However, there may be times when it is desirable to continue computing. Perhaps you may even want to continue creating and displaying new output while waiting for input. You may also find direct keyboard hardware input using $C000 and $C010 convenient if you are writing interactive games or developing programs for laboratory data reduction.

18.3.4 The Cassette Output Toggle ($C02x) and The Speaker Output Toggle ($C03x)

The cassette output toggle and speaker output toggle are single-bit outputs from toggle flip-flops. These single-bit outputs become a sequential (serial) string of bits as the output is toggled from one condition to the other as a function of time.

In one case, the output goes to the cassette tape recorder, in the other to the Apple's built-in speaker.

Audio tones in the speaker or on a cassette tape recording are obtained by toggling the output from '0' to '1' and back at an audible rate; e.g., 3000 times per second for a 3000-cycle audio tone.

The cassette output can also be used for digital storage of programs and/or data using special built-in software and commands provided in the Apple system monitor and BASIC interpreter firmware.

The addresses for these two outputs are used without decoding the last hexadecimal digit, so the least significant hex digit of the address is ignored.

Since these outputs are implemented as toggles there is no practical way of determining their current setting; their bit value changes with every memory access; e.g., every time their address is PEEKed or accessed by a machine-language instruction. User programs should never write to (e.g., POKE) these toggles.

18.3.5 Utility Strobe ($C040x)

If a program accesses the Utility Strobe ($C04x), the mere act of usage (even the act of PEEKing) will trigger actions that may be used as an Apple system output. (The last hexadecimal digit of the address is not decoded so any address in the range $C040 = > $C04F is completely indistinguishable from any other.)

If one of the Utility Strobe's addresses is used, pin 5 on the Game I/O connector will drop from +5 volts to 0 volts for a period of .98 microsecond, then rise back to +5 volts again.

(Note: You should not do a BASIC POKE or otherwise *write* to the utility strobe unless you want *two* outputs about 25 nanoseconds apart. A write operation involves two memory accesses; the first to read the contents of the location and the second to overwrite it.)

18.3.6 Video Screen Display Mode-Selection Soft-Switches ($C050 = > $C057)

We have seen these soft-switches several times before because of their usefulness in display control. The two adjacent addresses not separated by a dotted line represent the two sides of a flip-flop; one is always on (has value '1') at the same time the other is off (has value '0'). To turn one side on you just access the memory location: PEEK it, POKE it (with any value), or use that address in any machine-language instruction. Since an access to the memory location forces the flip-flop into that position, there is no direct way to determine the status of the switch other than observing its effect on the display screen.

Figure 18.3C
Video Screen Display Mode-Selection Soft-Switches

Hex	Decimal	Effect
$C050	49232 or -16304	Display Graphics Mode
$C051	49233 or -16303	Display Text Mode
$C052	49234 or -16302	Display All Text or All Graphics
$C053	49235 or -16301	Display MIXED Text & Graphics
$C054	49236 or -16300	Display Primary Page (Page 1)
$C055	49237 or -16299	Display Secondary Page (Page 2)
$C056	49238 or -16298	Display LO-RES (If graphics on)
$C057	49239 or -16297	Display HI-RES (If graphics on)

18.3.7 Annunciator Output Soft-Switches ($C058 = > $C05F)

The Apple has four relatively little-known one-bit outputs called annunciators that appear as extra pins on the game paddle connector. Each is associated with a soft-switch. An annunciator output can be used as a low-power, low-voltage control input to some other electronic device. Thus annunciator outputs can be used to control relays, triacs, etc., and through them almost any kind of external device.

In the figure 18.3D each annunciator soft-switch appears as a pair of addresses not separated by a dotted line. If you access the first address in the pair you turn the output of its corresponding annunciator off; that is, the voltage on its pin of the Game I/O connector is approximately 0 volts. If you access the second address in the pair you turn it on (the voltage on its pin of the Game I/O is approximately 5 volts).

Figure 18.3D
Annunciator Outputs

Annunciator	State	Hex Address	Decimal Addresses
0	off	$C058	49240 or -16296
	on	$C059	49241 or -16295
1	off	$C05A	49242 or -16294
	on	$C05B	49243 or -16293
2	off	$C05C	49244 or -16292
	on	$C05D	49245 or -16291
3	off	$C05E	49246 or -16290
	on	$C05F	49247 or -16289

As previously indicated, accessing a soft-switch may be done by a PEEK, a POKE (of any value), or by using a machine-language instruction that uses the relevant memory address. Since accessing forces the soft-switch to the position accessed there is no easily programmable way of determining the status of an annunciator output other than by observing its external effect; e.g., bringing the output back in by connecting it to a flag input.

18.3.8 Cassette and Pushbutton/Flag Inputs ($C060 = > $C063 or $C068 = > $C06B)

The cassette input ($C060) and the pushbutton inputs ($C061-$C063) are single-bit flag inputs. The high-order bit of the last hexadecimal digit in the address is not decoded so $C068 is indistinguishable from $C060, $C069 from $C061, etc.

These inputs have only two conditions: off and on. They are considered to be flags because they appear in the highest order (or sign) bit position of the location specified by their address. This bit location is so easy to test it is often used as a quick and easy method of flagging and testing for special conditions of data or programs.

The off condition is represented by a 0 in the highest order bit position, and the on condition by a 1. The condition is easily tested in either BASIC or machine language. Since the highest order bit position has binary value 2 ^ 7 (= 128), a PEEK of the location that shows a value > 127 indicates the flag is in the on condition, while one that shows a value < = 127 indicates the flag is in the off condition. For testing with hardware instructions just load the location into one of the microprocessor's hardware registers, thereby setting the 'N' (or negative sign) bit of the status register. (If the flag bit is on, the bit in the highest order position will cause the 'N' bit to go on when the register is loaded as an indicator that its sign is negative if the byte is treated as a signed binary number.) Thus a sign test using a BMI (Branch MInus) will cause a branch if the flag is on. A sign test using a BPL (Branch PLus) will branch if it is off.

18.3.9 Analog/Game Controller Inputs ($C064 = > $C067 or $C06C = > $C06F) And the Analog Clear/Game Controller Strobe ($C070)

Four analog inputs also appear on the Game I/O connector. They can be connected to 150K Ohm variable resistors or potentiometers to provide rotary paddle or joystick input to the Apple. For each input this is accomplished using +5 volt supply and a 100 Ohm current-limiting resistor to charge a small (0.022 microfarad) capacitor and let the charge leak off through the variable resistance. The less the resistance the more electrical charge leaks off and the less time is required to discharge the capacitor. A timing counter keeps track of the time to discharge and measures the setting of the variable resistor and hence the paddle or joystick position. Either the BASIC PDL( ) function or a machine-language program can access the timing counter and thus read the potentiometer setting.

Before a program can start to read the setting of a potentiometer, it must first reset the timing circuits. The Analog Clear/Game Controller Strobe ($C070 or decimal 49264 or decimal −16272) does this. When accessed it sets the MSB (sign or 'flag' bit) of the analog inputs and countdown begins. Within approximately three milliseconds the threshold should be reached and the MSB dropped. Discharge time will be measured by counts the counter has performed before this happens. Notice that readings that might be taken before the MSB goes back off will not accurately represent the potentiometer setting.

If no potentiometer is connected to the Game I/O connector at the analog input specified, then the values in the game controller location may never drop to zero. Potentiometer values 150K in maximum will also not leak enough charge at the high end of their resistance range to be usable except at the low end of their variable resistance ranges.

You can take advantage of the other side of this coin to use other than the Apple standard 150K variable resistors. If you want to use a smaller resistor, which lets more charge leak off the capacitor, just use additional capacitance so that more charge is stored and more charge must be leaked to drop the capacitor voltage to the counter turn-off threshold. Adjust so that the time to discharge the combined capacitors to the threshold level is the same (trial and error is satisfactory).

If a program accesses the Game Controller Strobe ($C07x), the mere act of usage (even by a PEEK) will trigger actions that may be used as an Apple System Strobe output. If the Game Controller Strobe's address is used (for example, by a PEEK), all of the flag inputs of the Game Controllers will be turned off and their timing loops restarted.

Note that the last hexadecimal digit of a $C07x address is not decoded. Thus any address in the range $C070 => $C07F will be totally indistinguishable from any other. Also note that a

double pulse initiated by a POKE (or any other write-type access) will have no noticeable or adverse effect on the strobing action.

18.3.10 'Slot' or Peripheral Card I/O Space ($C080-$C0FF)

The top half of the strange $C0xx page of memory is more conventional in organization and implementation than the bottom half, but it too is dedicated to highly specialized functions. Its functions are tied to support of activities that involve use of the eight slots located along the back of the Apple's main board with allocations of blocks of memory to individual slots, as shown in figure 18.3B. Full coverage of this area will be deferred to section 18.4.3 where it can be presented in the overall context of memory support for slots and peripheral cards, the topic of section 18.4.

18.4 Slot/Peripheral Card I/O Locations ($C100-$CFFF)

Inside the cover, along the rear of the Apple II's main circuit card, there is a row of eight printed-circuit connector board sockets or slots, numbered 0 through 7. Slot 0 is the leftmost slot (closest to the power supply), while slot 7 is the rightmost (closest to the video and cassette connectors).

These slots are provided to allow the user to plug in additional circuit boards. Originally these slots were intended to allow the user to plug in controllers or interface units to connect the Apple II to optional peripheral devices.

For example, Apple sells a communications card to interface to communications lines, a high-speed serial card to interface with serial printers and other serial-by-bit devices, and a parallel-printer card to provide parallel interface to computer line printers.

Slots are actually general-purpose bus interfaces; they are not narrowly restricted into what they can interface. For example, you can buy a Z-80 microprocessor that will allow the Apple to run programs in Z-80 machine language or to use software from Z-80-based microcomputer systems, which use the CPM operating system. Or you can buy a high-speed arithmetic processor to increase the brute computational capability of the Apple.

Slot 0 (the leftmost as you sit at the Apple II keyboard) is a special slot reserved for RAM, ROM, or interface expansion. It is the slot into which you plug such things as the Applesoft Card, the Integer BASIC Card, or the Language System Card. All other slots are identical and are provided with special control lines that provide for highly flexible interfacing.

18.4.1 Overview of Memory Assigned to Each Peripheral Slot

One particularly interesting characteristic of the Apple II system is its use of a standardized scheme for interfacing not just the hardware, but the software/firmware associated with peripherals interfaced *via* hardware, which plugs into these slots. Two-hundred-eighty (280) addresses are allocated for the exclusive use of each of these peripheral interfaces. The locations assigned for use by one slot have different addresses and are totally independent of the locations assigned to any other slot.

In addition, the peripheral slots as a group are allocated another 2K of expansion address space. Any one slot can take over and exercise control over this entire block of addresses, assigning part or all of it to RAM or ROM memory, which may be located on the plug-in card in that slot. The overall memory allocation plan for providing support to slots is summarized in figure 18.4A.

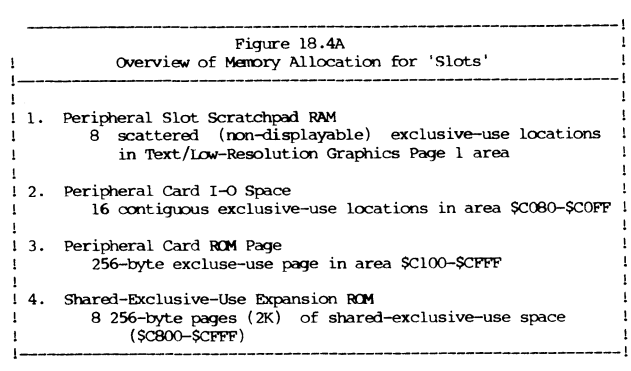


Figure 18.4A
Overview of Memory Allocation for 'Slots'

18.4.2 Peripheral Slot Scratchpad RAM

Each of the eight peripheral slots has eight locations assigned to it, one in each of the Page 1 text/low-resolution graphics macro-lines. A macro-line is a half-page (128 bytes). One-hundred-twenty (120) bytes are required for the three display lines that make up a macro-line, leaving eight bytes to be assigned. One is assigned to each of the eight slots: the first to slot 0, the second to slot 1, and so on through the eighth to

slot 7. Figure 18.4B identifies the locations assigned to each slot.

Figure 18.4B

I-O SCRATCHPAD RAM ADDRESSES (TEXT/LO-RES GRAPHICS PAGE 1)

Base Address	Slot Number 1	2	3	4	5	6	7
$0478	$0479	$047A	$047B	$047C	$047D	$047E	$047F
$04F8	$04F9	$04FA	$04FB	$04FC	$04FD	$04FE	$04FF
$0578	$0579	$057A	$057B	$057C	$057D	$057E	$057F
$05F8	$05F9	$05FA	$05FB	$05FC	$05FD	$05FE	$05FF
$0678	$0679	$067A	$067B	$067C	$067D	$067E	$067F
$06F8	$06F9	$06FA	$06FB	$06FC	$06FD	$06FE	$06FF
$0778	$0779	$077A	$077B	$077C	$077D	$077E	$077F
$07F8	$07F9	$07FA	$07FB	$07FC	$07FD	$07FE	$07FF

(Note: Similar areas are available in Page 2 of text/low-resolution graphics and in both Pages 1 and 2 of high-resolution graphics. However, since only Page 1 of the text/low-resolution graphics area (the area of the scrolling buffer) is used in almost any program, only that area is *permanently* allocated for scratchpad.

When the other screen buffer areas are used, the comparable locations in their structure become additionally available for such allocation and use in extension of this basic plan. They may, of course, be used instead in any other way preferred by the programmer.

18.4.3 Peripheral Card I/O Space

Each of the eight peripheral slots also has a block of 16 contiguous addresses assigned to it in the special I/O area, $C080 = > $C0FF, to do with as it will. Figure 18.3C showed this allocation pictorially.

The slot 0 address for any of the 16 words may be used as a base address to be indexed by the amount $S0, where S is the slot number, to point to the corresponding word in the S-th slot. This relationship is shown in figure 18.4C.

The Apple convention for making Peripheral Card PROM programs slot-independent puts the slot number in the form $CS in memory location $07F8. In machine language this can be AND'ed with $0F to get the slot number in the form $0S, then shifted four bits to the left to get the form $S0 needed for this indexing.

In BASIC, similar indexing can be done by adding the base address and modified slot number. However, decimal rather than hexadecimal addresses must be used. In BASIC you can get the slot number S by doing a

```
LET S = PEEK(2040) - 192
```

Since slot numbers are less than decimal 10, decimal and hexadecimal slot numbers are identical. The decimal equivalent of the $S0 needed for indexing is 16*S.

Figure 18.4C
I/O Location Base Address/ Indexing Pattern
for Card/Slot Portability
for Card/Slot Portability

Base Address	Slot 0	1	2	3	4	5	6	7
$C080	$C080	$C090	$C0A0	$C0B0	$C0C0	$C0D0	$C0E0	$C0F0
$C080	$C080	$C090	$C0A0	$C0B0	$C0C0	$C0D0	$C0E0	$C0F0
$C081	$C081	$C091	$C0A1	$C0B1	$C0C1	$C0D1	$C0E1	$C0F1
$C082	$C082	$C092	$C0A2	$C0B2	$C0C2	$C0D2	$C0E2	$C0F2
$C083	$C083	$C093	$C0A3	$C0B3	$C0C3	$C0D3	$C0E3	$C0F3
$C084	$C084	$C094	$C0A4	$C0B4	$C0C4	$C0D4	$C0E4	$C0F4
$C085	$C085	$C095	$C0A5	$C0B5	$C0C5	$C0D5	$C0E5	$C0F5
$C086	$C086	$C096	$C0A6	$C0B6	$C0C6	$C0D6	$C0E6	$C0F6
$C087	$C087	$C097	$C0A7	$C0B7	$C0C7	$C0D7	$C0E7	$C0F7
$C088	$C088	$C098	$C0A8	$C0B8	$C0C8	$C0D8	$C0E8	$C0F8
$C089	$C089	$C099	$C0A9	$C0B9	$C0C9	$C0D9	$C0E9	$C0F9
$C08A	$C08A	$C09A	$C0AA	$C0BA	$C0CA	$C0DA	$C0EA	$C0FA
$C08B	$C08B	$C09B	$C0AB	$C0BB	$C0CB	$C0DB	$C0EB	$C0FB
$C08C	$C08C	$C09C	$C0AC	$C0BC	$C0CC	$C0DC	$C0EC	$C0FC
$C08D	$C08D	$C09D	$C0AD	$C0BD	$C0CD	$C0DD	$C0ED	$C0FD
$C08E	$C08E	$C09E	$C0AE	$C0BE	$C0CE	$C0DE	$C0EE	$C0FE
$C08F	$C08F	$C09F	$C0AF	$C0BF	$C0C0F	$C0D0F	$C0EF	$C0FF

Associated with this block of addresses are special control features, which make these locations particularly convenient for intercommunication with the central machine.

Each peripheral card can determine if it is selected for operation, and when, by testing the condition of a special control line, the DEVICE SELECT (negated), located at pin 41 on its peripheral connector. Whenever the voltage on this pin drops to 0 volts, the address that the microprocessor is calling for is located somewhere in the 16-byte block of addresses belonging to that particular peripheral. The peripheral card can then look at the bottom four address lines to determine which of the addresses in this special 16-address block is being called for.

18.4.4 Peripheral Card ROM Page

Each peripheral slot also has reserved for its exclusive use one 256-byte page of memory. This page is normally used for ROM or PROM, which contains the driving and interfacing routines needed by the peripheral card.

The allocation of this space, which is addressable within the main system addressing scheme, permits the individual peripheral cards to contain their own driving software usable by the main system. This means that it is possible,

in many cases, for the system to avoid loading special interface programs to use individual interface cards. Those programs can be on the card itself, but accessible from the main system.

The page of memory reserved for each peripheral card has the page number $Cs (memory addresses $Cs00-$CsFF), where s is the slot number 1-7 (see figure 18.4D).

Figure 18.4D
Slot/ Peripheral Card PROM Page Assignments

Slot Number	Page	Memory Addresses
1	$C1	$C100-$C1FF
2	$C2	$C200-$C2FF
3	$C3	$C300-$C3FF
4	$C4	$C400-$C4FF
5	$C5	$C500-$C5FF
6	$C6	$C600-$C6FF
7	$C7	$C700-$C7FF

The space that would have been used to provide a page of memory for slot zero was used up giving each of the slots its 16-bytes of Peripheral Card I/O space. This means that most Apple interface cards will not work in Slot 0.

When the central microprocessor references an address within the peripheral card ROM page assigned to a particular slot, a special signal, the I/O SELECT (negated) connected to Pin 1 on the slot's plugpin connector drops from +5 volts to 0 volts. The peripheral card can then use this signal to enable their ROMs and use the lower eight address lines to determine which of the 2 ∧ 8 (= 256) locations in the page the central machine is accessing.

Apple strongly recommends the use of software conventions that make the programming of peripheral card PROMs slot-independent. The conventions include such practices as saving the values of all 6502 hardware registers on entry to a PROM subroutine, using a short standard program to determine the slot number and storing it in the form $CS in location $07F8, and use of the Base Address/Indexing technique described above. Detailed documentation is provided with Apple's blank general-purpose expansion card. If you do not use Apple cards the key information needed may be found in "I/O Programming Suggestions" found on page 81 of the *Apple II Reference Manual* you received with your computer.

18.4.5 Shared-Exclusive-Use Expansion ROM

The address space from $C800-$CFFF, constituting eight pages or 2K of memory space, is held in common for use by the peripheral slots. Any or all of the peripheral cards can contain up to 2K of ROM (or RAM), which makes use of this address space, but only one can share it with the central computer at any one time.

The peripheral card is expected to contain a flip-flop, which is to be turned on by the DEVICE SELECT (negated) signal previously mentioned (the one which activates the 256-byte page of exclusively addressed ROM for that slot). This warning occurs when the central machine selects the individual peripheral card. In effect, it notifies the peripheral that it is responsible for responding to any requests for information from within the shared (common) address range $C800-$CFFF. Full activation occurs only when the central machine calls for an address within that range. The I/O STROBE (negated) associated with pin 20 on each peripheral connector notifies the peripheral cards that the central machine is accessing this common area, but only one will have been pre-selected to provide the information, and hence only one will respond.

A peripheral card's 256-byte ROM can regain sole access to this address space whenever required, by referring to location $CFFF, a special location that all peripheral cards should recognize as a signal to turn off their flip-flops to disable the expansion ROM. Such a call should be part of every peripheral's initialization routine to make sure that other peripheral slots do not accidentally have their flip-flops still active and hence might accidentally also respond to the central machine's request directed to the selected slot. (It will, of course also turn off its own flip-flop, but the next access by the central machine will turn the flip-flop back on for the the selected slot and the selected slot only.)

Chapter XIX
Applesoft BASIC Interpreter

19.1
The Applesoft Dialect of BASIC

19.1.1 Features of the Applesoft Dialect

The Applesoft interpreter allows the user to specify problem-solving procedures using the Applesoft dialect of BASIC. This dialect is a rich, extended precision floating-point dialect of BASIC. Applesoft includes the ability to perform significant floating-point arithmetic and string operations not available in the other major Apple BASIC dialect, Integer BASIC.

The Applesoft interpreter supports all the functions of minimal BASIC plus many BASIC extensions. It was originally written for Apple Computer Inc., by Microsoft. As first written in 1976, it was mostly a transfer to the Apple hardware/firmware environment of Microsoft's MITS BASIC. Programs written in that version of the BASIC language, which do not depend too heavily upon system-specific programming techniques, are easily transposed into Applesoft.

Specifically, Applesoft supports the use of both Integer and floating-point arithmetic for numbers, numeric variables, and for multi-dimensional numeric arrays. Matrix operations are not explicitly supported. Applesoft also supports the use of strings of characters, string variables and string arrays. A variety of useful string manipulative operations and functions are also imbedded in Applesoft. These include concatenation, splitting strings apart and finding substrings, converting characters to their ASCII code numeric equivalents and *vice versa*, etc. Finally, the Applesoft interpreter also supports a variety of system-specific extensions to the BASIC language. These include special input, output, display-control, and low- and high-resolution graphics commands, as well as useful error-handling capabilities.

This chapter does not attempt to duplicate the *Applesoft BASIC Programming Reference Manual*. Instead, its emphasis is on how Applesoft fits into the overall hardware/software environment of the Apple system. In the process of covering this, it attempts to cover enough of the inner workings to enable a sophisticated user to understand and use them to his advantage.

19.1.2 Variations in the Applesoft Interpreter For Different Hardware/Software Environments

In an Apple II Plus, Applesoft BASIC is the language of the BUILT-IN ROMs. In the Apple II (non plus), it is not.

In an Apple II Plus the Applesoft interpreter is located in five large ROM chips on the main circuit board of the Apple (ROMs D0, D8, E0, E8, and F0). This version of the interpreter is known as ROM Applesoft or, less frequently, as 'firmware Applesoft.' Architecturally it occupies addressable memory locations $D000-$F7FF.

In an Apple II, Integer BASIC is the language of the BUILT-IN ROMs. Of course, that does not mean you can't use Applesoft if you don't have an Apple II Plus. Applesoft adapter cards are available to provide built-in (ROM) Applesoft. In this case, the ROMs are located on the Applesoft card. Architecturally the card is arranged to provide for automatic bank switching between this set of ROMs and those on the Apple's main circuit board. Thus both sets of ROMs, the built-in set for Integer BASIC and the Applesoft set on the adapter card, are able to use addressable memory space in the region $D000-$F7FF.

Alternatively, you can use a language card, such as that used by Apple Pascal, or a similar 16K RAM card. If you have such a card (or a 32K, 64K, or 128K card with similar characteristics) you can automatically load the firmware version of the Applesoft interpreter into it from a DOS 3.3 System Master or from a BASICS diskette during a system boot. Once this is done the language card RAM is automatically write-protected (under software control). The Applesoft interpreter then becomes almost indistinguishable from the ROM version built into an Apple II Plus or the language card. In this situation, even though the interpreter is located in special write-protected memory, the memory now functions as a Read Only Memory (ROM).

Because it is protected against writing (like a ROM) and is located in the ROM area of memory, the version of the Applesoft interpreter made available to the Apple system in this way is also called ROM Applesoft.

If you have an Apple II (as oppposed to an Apple II Plus) with neither an Applesoft card nor a language card (or equivalent) you can still use Applesoft. However, you will not be able to locate it in the ROM area of memory ($D000 up). Instead you will have to use another, older version of Applesoft that can be automatically loaded into normal RAM memory space locations $800

through $3000. This version of the Applesoft interpreter is called RAM Applesoft because it resides electronically in RAM and also resides in RAM address space rather than in the ROM address space.

In older Apple publications it is sometimes called Cassette Applesoft because it was loaded into the Apple from cassette tape before Apple computers had diskettes available to them. (It still can be, but this is not recommended.)

The internal structure of the RAM version is older and slightly different in detail from the more modern ROM version and is not documented in detail here. Some of the routines may be found by using a downwards offset of $C800 (unsigned decimal 51200; signed decimal −14336) bytes from the ROM versions which are documented.

For most routine programming activities, RAM Applesoft is functionally almost the same as ROM Applesoft, but it does use up approximately 10K of RAM space that would otherwise be available for user programs and data.

Unfortunately, the 10K area, which thus becomes unavailable, includes high-resolution graphics Page 1 and text/low-resolution graphics Page 2. As a result, RAM Applesoft has some severe limitations compared to ROM Applesoft for users interested in doing animation and other types of graphics programming which require availability of both high-resolution or low-resolution graphics pages.

19.2 The Functioning of the Interpreter

19.2.1 Overview

The Applesoft interpreter simulates and provides a program-development and operating environment of a computer that accepts and executes BASIC programs written in the Applesoft dialect of BASIC.

Unlike a compiler (or an assembler) the interpreter does not translate the entire program into machine language at one time, then as a separate activity, execute the machine code. Instead the interpreter compacts the program into a tokenized form and stores that compacted form of the instructions as well as space for appropriate simple variables, arrays, character strings, and constants as if it were loading a program.

When told to 'RUN' the program, the computer translates each instruction on-the-fly by means of firmware just before execution.

If these functions were performed by hardware instead of firmware, you would have a computer that would accept instructions in BASIC, store them in compacted form, and execute them directly. This same organization permits the simulated Applesoft BASIC computer to accept instructions written without line numbers (which specify the order they are to be stored in memory) as instructions to be executed immediately in a 'desk calculator' mode.

19.2.2 The BASIC Cycle of Functional Operation For the Applesoft Interpreter

Figure 19.2A shows a simplified version of the basic cycle of functional operation of the Applesoft interpreter.

Figure 19.2A
Functional Operating Cycle of Applesoft Interpreter

NOTE: For simplicity one BASIC instruction per line assumed
Many options, particularly ones associated with utility commands and errors, also ignored for simplicity

```
1. Initialization:
   (Initial set-up)

2. Accept Instructions:
   A.  Request input by displaying an Applesoft prompt `]'
   B.  Accept input from keyboard - or from cassette or disk
   C.  Compact instructions into `tokenized' form

3. Test First Character of Input:
   If first character of the input is a numeric digit, then
   go to line 4, otherwise go to line 5

4. (Deferred Mode - First character is a digit)
   A.  Store `tokenized' instruction in position determined by
       line number, `pushing down' any instructions, variables
       etc which must be moved to make this possible
   B.  Go to step 2.

5. (Immediate Mode - First character is not a digit)
   A.  Execute the inputted instruction line
       NOTE: This may cause desk-calculator type of operatation
       NOTE: If RUN instruction encountered go to step 6.
   B.  Go to step 2.

6.  Run the program using FETCH-EXECUTE Cycle of Operation
   A.  FETCH THE NEXT INSTRUCTION:

       Initial Entry: Start at the BASIC line-nr specified with
       the RUN or if none specified at the lowest in the program
       Non-Initial Entries: Use the next BASIC line-nr.

       Either type of entry: If there is a next line-nr then
       go to step 6B.  If not, go to step 2.

   B.  EXECUTE THE INSTRUCTION:
       When instruction execution completed go to step 6A
```

19.2.3 Functional Utilization of Memory Space By Applesoft Programs

When the Functional Cycle described in the previous section is being used to enter a User's Applesoft program, it has to store the tokenized version of that program away in available user memory. It also has to make provision for space

to locate the constants, variables, arrays, and character strings needed for the program to execute properly. Chapter 16 treated the allocation of space in user memory in careful detail, so it is only necessary to indicate the functional building-blocks the interpreter must set aside.

Figure 19.2B provides such a functional breakout. Remember that, when Applesoft is ready to begin entry of a new program:

1. The lowest memory address available in user memory is called LOMEM.
2. The highest available is called HIMEM.
3. The as yet unused space between is called user free space. This is the space into which user programs and program data (constants, variables, arrays, character strings, etc.) are automatically put by the Applesoft interpreter.
4. LOMEM does *not* remain fixed while a BASIC program is being entered. It is the bottom of space available for variables and is *pushed upward* by the growing program.
5. When you create a BASIC program using the Applesoft interpreter, the interpreter automatically allocates space out of the free space area to meet the four major needs indicated in figure 19.2B.

Figure 19.2B
Functional Utilization of Memory Space by Applesoft Programs
(Blocks Maintained Separately by the Applesoft Interpreter)

1. Space for your BASIC program:
   The Applesoft interpreter puts a `tokenized` (specially abbreviated) copy of your source (BASIC) program at the beginning of the originally available free space. The detailed organization of this space was covered in Chapter 15. Note that as statements are added each statement added pushes LOMEM and the space allocated for simple variables and arrays upward out of its way.

2. Space for Simple Variables:
   The Applesoft interpreter assigns space above the program to simple variables, i.e. variables which are not part of an array. There are three types of these: Real number variables, Integer number variables and String Pointers. String pointers are associated with string variables, but they do not contain the alphanumeric text of the string variable. They merely point to the location of the start of the string of characters and specify its length. The exact organization of this space was covered in Chapter 15.

3. Space for Arrays:

   The Applesoft interpreter assigns space above that assigned to simple variables to arrays. As with simple variables there are three types of arrays: Real number arrays, Integer number arrays and String pointer arrays. As was sthe case for string variables the actual alphanumeric characters of string arrays do not appear in the string pointer arrays, only `pointers` which specify where the alphanumeric characters are located. The exact organization of this space was covered in Chapter 15.

4. Space for Character Strings:
   The actual alphanumeric characters associated with string variables and string arrays as well as `quoted` character strings used as parts of a program statement are put into memory in the order of receipt. As demostrated in Chapter 15 these strings are often imbedded in the body of the program, but if entered during the execution of a program they are allocated at the top of free space and work DOWNWARD from HIMEM. The current string and multiple superseded copies of strings assigned to the same string variable may be in this space at the same time and gradually eat away at the free space. The user may clean out the no-longer-needed `garbage' by using the FRE() functionat a time convenient to him or let the computer do it automatically when space is exhausted.

Notice that with the scheme of allocation shown in figure 19.2B, as a program increases in size, it eats away at the available free space from both the original LOMEM upward and HIMEM downward, leaving an ever-decreasing residue of the original free space somewhere in the middle.

19.3
Structure of the Applesoft Interpreter

19.3.1 The Interpreter as Simulator of a Computer Whose Machine-Language is BASIC

If you feel more comfortable with hardware than with software and like the idea of analyzing systems from a hardware-oriented viewpoint, you will find that the Applesoft interpreter is, in effect, a simulator that makes the Apple simulate a computer operating in BASIC. The Functional Operating Cycle of the Applesoft interpreter documented in figure 19.2A is, in effect, the FETCH-EXECUTE cycle of this simulated computer.

If we look at the interpreter as implementing a simulated machine, certain questions about key parts of the control unit of the simulated machine immediately come to mind: Where does the simulated program counter keep track of where to FETCH the next line of BASIC? Where is the BASIC statement/instruction register that provides linkage between the instruction and the decoder? (The decoder analyzes the BASIC statement/instructions to determine what action is to be EXECUTEd.) Answer: They don't exist as specific locations, like hardware registers, to which all program-control information is moved. They are phantoms which, as soon as you locate them, fade away and appear somewhere else.

In effect, the program counter and decoding circuitry move to the spot where they are needed in the program being executed rather than *vice versa*.

You have already had a chance, in chapter 15, to look at the inner structure of several different BASIC programs in the tokenized form they use in computer memory. Each statement is one instruction for the simulated BASIC computer.

As we have seen, each statement consists of the following components:

1 . A POINTER to the next line of the program.
2. The LINE NUMBER of the statement itself.
3. A BASIC TOKEN that specifies the kind of OPERATION to be performed.

4. Zero or more PARAMETERS that specify what information is to be used in performing that operation.
5. A delimiter specifying that the END of the statement (or end of a line of statements) has been reached.

The pointer and the line number perform the functions of a hardware instruction counter that moves around in memory with the point in the program currently being executed. Two of these four bytes keep track of where you are in the program (in BASIC), and the other two keep track of where (in hardware memory) the simulator must go to FETCH the next statement/instruction.

The remainder of the tokenized statement acts as a floating instruction register. Machine-language instructions normally consist of two parts:

1. An OPERATION CODE that tells what is to be done, and
2. Zero or more addresses or parameters that specify what information is to be used in performing the operation.

The internal structure of the tokenized statement follows this same pattern:

1. The TOKEN takes the place of the OPERATION CODE; it tells what is to be done.
2. The PARAMETER LIST takes the place of the machine-language address parameters; it tells what information is to be used and where to find it.

The structure of BASIC is more free in form that that of machine language. In machine language, the hardware has built-in knowledge of how many parameters/addresses are to be used for each operation code. With free-form statements such as the LET or PRINT, there is no way to tell in advance how long the statement is going to be until the user terminates it with a colon or with a carriage return at the end of the line of typing. Thus it makes sense to use an end-of-parameters delimiter to specify the length of the statement. In the early days of the modern computer this technique was actually used in the hardware of some computers, called variable-word-length computers. At one time this technique was popular with business data processing computers. It was built into the hardware of computers such as the IBM 705.

Various other locations in memory, especially zero page, act like other registers for the simulated machine, keeping track of information while it is needed, flagging special conditions (like the hardware status register), etc.

Instead of using decoding circuits in the control unit to analyze what is to be done, the simulator uses tables with software-implemented look-up for analysis of the statements, and to choose subroutines that will actually execute the required operations.

19.3.2 Program Structure of the Interpreter

I find it convenient to visualize the program structure of the Applesoft interpreter in terms of the eight program structure units described below. While this division and breakdown makes sense to me, it is not perfect. For other viewpoints, I recommend reading C.K. Mesztenyi, "Applesoft Internal Structure," *Washington Apple Pi*, Vol. 3, Number 10 (Nov 81); *Call —A.P.P.L.E.*, Vol. 5, Number 1 (Jan 82); John Crossley, "Applesoft Internals," *Apple Orchard*, Vol. 1, Number 1 (Mar/Apr 80); and Val Golding, "Applesoft from Bottom to Top," *Call —A.P.P.L.E.*

1. The BASIC Program
   The User Memory area from Start-of-Program (often $0801)to LOMEM. (Refer back to Chapter 16 for specific information and details as to how this area is organized and used.)
2. The BASIC Variables and Arrays
   The User Memory area from LOMEM to the End of Array space. (Also refer to Chapter 16 for specific information and details on how this area is organized and used.)
3. The Statement and Program Building Software
   This consists of a diffuse group of program packages that perform functions associated with the input and set-up of BASIC programs. I put in this category such functions as Initialization of the interpreter (its zero-page locations and pointer locations), the actual input of BASIC programs (the input request software starts at $D43C), and the tokenization and laying down of the program and data. (The tokenization subroutine is located at $D559-$D619 with entry at $D559.)
4. The Applesoft Interpreter Control
   This control is diffuse but involves activities associated with the FETCH-EXECUTE cycle for BASIC. It centers around the CHRGET/CHRGOT routine. This routine resides in ROM at $F10B-$F126, but is copied into Page Zero locations starting at $B1 for actual use. The execution phase of this interpreter control is associated with an execution loop entered at $D805. Page Zero memory location TXTPTR

($B8, $B9), imbedded in CHRGET/CHRGOT, is the closest approximation to a classic Program Counter for BASIC programs existing inside the Applesoft Interpreter.

5. The Keyword Token Table ($D0D0-$D25F)
   This is the table the interpreter control (CHRGET and its execution loop) uses to figure out what operation is required in each BASIC statement.

6. The Statement Type Entry Table and its ancillaries, the Operator Tag and Entry Table, and Function Entry Table
   These are used by the interpreter control after the keyword token table. They bring the process closer to performing useful action by relating a particular operation, specified by a particular keyword, to specific subroutines that will implement that operation using the parameters supplied.

7. The Execution Subroutines
   These acutally perform (in most cases with considerable help from the system monitor) whatever it is that the BASIC statement was supposed to do. ADDing, PRINTing, etc.

8. Miscellaneous
   a. Flags and temporaries used in analyzing and executing the program.
   b. Scattered, locally used data interspersed in the program.
   c. A table of ASCII error messages for use whenever errors are detected.

Figure 19.3A/1/
Table of Entry Points for BASIC Statement Implementation Subroutines (1st $20 tokens/keywords)

Hex Token	Keyword	Subroutine Entry Pt
<—Table Start		$D000—>
$80	END	$D870
$81	FOR	$D766
$82	NEXT	$DCF9
$83	DATA	$D995
$84	INPUT	$DBB2
$85	DEL	$F331
$86	DIM	$DFD9
$87	READ	$DBE2
$88	GR	$F390
$89	TEXT	$F399
$8A	PR#	$F1E5
$8B	IN#	$F1DE
$8C	CALL	$F1D5
$8D	PLOT	$F225
$8E	HLIN	$F232
$8F	VLIN	$F241
$90	HGR2	$F3D8
$91	HGR	$F3E2
$92	HCOLOR=	$F6E9
$93	HPLOT	$F6FE
$94	DRAW	$F769
$95	XDRAW	$F76F
$96	HTAB	$F7E7
$97	HOME	$FC58
$98	ROT=	$F721
$99	SCALE=	$F727
$9A	SHLOAD	$F775
$9B	TRACE	$F26D
$9C	NOTRACE	$F26F
$9D	NORMAL	$F273
$9E	INVERSE	$F277
$9F	FLASH	$F280
.End of first $20 entries.		
..Continue with next $20..		

Figure 19.3A/2/
Table of Entry Points for BASIC Statement Implementation Subroutines (2nd $20 tokens/keywords)

Hex Token	Keyword	Subroutine Entry Pt
$A0	COLOR=	$F24F
$A1	POP	$D96B
$A2	VTAB	$F256
$A3	HIMEM:	$F286
$A4	LOMEM:	$F2A6
$A5	ONERR	$F2CB
$A6	RESUME	$F318
$A7	RECALL	$F3BC
$A8	STORE	$F39F
$A9	SPEED=	$F262
$AA	LET	$DA46
$AB	GOTO	$D93E
$AC	RUN	$D912
$AD	IF	$D9C9
$AE	RESTORE	$D849
$AF	&	$03F5
$B0	GOSUB	$D921
$B1	RETURN	$D96B
$B2	REM	$D9DC
$B3	STOP	$D86E
$B4	ON	$D9EC
$B5	WAIT	$E784
$B6	LOAD	$D869
$B7	SAVE	$D8B0
$B8	DEF	$E313
$B9	POKE	$E77B
$BA	PRINT	$DAD5
$BB	CONT	$D896
$BC	LIST	$D6A5
$BD	CLEAR	$D66A
$BE	GET	$DBA0
$BF	NEW	$D649
<—Table End		$D07F—>

Figure 19.3B
Function Entry Table

Hex Token	Keyword	Function Entry Point
<—Starts at		$D080 —>
$D2	SGN	$EB90
$D3	INT	$EC23
$D4	ABS	$EBAF
$D5	USR	$000A
$D6	FRE	$E2DE
$D7	SCRN(	$D412
$D8	PDL	$DFCD
$D9	POS	$E2FF
$DA	SQR	$EE8D
$DB	RND	$EFAE
$DC	LOG	$E941
$DD	EXP	$EF09
$DE	COS	$EFEA
$DF	SIN	$EFF1
$E0	TAN	$F03A
$E1	ATN	$F09E
$E2	PEEK	$E764
$E3	LEN	$E6D6
$E4	STR$	$E3C5
$E5	VAL	$E707
$E6	ASC	$E6E5
$E7	CHR$	$E646
$E8	LEFT$	$E65A
$E9	RIGHT$	$E686
$EA	MID$	$E691
<—Table end		$D0B1—>

Figure 19.3C
Table of Operator Tags & Entry Points

Hex Token	Keyword or Opr	Hex Tag	Entry Point
<- Table Starts at			$D0B2->
$C8	+	$79	$E7C1
$C9	-	$79	$E7AA
$CA	*	$7B	$E982
$CB	/	$7B	$EA69
$CC	^	$7D	$EE97
$CD	AND	$50	$DF55
$CE	OR	$46	$DF4F
$CF	>	$7F	$EED0
$D0	=	$7F	$DE98
$D1	<	$64	$DF65
<— Table Ends at			$D0CF->

Chapter XX
The System Monitor Location—Memory Pages 248-255 ($F800-$FFFF)

20.1
Overview

The system monitor, an important item of software/firmware in the Apple II, is used whether you are programming in Applesoft, Integer BASIC, machine language, PASCAL, FORTRAN, LOGO, or almost any other language. Without it you could not get information into or out of the Apple. The keyboard, text display, graphics display, and the disk drives would all be inactive and unusable.

Actually, as we stepped our way up the Apple's memory, most of the features that we described were a description of functions implemented through the Apple monitor firmware, rather than a discription of features built into the Apple hardware. Indeed, changes in the Apple system monitor are likely to be more noticeable to the Apple system user than changes in hardware.

As with the hardware of the computer, the monitor speaks binary. It is the firmware of the monitor that makes it possible for you to enter hexadecimal digits, decimal numbers or alphabetic characters into the keyboard. It is the firmware of the monitor that echos back your keystrokes, controls the display of information on the screen, scrolls the screen, beeps the bell, and creates the convenient interactive man-computer communication that is so characteristic of the Apple. Without the monitor, the Apple would need a console full of lights and switches.

Some of the functions performed by the monitor in the Apple are performed by hardware in some personal computers. However, in the early years of computer development, engineers learned that it is neither feasible nor desirable to build the majority of the functions performed by the monitor into the system hardware. More than any other software used in the computer, the monitor demonstrates the complete inseparability of hardware and software design at the machine level in a modern computer.

In another sense, the monitor firmware is virtually indistinguishable from hardware. It is physically present in every Apple II or II+ because it is permanently imbedded in the F8 ROM of each system as delivered. Moreover, it is unchangeable by the user (unless he has the right kind of memory expansion card and wants to play special tricks).

20.2
The Two Varieties of Apple Monitors

There are two major versions of the Apple System monitor:

a. The Autostart Monitor, used in Apple II+ systems and
b. The (old) Apple System Monitor, used in Apple II systems which are not II+ systems.

The presence of the system monitor is more noticeable in the Apple II than in the Apple II+. The Autostart version of the monitor in the II+ is shy and self-effacing; you almost never see it unless you specifically ask to do so. You seldom have to do so, unless you wish to examine or use the detailed inner workings and hidden mechanisms of the system. Another way to see it is to get the system so thoroughly bollixed up that the system has to drop out of BASIC into the machine-language level.

In contrast, the original Apple II monitor brazenly showed its '*' prompt every time you turned your system on. To get out of the grip of the monitor you have to take overt action, e.g. enter a CTRL-B to get into the BASIC language.

The major differences between the II and II+ are as follows:

a. AUTOSTART/RESET: When the system power is turned on, or the 'Reset' button is pushed, the Apple II+ (Autostart) monitor will initiate a cold- or warm-start and bring the Apple II+ up in BASIC. On startup it may automatically start running the 'Hello' program. When you turn an Apple II (old monitor) system on, the system comes up in the monitor mode, ready only to accept a monitor command.

b. EDITING: The Apple II+ (Autostart) monitor provides the easy-use ESC-I -J -K and -M keyboard-control capabilities for moving the cursor up-, left- right- and downward by arbitrarily large amounts. The Apple II (old) monitor does not have these capabilities, only the much less convenient EXC-A -B -C and -D capabilities for moving only a single step (capabilities which remain available in the II+).

c. STOP OUTPUT/RESTART: The Apple+ (Autostart) monitor provides the CTRL-S 'stop-list' capability for suspending output of most BASIC programs and listings. Output can later be restarted by pressing any key. The Apple II (old) monitor does not provide such capabilities.

d. SINGLE-STEP and TRACE: The Apple II+'s Autostart monitor does not support the important machine-language debugging aids of SINGLE-STEP and TRACE, whereas the Apple II's old monitor does.

e. MINI-ASSEMBLER, FLOATING-POINT ARITHMETIC PACKAGE and SWEET-16 INTERPRETER: These important machine-language development tools are not available on the Apple II+'s monitor ROM. They were squeezed out by the code needed to implement the extra features described earlier. However, many machine-language tools such as the Disassembler remain.

20.3 Communicating With the System Monitor

You communicate with the monitor by means of monitor commands entered from the keyboard, by monitor commands imbedded in programs, or by setting monitor parameters and running monitor subroutines directly without use of monitor commands. We have repeatedly used all three of these methods for communicating with the monitor in earlier chapters.

For example, starting in Chapter 3 we have used monitor commands entered from the keyboard whenever we wanted to get information about the contents of a memory location in binary/hexidecimal form. Starting in Chapter 5, we have also used monitor commands inside BASIC programs. We have been using monitor subroutines directly ever since we learned to use the CALL statement in Chapter 5. However, it seems worthwhile to summarize how you communicate with the monitor, especially when you do so directly from the keyboard.

First, how do you know if you are in direct communication with the monitor? You look at the prompt on the computer screen. If it is an asterisk ('*'), then you are in direct contact with the monitor.

Next, how do you get into direct contact with the monitor if you are not already there? If you are using an Apple II (which uses the old monitor), just press the 'RESET' key (or CTRL-'RESET' if your Apple is set up to protect against accidental resets). If you have an Apple II+ or a system which uses the Autostart version of the monitor, just CALL -151.

How do you type commands into the monitor? The monitor recognizes 22 different command characters, which in appropriate context specify WHAT action is to be taken. In many cases a command is not complete or grammatically correct unless additional information in the form of ADDRESSES or DATA VALUES is also supplied.

Addresses and data values are always specified in bit-oriented form. Since it is difficult to keep track of bits, the monitor uses hexadecimal abbreviations to accept (and printout when relevant) addresses and data values.

Today we have a widely accepted convention which says that hexadecimal numbers are written with '$' prefix to provide quick and easy visual distinction from decimal numbers. Unfortunately this convention is NOT used in the Apple monitor. Why not? The Apple monitor was one of the earliest parts of the Apple system to be developed and it was developed before this convention had been as widely adopted as it has today. The '$' convention should NOT be used with the Apple monitor. '$' has its own unique meaning within the monitor as a command to the mini-assembler (which is built into the non-autostart version of the monitor) to execute a monitor command from the mini-assembler.

ANY number typed into the monitor as an address or data value is ALWAYS treated as a hexadecimal number without any special designation preceding or following it.

20.4 Summary of Monitor Commands Directly Available to the Programmer

The 'Apple II Reference Manual' supplied to you when you purchased your Apple has detailed information about each of the monitor commands. The summary of monitor commands following in Figure 20.4A is adapted from a table in that key reference source and should give you an idea of the most important commands and variants on commands.

Figure 20.4A - Summary of Monitor Commands

```
EXAMINING 6502 'HARDWARE REGISTERS

  CTRL-E                    Displays the contents of the 6502`s registers

EXAMINING MEMORY:
  {adrs}                    Displays the hex value of the data in {adrs}
  {adrs}.{adrs}             Displays the hex values of the data in all locns
                            from {adrs1} to {adrs2}
  'RETURN'                  Displays the hex values of the contents of up to
                            eight locations following the last opened locn

CHANGING THE CONTENTS OF MEMORY:
  {adrs}: {val} {val} ...   Stores the values specified in consecutive
                            memory locations starting with {adrs}
  : {val} {val} ...         Stroes the values specified in consecutive order
                            starting with the next changeable location

MOVING AND COMPARING THE CONTENTS OF MEMORY:

  {dest}<{start}.{end}M     Moves (copies) the values in the range
                            {start}.{end} into the range starting at {dest}
  {dest}<{start}.{end}V     Verifies (Compares) that values of locns in the
                            range {start}.{end} have the same values as
                            those in thecomparison range beginning at {dest}

SAVING AND LOADING INFORMATION VIA CASSETTE TAPE

  {start}.{end}W            Writes the values of info in the range
                            {start}.{end} onto tape preceded by 10sec leader
  {start}.{end}R            Reads values from tape, storing them in memory
                            locations beginning at {start} and stopping at
                            {end}.  Prints `ERR` if mismatch occurs

RUNNING PROGRAMS
  {adrs}G                   Goto {adrs}, i.e.,transfer control to the
                            machine-language program beginning at {adrs}
  CTRL-Y                    Jump to subroutine whose locn specified in $3F8

DISASSEMBLING/LISTING PROGRAMS

  {adrs}L                   Disassemble and Displace as symbolic machine-
                            language the next 20 instructions starting with
                            {adrs} as the first byte of the first instruct`n

ASSEMBLING MACHINE-LANGUAGE PROGRAMS (MiniAssembler not available in II+)**

  F666G                     Invoke the Mini-Assembler
  $                         Execute a monitor command from Mini-Assembler
  $FF69G                    Exit the MiniAssembler
  {adrs}S                   Disassemble, display and execute the instruction
                            at {adrs} and display contents of 6502`s
                            internal registers. Each `S` another instruction
  {adrs}T                   Trace or step infinitely. Stop only when a BRK
                            instruction is encountered or `RESET` key pushed

DIVERT INPUT OR OUTPUT

  {slot} CTRL-P             Divert output to the device whose interface card
                            is in slot# {slot}. Slot 0 = display screen
  {slot} CTRL-K             Divert input to the device whose interface card
                            is in slot# {slot}. Slot 0 = keyboard

CHANGE DISPLAY MODE
  I                         Set Inverse Display Mode
  N                         Set Normal Display Mode

ENTER OR REENTER BASIC

  CTRL-B                    Enter language built-into specific Apple`s ROM
  CTRL-C                    Warm-start re-entry w/o total re-initialization

HEXADECIMAL ARITHMETIC

  {val1}+{val2}             Add two hex values and print hex result
  {val1}-{val2}             Subtract second hex value from first and
                            print answer
```

** Unless RAM card is loaded with Integer BASIC.

INDEX TO PROGRAMMERS' GUIDE
(Part I of What's Where in the Apple)

ATLAS

***** For Apple //e specific information, see Appendix A *****

Use-Type Guide

/SE/

1st letter — type information
2nd letter — usage/length information

Type Codes:

S — Subroutine
P — Parameter
H — Hardware
B — Buffer

Usage/Length Codes:

E — Entry
B — Block
n — n-Byte Long
L — Label
F — Flag

Some Common Combinations:

P1: 1-Byte Parameter
Pn: n-Byte Parameter
PB: Parameter Block
HL: Hardware Location
HB: Hardware Block
FF: Hardware Flag
SE: Subroutine Entry Point
SB: Subroutine Block
SL: Subroutine Label
BB: Buffer Block

What's Where Atlas Updates

The following subroutines have been relocated in the new (autostart) ROMS

Subroutine	Old Monitor Applesoft	New Autostart Applesoft
HGR2	F3D4	F3D8
HGR	F3D3	F3E2
HCLR	F3EE	F3F2
BKGND	F3F2	F3F4
HPOSN	F40D	F411
HPLOT	F453	F457
HLIN	F530	F53A

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION

$0000~$FFFF (0~-1) \HB\           ADDRESS RANGE OF APPLE II ($0000~$FFFF SIGNED DECIMAL EQUIV IS 0~32767 FOLLOWED BY
                                  -32768~-1)
$0000~$BFFF (0~-16385) \HB\       RAM ADDRESS RANGE OF APPLE II (NOT INCLUDING RAM IN LANGUAGE CARD IF PRESENT)
$0000~$00FF (0~255) \HB\          HARDWARE PAGE ZERO
$0000~$001F (0~31) [(R0-R15)] \PB\'SWEET-16' REGISTERS R0 THRU R15 OF 'SWEET-16' (16-BIT INTERPRETER IN MONITOR)
$0000~$0002 (0~2) \SE\            APPLESOFT SOFT REENTRY (OG IS EQUIVALENT TO CTRL~C)
$0000~$0001 (0~1) [ROL~ROH] \P2\  'SWEET-16' REGISTER R0 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0000 (0) [LOC0] \P1\             MONITOR MEMORY LOCATION 'LOC0'. PRESET TO $4C (JMP) - (JUMP ADDRESS IN $001~$002)
$0001~$0002 (1~2) [LOC1] \P2\     MONITOR MEMORY LOCATION 'LOC1' - POINTER PRESET TO ADDRESS OF APPLESOFT SOFT ENTRY
$0002~$0003 (2~3) [(R1)] \P2\     'SWEET-16' REGISTER R1 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0003~ $0005\SE\                  APPLESOFT JUMP COMMAND TO $F128 (HARD ENTRY?)
$0004~$0005 (4~5) [(R2)] \P2\     'SWEET-16' REGISTER R2 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0006~$0007 (6~7) [(R3)] \P2\     'SWEET-16' REGISTER R3 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0008~$0009 (8~9) [(R4)] \P2\     'SWEET-16' REGISTER R4 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$000A~$0016 (10~22) [(A/S RESVD)] \PB\APPLESOFT RESERVED BLOCK IN PAGE ZERO
$000A~$000C (10~12) \SI\          APPLESOFT LOCN FOR USR FUNCTION'S JUMP INSTRUCTION
$000A~$000B (10~11) [(R5)] \P2\   'SWEET-16' REGISTER R5 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$000C~$000D (12~13) [(R6)] \P2\   'SWEET-16' REGISTER R6 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$000D (13) [CHARAC]               APPLESOFT - USED BY STRLT2 STRING UTILITY
$000D~$0016 (13~22) \PB\          GENERAL PURPOSE COUNTERS/FLAGS FOR APPLESOFT
$000E~$000F (14~15) [(R7)] \P2\   'SWEET-16' REGISTER R7 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$000E (14) [ENDCHR]               APPLESOFT - USED BY STRLT2 STRING UTILITY
$0010~$0011 (16~17) [(R8)] \P2\   'SWEET-16' REGISTER R8 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0011 (17) [VALTYP]               APPLESOFT FLAG FOR LAST FAC (FLOATING ACCUMULATOR) OPERATION: $00 = NUMBER; $FF=STRING
$0012~$0013 (18~19) [(R9)] \P2\   'SWEET-16' REGISTER R9 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0014~$0015 (20~21) [(R10)] \P2\  'SWEET-16' REGISTER R10 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0014 (20) [SUBFLG]               APPLESOFT SUBSCRIPT FLAG: $00= SUBSCRIPTS ALLOWED;$80= SUBSCRIPTS NOT ALLOWED
$0016~$0017 (22~23) [(R11)] \P2\  'SWEET-16' REGISTER R11 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$0016 (22) [(COMPRTYP)] \P1\      APPLESOFT~ PARAMETER TO CONTROL TYPE OF COMPARISON MADE BY FLOATING POINT COMPARISON
                                  ROUTINE AT $DF6A (1:> ;2:= ;3:>= ;4:< ;5:<> ;6:<=)
$0018~$0019 (24~25) [(R12)] \P2\  'SWEET-16' REGISTER R12 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$001A~$001B (26~27) [(R13)] \P2\  'SWEET-16' REGISTER R13 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$001A~$001B (26~27) [SHAPEL~SHAPEH] \P2\HI-RES POINTER TO SHAPE LIST (ON-THE-FLY SHAPE POINTER)
$001C~$001D (28~29) [(R14)] \P2\  'SWEET-16' REGISTER R14 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
$001C (28) [HCOLOR1] \P1\         HI-RES RUNNING COLOR MASK (ON-THE-FLY COLOR BYTE)
$001D (29) [COUNTH] \P1\          HI-RES GRAPHICS HIGH-ORDER BYTE OF STEP COUNT FOR LINE
$001E~$001F (30~31) [R15L~R15H] \P2\'SWEET-16' REGISTER R15 (USED AS PROGRAM COUNTER IN 16-BIT PSEUDOMACHINE IN APPLE
                                  SYSTEM MONITOR) (REG-R15 )
$0020~$0055 (32~85) [(MONITOR RESVD)] \PB\APPLE II SYSTEM MONITOR RESERVED LOCATIONS ($0050~$0055 USED ONLY BY
                                  MULTIPLY-DIVIDE ROUTINES AND THUS AVAILABLE IN MANY SITUATIONS)
$0020~$004F (32~79) [(AUTOSTART RESVD)] \PB\AUTOSTART MONITOR RESERVED LOCATIONS
$0020 (32) [WNDLFT] \P1\          LEFT COLUMN OF SCROLL WINDOW: RANGE 0-39 OR $0~$27. USED ONLY IN VTABZ.
$0021 (33) [WNDWDTH] \P1\         WIDTH OF THE SCROLL WINDOW: RANGE:1 TO 40-(WNDLFT) OR $1 TO $28 - (WNDLFT)
$0022 (34) [WNDTOP] \P1\          TOP LINE OF SCROLL WINDOW: RANGE 0-22($16) FOR FULL TEXT SCREEN 20-22($14~$16) FOR
                                  MIXED SCREEN
$0023 (35) [WNDBTM] \P1\          BOTTOM LINE OF SCROLL WINDOW: RANGE (WNDTOP)+1 TO 24($18).
$0024 (36) [CH] \P1\              CURSOR HORIZONTAL DISPLACEMENT FROM WNDLFT: RANGE 0 TO (WNDWDTH)-1
$0025 (37) [CV] \P1\              CURSOR VERTICAL POSITION RELATIVE TO TOP OF SCREEN: RANGE 0-23 ($0~$17)
$0026~$0027 (38~39) \P2\          PAGE ZERO LOCATIONS USED BY DOS
$0026~$0027 (38~39) [GBASL~GBASH] \P2\MEMORY ADDRESS OF LEFT END POINT OF DESIRED LINE FOR LO-RES PLOT (SET BY GBASCALC)
$0026~$0027 (38~39) [HBASL~HBASH] \P2\HI-RES GRAPHICS ON-THE-FLY BASE ADDRESS (LEFT END POINT OF DESIRED LINE FOR
                                  HI-RES PLOT)
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------
$0026~$0027 (38~39)                 USED AS SCRATCH BY DOS
$0028~$0029 (40~41) [BASL~BASH] \P2\MEMORY ADDRESS FOR LEFT END CHARACTER POS'N OF CURRENT TEXT LINE
$002A~$002F (42~47) \PB\            PAGE ZERO LOCATIONS USED BY DOS
$002A~$002B (42~43) \P2\            USED AS SCRATCH BY DOS
$002A~$002B (42~43) [BAS2L~BAS2H] \P2\USED DURING SCROLLING AS DESTINATION LINE POINTER AS EACH LINE IS MOVED TO
                                    POSITION ABOVE CURRENT
$002C~$002D (44~45) [RTNL~RTNH] \P2\MONITOR RETURN POINTER (POINTS TO SAVE AREA USED BY INSTRUCTION TRACE ROUTINE)
$002C~$002D (44~45) [LMNEM~RMNEM] \P2\ADDRESS POINTER USED BY DISASSEMBLER FOR INDEX TO MNEMONICS TABLE
$002C~$002D (44~45) [TEMP] \P2\     DOS RWTS (READ-WRITE TRACK-SECTOR TEMPORARY STORAGE FOR ADDRESS INFORMATION
$002C (44) [COUNT - CSUM] \P1\      DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER (RETURNS CHECKSUM)
$002C (44) [H2] \P1\                RIGHT END POINT OF A HORIZONTAL LINE BEING DRAWN BY HLINE: RANGE 0-39 ($0~$27)
$002D (45) [V2] \P1\                BOTTOM PT OF LO-RES VERT LINE DRAWN BY VLINE. RANGE: 0-19(~$21) FOR MIXED SCR;
                                    0-23(~$17) FOR FULL SCR
$002D (45) [SECT] \P1\              DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER FOR CURRENT DISK SECTOR
$002E (46) [CHKSUM] \P1\            LOCN WHERE CHECKSUM IS ACCUMULATED DURING CASSETTE TAPE READ
$002E (46) [FORMAT] \P1\            USED BY MINIASSEMBLER & DISASSEMBLER TO SPECIFY FORMAT OF INSTRUCTION FOR DISPLAY
                                    PURPOSES
$002E (46) [MASK] \P1\              LOW-RES COLOR GRAPHICS MASK. $0F OR $F0 TO SELECT HIGH OR LOW NIBBLE TO SPECIFY WHICH
                                    OF 2 PLOT LINES REP BY GBASL~H POINTER
$002E (46) [TRACK - TRKN] \P1\      DOS RWTS (READ-WRITE TRACK-SECTOR) TRACK NUMBER
$002F (47) [LASTIN] \P1\            USED IN CASSETTE INPUT BY RDBIT AS WORK AREA TO DETERMINE WHETHER INPUT HAS CHANGED
$002F (47) [LENGTH] \P1\            USED BY DISASSEMBLER TO INDICATE LENGTH OF THE INSTRUCTION. ALSO BY TRACE
$002F (47) [SIGN] \P1\              $01 BIT SET AFTER CALL TO MULPM OR DIVPM (SIGNED 16 BIT MULT OR DIV) TO SPECIFY
                                    WHETHER COMPLEMENT NEEDED (NOTE MULPM & DIVPM IN OLD MONITOR ONLY ~ NOT IN AUTOSTART)
$002F (47) [VOLUME] \P1\            DOS RWTS (READ-WRITE TRACK-SECTOR) DISK VOLUME NUMBER
$0030 (48) [COLOR] \P1\             LOW-RES COLOR GRAPHICS COLOR CODE (FOR PLOT/HLIN/VLIN FUNCTIONS) - CONTAINS SELECTED
                                    COLOR VALUES FOR TWO LOW-RES GRAPHICS 'LINES' ONE IN EACH NIBBLE OF BYTE
$0030 (48) [HMASK] \P1\             HI-RES GRAPHICS ON-THE-FLY BIT MASK
$0031 (49) [MODE] \P1\              USED BY MONITOR COMMAND PROCESSING TO INDICATE DISPOSITION OF HEX INFO IN THE INPUT
                                    LINE
$0032 (50) [INVFLG] \P1\            VIDEO FORMAT CONTROL: 255($FF)=NORMAL;127($7F)=FLASHING;63($3F)=INVERSE
$0033 (51) [PROMPT] \P1\            PROMPT CHARACTER WRITTEN TO SCREEN WHENEVER A LINE OF INPUT IS CALLED FOR BY GETLN
                                    ROUTINE
$0034 (52) [YSAV] \P1\              USED BY MONITOR COMMAND PROCESSOR TO SAVE CONTENTS OF Y-REGISTER DURING PROCESSOR
                                    (Y-REGISTER SAVE LOCN FOR MONITOR)
$0035~$0039 (53~57) \PB\            PAGE ZERO LOCATIONS USED BY DOS FOR INTERFACE (DRIVENO CSW & KSW)
$0035 (53) [YSAV1] \P1\             USED TO SAVE CONTENTS OF Y-REGISTER ACROSS A CALL TO SCREEN OUTPUT ROUTINES.
                                    (Y-REGISTER SAVE LOCN FOR COUT1)
$0035 (53) [L] \P1\                 MINIASSEMBER MEMORY LOCATION 'L'
$0035 (53) [DRIVENO] \P1\           DOS DISK DRIVE NO
$0036~$0039 (54~57) [(I/O HOOK TBLS)] \PB\MONITOR OUTPUT & INPUT HOOKS (VECTORS TO DOS OUTPUT & INPUT ROUTINES)
$0036~$0037 (54~55) [CSWL~CSWH] \P2\MONITOR OUTPUT REG & OUTPUT HOOK TO DOS; I.E. ADDRESS OF ROUTINE WHICH IS TO
                                    RECEIVE AND DISPOSE OF OUTPUT CHARACTERS. RESET 0 CTRL-P & PR#0 SET THIS LOCN TO
                                    $FDF0 (MONITOR OUTPUT TO SCREEN); S CTRL-P & PR#S SET THIS LOCN TO $CS00 (SLOT S ROM)
$0038~$0039 (56~57) [KSWL~KSWH] \P2\DOS INPUT HOOK; I.E. ADDRESS OF THE USER INPUT ROUTINE. CONTROLLED BY CURRIN PORT
                                    IN# & KEYIN. RESET ~ 0 CTRL-K & IN#0 SET THIS LOCN TO $FD1B (MONITOR KEYBOARD INPUT
                                    ROUTINE); S CTRL-K &IN#S SET THIS LOCN TO $CS00(SLOT S ROM) (MONITOR INPUT REG)
$003A~$003B (58~59) [PCL~PCH] \P2\SAVE AND CONTROL AREA FOR PROGRAM COUNTER. USED IN BREAK PROCESSING AND
                                    MINIASSEMBLER. SET BY MONITOR CMDS L G S &T (PC SAVED HERE BY MONITOR)
$003C~$0043 (60~67) [XQT/XQTNZ] \PB\8 BYTE WORK AREA FOR INSTRUCTION STEP/TRACE. NEXT INSTRUCTION SOMETIMES MOVED HERE
$003C~$003D (60~61) [A1L-A1H] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A1. MANY USES INCLUDE SOURCE POINTER
                                    DURING MONITOR MOVE
-------------------------------------------------------------------------------------------------------------------------------
$0026 - $003C                  Prof. Luebbert's "What's Where in the Apple"                  NUMERIC ATLAS
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------
$003C~$003D (60~61) [DEVCTBL] \P2\DOS RWTS  DEVICE IN READ-WRITE TRACK-SECTOR PARAMETER POINTING TO DEVICE TABLE.
                              PRESET TO 'PTRSDEST' = POINTER TO DESTINATION  DEVICE IN DEVICE TABLE. NOT A SYNONYM
                              FOR BUFPTR
$003C~$003D (60~61) [DEVCTBL] DOS RWTS (READ-WRITE TRACK-SECTOR) DEVICE TABLE ~ SYNONYM FOR BUFPTR
$003E~$003F (62~63) [BUFPTR] \P2\DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER 'BUFPTR' (POINTS TO DATA BUFFER IN RWTS)
$003E~$003F (62~63) [A2L~A2H] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A2. USED IN CALLING LIST OF MANY MONITOR
                              SUBROUTINES SUCH AS MOVE & CASSETTE ROUTINES
$0040~$0048 (64~72)           PAGE ZERO LOCATIONS USED BY DOS
$0040~$0041 (64~65) [A3L~A3H] \P1\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A3. USED IN CALLING LIST OF MOST MONITOR
                              SUBROUTINES
$0040~$0041 (64~65) [FCBFOP ZPGWRK V NPE] DOS - USED AS GENERAL POINTER BY 1ST LEVEL (COMMAND DECODE) ROUTINES IN DOS
$0041 (65) [TRKCNT] \P1\      DOS DISK SYSTEM FORMATTER SPECIAL TRACK COUNTER
$0042~$0043 (66~67) [A4L~A4H] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A4.  USED IN CALLING LIST OF SOME MONITOR
                              SUBROUTINES
$0043~$0043 (67~67) [ZPGBM3 ZPGFCB] DOS - USED AS GENERAL PURPOSE POINTER BY SECOND-LEVEL DOS ROUTINES
$0044~$0045 (68~69) [A5L~A5H] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A5. USED MOSTLY BY SINGLE-CYLCLE & TRACE
$0044~$0045 (68~69) [CNUM]    DOS - POINTS TO AVAILABLE BUFFER IN OPEN. ALSO USED AS ARITHMETIC REGISTER BY DOS FIRST &
                              SECOND LEVEL ROUTINES
$0044 (68) [FMT] \P1\         MINIASSEMBLER MEMORY LOCATION 'FMT'
$0045 (69) [ACC] \P1\         USER A-REG SAVED HERE ON BRK TO MONITOR & DURING TRACE
$0046 (70) [XREG] \P1\        USER X-REG SAVED HERE ON BRK TO MONITOR & DURING TRACE
$0046 (70) [MONTIME] \P1\     DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER 'MONTIME'
$0046 (70) [EXCNT] \P1\       DOS DISK SYSTEM FORMATTER GENERAL COUNTER
$0047 (71) [YREG] \P1\        USER Y-REG SAVED HERE ON BRK TO MONITOR & DURING TRACE (Y-REG SAVED HERE ON BRK)
$0047 (71) [YCNT] \P1\        DOS DISK SYSTEM FORMATTER NYBBLE COUNTER (ALSO COUNTER FOR DISK-DRIVE MOTOR-ON TIME?)
$0048~$0049 (72~73) [IOBPL~H] \P2\DOS READ-WRITE-TRACK-SECTOR (RWTS)  'IOBPL~H' (INPUT-OUTPUT CONTROL BLOCK POINTER)
$0048 (72) [STATUS] \P1\      USER STATUS REGISTER (P-REGISTER) SAVED HERE ON BRK TO MONITOR & DURING TRACE. WARNING:
                              INITIALIZE BEFORE G FUNCTION TO AVOID DECIMAL MODE IF DOS HAS BEEN USED
$0049 (73) [SPNT] \P1\        USER STACK POINTER (S-REGISTER) SAVED HERE BY MONITOR 'SAVE' ROUTINE  ON BRK & DURING
                              TRACE
$004A~$00DF (74~223) \PB\      PAGE ZERO LOCATIONS USED BY INTEGER BASIC (GAP AT $004E~$0054)
$004A~$004D (74~77)            PAGE ZERO LOCATIONS USED BY DOS
$004A~$004B (74~75) [LOMEML~LOMEMH] \P2\POINTER TO LOMEM (CONTAINS 'START OF BASIC VARIABLES' FOR INTEGER BASIC - START
                              OF PROGRAM FOR APPLESOFT BASIC)
$004A (74) [A] \P1\           DOS DISK SYSTEM FORMATTER DUMMY LOCATION FOR TIMING PURPOSES AND SCRATCH. DOS WILL
                              REPAIR IN INIT COMMAND; USER MUST REPAIR IF RWTS FORMATTER CALLED DIRECTLY
$004B (75) [FILLCNT - SCTR] \P1\DOS DISK SYSTEM FORMATTER GENERAL COUNTER & SECTOR NUMBER
$004C~$004D (76~77) [HIMEML~HIMEMH] \P2\ADDRESS POINTER TO HIMEM (INTEGER BASIC - END OF BASIC PROGRAM)(APPLESOFT -
                              START OF STRING DATA)
$004E~$004F (78~79) [RNDL~RNDH] \P2\16 BIT NO. RANDOMIZED WITH EACH KEY ENTRY DONE BY MONITOR KEYIN ROUTINE (AND BY
                              MANY OTHER ROUTINES SUCH AS SERIAL & COMM CARD WHICH ARE USED TO REPLACE KEYIN).
                              RANDOMIZATION ACCOMPLISHED BY CONTINUOUSLY INCREMENTING WHILE AWAITING KEYBOARD
                              INPUT. HIGH ORDER BYTE $4F
$0050~$00F8 (80~248)          APPLESOFT - PAGE ZERO LOCATIONS USED (GAPS AT  $00D7 $00E3 & $00EB~$00EF)
$0050~$0061 (80~97) [(A/S POINTERS)] \PB\GENERAL PURPOSE POINTERS FOR APPLESOFT (PB)
$0050~$0057 (80~87) [NOUNSTKL] \P8\INTEGER BASIC MEMORY LOCATION 'NOUNSTKL'
$0050~$0055 (80~85) \P6\      MONITOR/INTEGER BASIC MULTIPLY-DIVIDE WORKAREA
$0050~$0053 (80~83) [AC] \P4\  32-BIT EXTENDED ACCUMULATOR USED IN MONITOR 16-BIT MULT & DIVIDE
$0050~$0051 (80~81) [LINNUM] \P2\APPLESOFT GENERAL PURPOSE 16 BIT NUMBER LOCATION (USES INCLUED LOCATION FOR LINE
                              NUMBER)
$0050~$0051 (80~81) [ACL~ACH] \P2\OLD MONITOR (NOT AUTOSTART). USED BY 16 BIT MULT & DIVIDE ROUTINES AS
                              PSEUDO-ACCUMULATOR
-------------------------------------------------------------------------------------------------------------------------------
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------
$0050~$0051 (80~81) [DXL~DXH] \P2\HI-RES GRAPHICS DELTA-X FOR HLIN SHAPE
$0051 (81) [SHAPEX] \P1\       HI-RES GRAPHICS SHAPE TEMP.
$0052~$0053 (82~83) [XTNDL~XTNDH] \P2\OLD MONITOR (NOT AUTOSTART) - USED IN 16-BIT MULT & DIVIDE AS ACCUMULATOR
                               EXTENSION (TO 32 BITS)
$0052 (82) [TEMPPT] \P1\       APPLESOFT TEMPORARY POINT - LAST USED TEMPORARY STRING DESCRIPTOR (SEE DSCTMP)
$0052 (82) [DY] \P1\           HI-RES GRAPHICS DELTA-Y FOR HLIN SHAPE
$0053 (83) [LASTPT] \P1\       APPLESOFT LAST USED TEMPORARY STRING POINTER
$0053 (83) [QDRNT] \P1\        HI-RES GRAPHICS QDRNT: 2 LSB'S ARE ROTATION QUADRANT FOR DRAW
$0054~$0055 (84~85) [AUXL~AUXH] \P2\OLD MONITOR (NOT AUTOSTART) - USED FOR 16-BIT MULT & DIVIDE AS AUXILLIARY REGISTER
$0054~$0055 (84~85) [EL~EH] \P2\HI-RES GRAPHICS ERROR FOR HLIN
$0055 (85)                     APPLESOFT - START OF STRING SCRATCH AREA (LENGTH UNKNOWN - AT LEAST 3 BYTES)
$0058 (88) [SYNSTKH]           INTEGER BASIC MEMORY LOCATION 'SYNSTKH'
$005E~$005F (94~95) [INDEX] \P2\APPLESOFT TEMPORARY (STACK) POINTER FOR MOVING STRINGS
$0060~$0061 (96~97) \P2\       APPLESOFT PARAMETER STORAGE SPACE FOR FLOATING POINT COMPARE ROUTINES
$0062~$0066 (98~102) \P5\      RESULT OF LAST MULTIPLY/DIVIDE (APPLESOFT)
$0067~$006A (103~106) \PB\      PAGE ZERO LOCATIONS USED BY DOS
$0067~$0068 (103~104) [TEXTTAB] \P2\APPLESOFT TEXT TABLE POINTER (POINTS TO TO BEGINNING OF PROGRAM TEXT . DEFAULT
                               VALUE $0801
$0069~$006A (105~106) [VARTAB:] \P2\APPLESOFT VARIABLE TABLE POINTER - POINTS TO TO START OF SIMPLE VARIABLE SPACE (AT
                               END OF APPLESOFT PROGRAM TEXT)
$006B~$006C (107~108) [ARYTAB] \P2\APPLESOFT ARRAY TABLE POINTER (POINTS TO BEGINNING OF ARRAY SPACE)
$006D~$006E (109~110) [STREND] \P2\APPLESOFT STORAGE END POINTER (POINTS TO TOP OF ARRAY STORAGE I.E. TO END OF NUMERIC
                               STORAGE IN USE)
$006F~$0070 (111~112) \PB\      PAGE ZERO LOCATIONS USED BY DOS
$006F~$0070 (111~112) [FRETOP] \P2\APPLESOFT POINTER TO END OF STRING STORAGE OR TOP OF USER-AVAILABLE FREE SPACE.
                               DEFAULTS TO HIMEM - USUALLY $BFFF FOR 48K APPLE)
$0071~$0072 (113~114) [FRESPC] \P2\APPLESOFT TEMPORARY POINTER FOR STRING-STORAGE ROUTINES
$0073~$0074 (115~116) [MEMSIZE] \P2\APPLESOFT HIMEM (HIGHEST LOC IN MEM AVAIL + 1). INIT TO HIGHEST RAM - $BFFF FOR 48K
                               APPLE IF DOS NOT ACTIVE BEGINNING OF DOS IF DOS ACTIVE
$0075~$0076 (117~118) [CURLIN] \P2\APPLESOFT - LINE # OF LINE CURRENTLY BEING EXECUTED NOTE: HI BYTE OF CURLIN TESTED
                               BY DOS FOR DIRECT-DEFERRED MODE USAGE - BYTE SET TO $FF IN DIRECT. IF CONTENTS OF
                               $AAB6<>0 AND IF PROMPT=']' OR IF THIS LOCN CONTAINS $FF DOS ASSUMES DIRECT MODE AND
                               WILL NOT DO OPEN OR OTHER DIRECT MODE COMMANDS
$0077~$0078 (119~120) [OLDLIN] \P2\APPLESOFT - LAST LINE EXECUTED -  LINE # AT WHICH EXECUTION INTERRUPED BY CTRL-C
                               STOP ETC.
$0078~$0097 (120~151) [NOUNSTKH] INTEGER BASIC MEMORY LOCATION 'NOUNSTKH' (NOUN STACK HI BYTE)
$0079~$007A (121~122) [(OLD TEXT PTR)] \P2\APPLESOFT OLD TEXT PTR. PTS TO LOC IN MEM FOR NEXT STMT TO BE EXE
$007B~$007C (123~124) [DATLIN] \P2\APPLESOFT CURRENT LINE # FROM WHICH DATA IS BEING READ
$007D~$007E (125~126) [DATPTR] \P2\POINTS TO ABS LOC IN MEM FROM WHICH DATA IS BEING READ BY APPLESOFT
$007F~$0080 (127~128) [(INP SOURCE PTR)] \P2\APPLESOFT - PTR TO CURRENT SOURCE OF INPUT. $201 DURING INPUT STATEMENT IF
                               STANDARD BUFFER IN USE
$0080~$009F (128~159) [SYNSTKL] INTEGER BASIC MEMORY LOCATION 'SYNSTKL' (SYNTAX STACK LOCATION)
$0081~$0082 (129~130) [(LAST VBL NAME)] \P2\APPLESOFT - HOLDS LAST-USED VARIABLE'S NAME
$0083~$0084 (131~132) [VARPNT] \P2\APPLESOFT POINTER TO THE LAST-USED VARIABLE'S VALUE (USED BY PTRGET)
$0085~$009C (133~156) \PB\      APPLESOFT GENERAL USAGE
$0085~$0086 (133~134) [FORPNT] \P2\APPLESOFT GENERAL POINTER. SEE COPY SUBROUTINE FOR EXAMPLE
$008A~$008E (138~142) [TEMP3] \P5\APPLESOFT REGISTER TEMP3 FOR FLOATING POINT MATH PACKAGE (PACKED 5-BYTE FORMAT)
$0090 (144)                    INITIALIZED TO $4C (JMP)
$0093~$0097 (147~151) [TEMP1] \P5\APPLESOFT REGISTER TEMP1 FOR FLOATING POINT MATH PACKAGE (PACKED 5-BYTE FORMAT)
$0094~$0095 (148~149) [HIGHDS] \P2\USED BY BLOCK TRANSFER UTILITY (BLTU) AS HIGH DESTINATION
$0096~$0097 (150~151) [HIGHTR] \P2\APPLESOFT - USED BY BLOCK TRANSFER UTILITY (BLTU) AS HIGH END OF BLOCK TO BE
                               TRANSFERRED
-------------------------------------------------------------------------------------------------------------------------------
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------

$0098~$009C (152~156) [TEMP2] \P5\APPLESOFT FLOATING POINT MATH PACKAGE REGISTER TEMP2 (PACKED 5-BYTE FORMAT)
$009B~$009C (155~156) [LOWTR] \P2\APPLESOFT GENERAL PURPOSE REGISTER USED BY GETARYPT~FNDLN~BLTU (E.G. LOW END OF BLOCK
                              TO BE TRANSFERRED IN BLTU)
$009D~$00A3 (157~163) [FAC] \P6\APPLESOFT MAIN FLOATING-POINT ACCUMULATOR (USES 6-BYTE UNPACKED MATH PACKAGE FORMAT
                              DESCRIBED BELOW)
$009D~$009F (157~159) [DSCTMP] \P3\APPLESOFT TEMPORARY STRING DESCRIPTOR (SEE VALTYP & TEMPPT)
$009D (157) [FACEXP] \P1\       EXPONENT BYTE OF FAC. SIGNED NUMBER IN EXCESS $80 FORM (SIGNED VALUE HAS $80 ADDED)
$009E~$00A1 (158~161) \P4\      FOUR BYTE MANTISSA OF FAC. BINARY POINT ASSUMED TO RIGHT OF MSB. NAMES OF BYTES IN MATH
                              PACKAGE HO~MOH~MO~LO RESPECTIVELY.
$009E (158) [FACHO] \P1\        HIGH ORDER BYTE OF MANTISSA OF FAC
$009F (159) [FACMOH] \P1\       MIDDLE ORDER HIGH BYTE OF MANTISSA OF FAC
$00A0~$00BF (160~191) [NOUNSTKC] INTEGER BASIC MEMORY LOCATION 'NOUNSTKC' (NOUN STACK COUNTER)
$00A0~$00A1 (160~161) [FACMO~FACLO] \P2\POINTER TO STRING DESCRIPTOR USED IN STRING UTILITIES
$00A0 (160) [FACMO] \P1\        MIDDLE ORDER BYTE OF MANTISSA OF FAC
$00A1 (161) [FACLO] \P1\        LOW ORDER BYTE OF MANTISSA OF FAC
$00A2 (162) [(FACSIGN)] \P1\    SINGLE BYTE SIGN OF FAC. WHILE IN MATH PKG SIGN IS KEPT IN SGN WHERE ONLY BIT 7 IS
                              SIGNIFICANT
$00A4 (164)                     GENERAL USE IN FLOATING POINT MATH ROUTINES
$00A5~$00AA (165~170) [ARG] \PB\APPLESOFT SECONDARY FLOATING POINT ACCUMULATOR (USES 6-BYTE UNPACKED MATH PACKAGE
                              FORMAT DESCRIBED BELOW)
$00A5 (165) [ARGEXP] \P1\       EXPONENT PART OF ARG. SINGLE BYTE SIGNED NUMBER IN EXCESS $80 FORM (SIGNED VALUE HAS
                              $80 ADDED TO IT)
$00A6~$00A9 (166~169) \P4\      FOUR BYTE MANTISSA PART OF ARG. BINARY POINT ASSUMED TO RIGHT OF MSB. NAMES OF BYTES IN
                              MATH PACKAGE HO~MOH~MO~LO RESPECITVELY.
$00A8~$00C7 (168~199) [TXTNDXSTK] INTEGER BASIC MEMORY LOCATION 'TXTNDXSTK' (TEXT INDEX STACK)
$00AA (170) \P1\                SIGN BYTE OF ARG (UNPACKED FORMAT). BYTE NAMED SGN
$00AB~$00AC (171~172) [STRNG1] \P2\APPLESOFT POINTER TO A STRING USED IN 'MOVINS' STRING UTILITY
$00AC~$00AE (172~174) \PB\      APPLESOFT GENERAL USAGE FLAGS/POINTERS
$00AD~$00AE (173~174) [STRNG2] \P2\APPLESOFT POINTER TO A STRING USED IN STRLT2 STRING UTILITY
$00AF~$00B0 (175~176)           PAGE ZERO LOCATIONS USED BY DOS
$00AF~$00B0 (175~176) [PRGEND] \P2\APPLESOFT POINTER TO END OF PROGRAM. NOT CHANGED BY LOMEM:
$00B1~$00C8 (177~200) [CHRGET] \SB\APPLESOFT CHRGET ROUTINE. CALLED WHEN WANTS ANOTHER CHARACTER (X- Y-REGS NOT ALTERED)
$00B1 (177) [CHRGET] \SE\       APPLESOFT CHRGET S/R CALL - GETS NEXT SEQUENTIAL CHR OR TOKEN - LOADS A-REG FROM LOCN
                              SPECIFIED BY TXTPTR($00B8~$00B9 & INCREMENTS TXTPTR. CARRY IS RESET TO ZERO IF
                              CHARACTER IS A DIGIT OTHERWISE IT IS SET; ZERO FLAT SET IF CHAR =0 (END OF LINE SIGN)
                              OR $3A (END OF STATEMENT SIGN ':') OTHERWISE RESET (X- Y-REGS NOT ALTERED)
$00B7 (183) [CHRGOT] \SE\       APPLESOFT CHRGOT S/R CALL. CHRGET INCREMENTS TXTPTR. CHRGOT DOES NOT
$00B8~$00B9 (184~185) [(LAST CHAR PTR)] \P2\APPLESOFT PTR TO LAST CHAR OBTAINED THRU CHRGET ROUTINE
$00B8~$00B9 (184~185) [TXTPTR] \P2\TXTPTR - POINTS AT NEXT CHAR OR TOKEN FROM PROG (C/A DEC 78)
$00C8 (200) [OUTVAL]            INTEGER BASIC MEMORY LOCATION 'OUTVAL' (OUTPUT VALUE TEMPORARY)
$00C8 (200) [TXTNDX]            INTEGER BASIC MEMORY LOCATION 'TXTNDX' (TEXT INDEX VALUE)
$00C9~$00CD (201~205) [RND] \P5\APPLESOFT FLOATING POINT RANDOM NUMBER (5-BYTE FLOATING POINT PACKED FORMAT C9=EXP
                              CA-CD=MANTISSA)
$00C9 (201) [LEADBL]            INTEGER BASIC MEMORY LOCATION 'LEADBL' (LEADING BLANKS INDEX)
$00C9 (201) [YTEMP]             INTEGER BASIC MEMORY LOCATION 'YTEMP' (TEMPORARY STORAGE FOR Y-REGISTER)
$00CA~$00CD (202~205) \PB\      PAGE ZERO LOCATIONS USED BY DOS
$00CA~$00CB (202~203) [PPL~PPH] \P2\INTEGER BASIC PROGRAM POINTER (START-OF-PROGRAM EQUAL TO HIMEM IF NO PROGRAM)
$00CC~$00CD (204~205) [PVL~PVH] \P2\INTEGER BASIC CURRENT VARIABLE POINTER (END OF CURRENT VARIABLE EQUAL TO LOMEM IF
                              NO ACTIVE CURRENT VARIABLE )
$00CE~$00CF (206~207) [ACL~ACH] \P2\INTEGER BASIC MAIN ACCUMULATOR
$00CE~$00CF (206~207) [~VALGETL~VALGETH~] \P2\INTEGER BASIC PRIMARY EVALUATOR TEMPORARY LOCATION
-------------------------------------------------------------------------------------------------------------------
$0098 - $00CE                    Prof. Luebbert's "What's Where in the Apple"                    NUMERIC ATLAS
```

```
$00CE~$00CF (206~207) [~VALL~VALH~] \P2\INTEGER BASIC 16-BIT TEMPORARY VALUE FOR MATHEMATICAL OPERATIONS
$00D0~$00DF (208~223) \PB\       ONERR POINTERS/SCRATCH
$00D0~$00D1 (208~209) [SRCHL~SRCHH] \P2\INTEGER BASIC MEMORY LOCATION 'SRCHL' (POINTER TO SEARCH VARIABLE TABLE)
$00D0 (208) [ERRFLG] \P1\        ERROR FLAG. ON IF BIT 7 SET (PEEK(216)>127). POKE 0 TO CLEAR.
$00D1~$00F0 (209~240) [TOKNDXSTK] INTEGER BASIC MEMORY LOCATION 'TOKNDXSTK' ('TOKEN INDEX STACK?)
$00D2~$00D3 (210~211) \P2\       IF ONERR GOTO OCCURS CONTAINS ADDRESS OF LINE # OF STMT WHERE ERROR OCCURED
$00D2~$00D3 (210~211) [SRCH2L~SRCH2H] \P2\INTEGER BASIC MEMORY LOCATION 'SRCH2L' (SECOND VARIABLE SEARCH POINTER)
$00D4 (212) [IFSKIP] \P1\        INTEGER BASIC MEMORY LOCATION 'IFSKIP' (IF\THEN FAIL FLAG)
$00D5 (213) [CRFLAG] \P1\        INTEGER BASIC MEMORY LOCATION 'CRFLAG' (CARRIAGE RETURN FLAG)
$00D6 (214) [VERBNOW] \P1\       INTEGER BASIC MEMORY LOCATION 'VERBNOW' (VERB CURRENTLY IN USE)
$00D6 (214) \P1\                 APPLESOFT MYSTERY PARAMETER. IF SET TO $80 MAKES ALL COMMANDS = RUN
$00D7 (215) [PRINOW] \P1\        INTEGER BASIC MEMORY LOCATION 'PRINOW' (PRINT IT NOW FLAG)
$00D8 (216)                      PAGE ZERO LOCATION USED BY DOS (INFO FROM DCT RELATED TO MOTOR-ON TIME-REQUIREMENT?)
$00D8 (216) [ERRFLG] \P1\        APPLESOFT ERROR FLAG: $80 IF ONERR ACTIVE. SET TO 0 TO DISABLE 'ONERR GOTO'
$00D8 (216) [XSAVE] \P1\         INTEGER BASIC MEMORY LOCATION 'XSAVE' (TEMPORARY STORAGE FOR CONTENTS OF X-REGISTER)
$00D9 (217) [RUNMODE] \P1\       USED BY DOS TO TEST FOR DIRECT-DEFERRED MODE USAGE. IF $AAB6 CONTAINS 0 AND BIT 7 OF
                                 THIS LOCATION IS CLEAR DOS ASSUMES DIRECT MODE AND WILL NOT DO OPEN OR OTHER DIRECT
                                 MODE COMMANDS
$00D9 (217) [RUNMODE] \P1\       INTEGER BASIC MEMORY LOCATION 'RUNMODE' USED AS RUN MODE FLAG BYTE
$00DA~$00DB (218~219) [AUXL~AUXH] \P2\INTEGER BASIC MEMORY LOCATIONS 'AUXL~AUXH' (AULILIARY COUNTER)
$00DA~$00DB (218~219) [ERRLIN] \P2\APPLESOFT LINE # WHERE ERROR OCCURED
$00DC~$00DD (220~221) [ERRPOS] \P2\APPLESOFT TEXPTR SAVE FOR HNDLERR SUBROUTINE
$00DC~$00DD (220~221) [PRL~PRH] \P2\INTEGER BASIC MEMORY LOCATIONS 'PRL~PRH' (CURRENT LINE VALUE)
$00DE~$00DF (222~223) [PNL~PNH] \P2\INTEGER BASIC MEMORY LOCATIONS 'PNL~PNH' (CURRENT NOUN POINTER)
$00DE (222) [ERRNUM] \P1\        APPLESOFT - WHEN ERROR OCCURS~ TYPE-OF-ERROR CODE APPEARS HERE - SEE MANUAL FOR CODE
                                 NUMBER MEANINGS
$00DF (223) [ERRSTK] \P1\        APPLESOFT STACK POINTER VALUE BEFORE ERROR OCCURED
$00E0~$00E1 (224~225) [PXL~PXH] \P2\INTEGER BASIC MEMORY LOCATIONS 'PXL~PXH' (CURRENT VERB POINTER)
$00E0~$00E1 (224~225) \P2\       HIGH-RES GRAPHICS X-COORDINATE
$00E2 (226) \P1\                 HIGH-RES GRAPHICS Y-COORDINATE
$0032~$00E3 (50~227) [P1L~P1H] \P2\INTEGER BASIC MEMORY LOCATIONS 'P1L~P1H' (AUXILIARY POINTER ONE)
$00E2~$00E3 (226~227) [DELL~DELH] \P2\INTEGER BASIC MEMORY LOCATIONS 'DELL~DELH' (DELETE LINE POINTER)
$00E4~$00E5 (228~229) [LNAL~LNAH] \P2\INTEGER BASIC MEMORY LOCATIONS 'LNAL~LNAH' (LINE NUMBER ADDRESS)(NEXT LINE NUMBER)
$00E4~$00E5 (228~229) [P2L~P2H] \P2\INTEGER BASIC MEMORY LOCATIONS 'P2L~P2H' (AUXILIARY POINTER TWO)
$00E4 (228) \P1\                 HI-RES GRAPHICS COLOR BYTE
$00E4 (228) [FLAG]               INTEGER BASIC MEMORY LOCATION 'FLAG' (GENERAL FLAG BYTE)
$00E5~$00E7 (229~231) \PB\       GENERAL USAGE FOR HI-RES GRAPHICS
$00E5 (229) \P1\                 HI-RES GRAPHICS HORIZONTAL BYTE INDEX FOR CURRENT POSITION(?)
$00E6~$00E7 (230~231) [NXTL~NXTH] \P2\INTEGER BASIC MEMORY LOCATIONS 'NXTL~NXTH' (NEXT POINTER)
$00E6~$00E7 (230~231) [P3L~P3H] \P2\INTEGER BASIC MEMORY LOCATIONS 'P3L~P3H' (AUXILIARY POINTER THREE)
$00E6 (230) [HPAG] \P1\          HI-RES PAGE TO PLOT ON REGARDLESS OF WHICH PAGE BEING DISPLAYED - $20 FOR PG1; $40 FOR
                                 PG2
$00E7 (231) [SCALE] \P1\         HI-RES GRAPHICS SCALE FACTOR
$00E8~$00E9 (232~233) \P2\       HI-RES GRAPHICS POINTER TO BEGINNING OF SHAPE TABLE
$00EA (234) \P1\                 COLLISION COUNTER FOR HI-RES GRAPHICS
$00F0~$00F3 (240~243) \PB\       GENERAL USE FLAGS
$00F0 (240) [FIRST] \P1\         APPLESOFT - USED BY UTILITY PLOTFNS FOR DESTINATION OF FIRST NUMBER OF LO-RES PLOT
                                 COORDINATES
$00F1 (241) [SPDBYT] \P1\        USED FOR SPEED CONTROL OF OUTPUT & DISPLAY. SPEED 0-255 ($00-$FF) CONTROLS INSERTED
                                 DELAY)
$00F1 (241) [TOKNDX] \P1\        INTEGER BASIC MEMORY LOCATION 'TOKNDX' (TOKEN INDEX VALUE)
```

$00F2~$00F3 (242~243) [CONL~CONH] \P2\INTEGER BASIC MEMORY LOCATIONS 'CONL~CONH' (CONTINUE POINTER)
$00F3 (243) [ORMASK] \P1\ MASK FOR OUTPUT CONTROL: NORMAL/FLASHING/INVERSE
$00F3 (243) [SIGN] \P1\ MONITOR & FLOATING POINT ROUTINES MEMORY LOC 'SIGN'
$00F4~$00F8 (244~248) \PB\ ONERR POINTERS
$00F4~$00F7 (244~247) [FP1] \P4\MONITOR & FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR 2 (CONTAINS X2 & M2)
$00F4~$00F5 (244~245) [AUTOINCL~AUTOINCH] \P2\INTEGER BASIC MEMORY LOCATIONS 'AUTOINCL~AUTOINCH' (CURRENT AUTO LINE NUMBER VALUE)
$00F4 (244) [X2] \P1\ MONITOR & OLD (NON-APPLESOFT) FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR 2 MEMORY LOC 'X2' (EXPONENT)
$00F5~$00F7 (245~247) [M2] \P3\ MONITOR & OLD (NON-APPLESOFT) FLOATING POINT ACCUMULATOR 2 MEMORY LOC 'M2' (MANTISSA - 3 BYTES)
$00F6~$00F7 (246~247) [AUTOLNL~AUTOLNH] \P2\INTEGER BASIC MEMORY LOCATIONS 'AUTOLNL~AUTOLNH'
$00F7 (247) [S16PAG] \P\ SWEET-16 MEMORY LOCATION 'S16PAG'
$00F8~$00FE (248~254) [FP1] \P6\OLD (NON-APPLESOFT) FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR FP1 (CONTAINS X1 M1 AND E (EXTENSION))
$00F8 (248) [AUTOMODE] \P1\ INTEGER BASIC MEMORY LOCATION 'AUTOMODE' (THE AUTOMODE FLAG)
$00F8 (248) [X1] OLD (NON-APPLESOFT) FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR FP1 MEMORY LOC 'X1' (EXPONENT)
$00F8 (248) [REMSTK] \P1\ APPLESOFT STACK POINTER SAVED BEFORE EACH STATEMENT
$00F9~$00FB (249~251) [M1] \P3\ FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR FP1 MEMORY LOC 'M1' (MANTISSA)
$00F9 (249) [CHAR] \P1\ INTEGER BASIC MEMORY LOCATION 'CHAR' (CURRENT CHARACTER)
$00F9 (249) [COUNT] \P1\ INTEGER BASIC MEMORY LOCATION 'COUNT'
$00FA (250) [LEAD2R] \P1\ INTEGER BASIC MEMORY LOCATION 'LEADZR' (LEADING ZEROS INDEX)
$00FB (251) [FORNDX] \P1\ INTEGER BASIC MEMORY LOCATION 'FORNDX' (FOR-NEXT LOOP INDEX)
$00FC~$00FE (252~254) [E] \P3\ MONITOR & FLOATING POINT ROUTINES MEMORY LOC 'E' (3 BYTE MANTISSA EXTENTION OF FP ACCUMULATOR 1)
$00FC (252) [GOSUBNDX] \P1\ INTEGER BASIC MEMORY LOCATION 'GOSUBNDX' (GOSUB INDEX)
$00FD (253) [SYNSTKDX] \P1\ INTEGER BASIC MEMORY LOCATION 'SYNSTKDX' (SYNTAX STACK INDEX VALUE)
$00FE~$00FF (254~255) [SYNPAGL~SYNPAGH] INTEGER BASIC SYNTAX PAGE POINTER. IF $00FF NOT ZERO THEN ERROR CONDITION EXISTS
$0100~$01FF (256~511) \HB\ APPLE SYSTEM STACK. MANY USES INCLUDING SUBROUTINE RETURN STACK
$0100~$01FF (256~511) THIS PAGE IS THE STACK USED BY DOS 3.2 TO GET THE SLOTNUMBER IN WHICH THE BOOT DISK IS LOCATED
$0100~$0110 (256~272) [FOUT] \PB\FOUT BUFFER
$0200~$02FF (512~767) [BUF INBUFF] \HB\KEYIN (CHARACTER INPUT) BUFFER (MONITOR~INTEGER BASIC~APPLESOFT BASIC)
$0200 (512) [IN] MONITOR & MINIASSEMBLER MEMORY LOCATION 'IN'
$0300~$03FF (768~1023) \HB\ MEMORY PAGE 3 (MONITOR VECTOR MEMORY PAGE - BUT MONITOR VECTORS ONLY IN $03F0~$0EFF)
$0300~$03E7 (768~999) \HB\ BLOCK OFTEN AVAILABLE AS FREE SPACE FOR USER PROGRAMS. NOTE CONSTRAINTS & COMPETING USES
$0300~$03FF (768~1023) \HB\ FIRST STAGE OF DOS 3.2 BOOT USES THIS AREA FOR PART 1 OF THE NIBBLE BUFFER. THEN LATER DOS 3.2 BOOT USES IT FOR CODE. AREA CLOBBERED BY EITHER MASTER OR SLAVE DISKETTE BOOT. READS DATA FROM TRACK 0; SECTORS 0-9 INTO MEMORY AT $B600~$BFFF (48K MACHINE) SLAVE DISKETTE OR $3600~$3FFF FOR A MASTER DISKETTE
$0300~$03AF (768~943) \SB\ EXAMPLE: DECWRITER PRINTER OUTPUT FOR SERIAL COMMUNICATIONS CARD (BLOADED FROM DISK)
$0301 (769) \SE\ DOS 3.2 BOOT PROCESS JUMPS HERE AFTER ROM BOOT IS FINISHED. TRANSFER (ENTRY) POINT FOR ENTRY TO PART 2 OR STAGE 2 OF DOS 3.2 BOOT. READS IN RWTS & ITS SUBPROGRAMS
$0320~$0321 (800~8C1) [XOL~XOH] \P2\HI-RES GRAPHICS- PRIOR X-COORD SAVE AFTER HLIN OR HPLOT
$0322 (802) [YO] \P1\ HI-RES GRAPHICS YO - MOST RECENT Y-COORDINATE
$0323 (803) [BXSAV] HI-RES GRAPHICS 'BXSAV'
$0324 (804) [HCOLOR] \P1\ HI-RES GRAPHICS COLOR FOR HPLOT~ HPOSN
$0325 (805) [HNDX] \P1\ HI-RES ON-THE-FLY BYTE INDEX FROM BASE ADDRESS TO CURRENT PLOT BYTE (FUNCTION OF CURRENT X-COORD)

```
$0326 (806) [HPAG] \P1\          HI-ORDER BYTE OF START ADDR OF CURRENT HI-RES DISPLAY MEM PG (POKE 32 FOR HI-RES PG1 ~
                                 64 FOR PG2)
$0326 (806) [HPAG] \P1\          HI-RES GRAPHICS MEM PAGE FOR PLOTTING GRAPHICS $20 FOR PG1 ~$40 FOR PG2
$0327 (807) [SCALE] \P1\         ON-THE-FLY SCALE FACTOR FOR DRAW~ SHAPE~ MOVE
$0328~$0329 (808~809) [SHAPXL~SHAPXH] \P2\START-OF-SHAPE-TABLE POINTER
$032A (810) [COLLSN] \P1\        COLLISION COUNT FROM DRAW~DRAW1
$0399 (921)                      DOS 3.2 OFFSET IN THE 1ST NIBBLE BUFFER USED IN RECONSTRUCTING THE REAL DATA
$03CC (972)                      DOS 3.2 OFFSET IN THE FIRST NIBBLE BUFFER USED IN RECONSTRUCTING THE REAL DATA
$03D0~$03E0 (976~992) \SB\       BLOCK OF COMMANDS ETC. COPIED FROM $9E50~$9E80 ON DOS 3.2 BOOT TO CONTROL TRANSFERS TO
                                 SOFT ENTRY~ HARD ENTRY~ I-O PKG~ RWTS AND TO GET END OF SYSTEM BUFFER~ IOB ADDRESS~
                                 AND TO UPDATE I-O HOOKS~ AND DO JUMP TRANSFERS TO AUTO BRK ENTRY~ CTRL~Y ENTRY~ NMI
                                 ENTRY AND PROVIDE IRQ ADDRESS
$03D0 (976) [(3D0G)] \SE\        DOS 3.2 SOFT-ENTRY POINT; I.E. RE-ENTRY POINT (3D0G) FOR RE-INITIALIZATION SAVING ALL
                                 VARIABLES & DATA OF CURRENT BASIC PROGRAM (JMP $9DBF)
$03D3 (979) \SE\                 DOS 3.1/3.2 HARD ENTRY POINT; I.E. RE-INITIALIZATION DESTROYS ALL INFORMATION RELATING
                                 TO CURRENT BASIC PROGRAM (JMP $9D84)
$03D6 (982) \SE\                 DOS 3.1~3.2 ENTRY POINT FOR I~O PACKAGE (JMP $AAFD)
$03D9 (985) \SE\                 DOS 3.1~3.2 ENTRY POINT FOR RWTS (JMP $B7B5)
$03DC (988) \SE\                 DOS 3.1~3.2 ENTRY POINT TO LOAD Y~A WITH ADDRESS AT END OF SYS BUFFER
$03E3 (995) [995] \SE\           DOS 3.1~3.2 ENTRY POINT TO LOAD Y~A WITH ADDRESS OF IOBLK
$03EA (1002) [1002] \SE\         DOS 3.2 ENTRY POINT FOR ROUTINE THAT UPDATES I/O HOOK TABLES IN $0036~$0039. (JMP
                                 $A851 - SAVES ADDRESSES OF CHARACTER INPUT & OUTPUT ROUTINES CURRENTLY IN USE AND
                                 RECONNECTS DOS I/O)
$03EA (1002) [(LOAD DOS 3.2 REGS)] \SE\RECONNECT DOS 3.2 VIA APPLE MONITOR REGS. PREVIOUS CONTENTS OF MONITOR I/O REGS
                                 ($0036~$0039) TO DOS 3.2 INPUT & OUTPUT REGS (DOS 3.2 REGS ALTERED)
$03F0~$03F1 (1008~1009) [BRKV] \P2\AUTOSTART ROM BREAK VECTOR - DEFAULT VALUE $FA59
$03F2~$03F3 (1010~1011) [SOFTEV] \P2\AUTOSTART ROM RESET VECTOR USED FOR SOFT ENTRY TO LANGUAGE IN USE - DEFAULT VALUE
                                 $E003 FOR APPLESOFT
$03F4 (1012) [PWREDUP] \P1\      AUTOSTART ROM POWER UP MASK. SET BY SETPWRC TO EXCLUSIVE 'OR' OF $03F3 & $00A5
$03F5~$03F7 (1013~1015) [AMPERV]  APPLESOFT - HOLDS JMP (JUMP) INSTRUCTION TO S/R WHICH HANDLES & COMMANDS.  DEFAULT
                                 $4C $58 $FF (JUMP TO $FF58)
$03F8~$03FA (1016~1018)          HOLDS JMP (JUMP) INSTRUCTION TO S/R WHICH HANDLES 'USER' COMMANDS (E.G. CTRL-Y)
$03F8 (1016) [USRADR]            IN MONITOR MODE KEYBOARD ENTRY OF CTL-Y WILL CAUSE JSR HERE
$03FB~$03FD (1019~1021)          HOLDS JMP (JUMP) INSTRUCTION TO S/R WHICH HANDLES NON-MASKABLE INTERRUPTS
$03FB (1019) [NMI]               NMI'S VECTORED TO THIS LOCATION
$03FE~$03FF (1022~1023) [IRQADR~IRQLOC] \P2\IRQ'S VECTORED BY POINTER HERE TO SUBROUTINE TO HANDLE INTERRUPT REQUESTS
$0400~$07FF (1024~2047) \HB\     SCREEN BUFFER (MEMORY PAGES 4-7)(LOW-RES GRAPHICS & TEXT PAGE 1); CONSISTS OF 8
                                 SUBPAGES: EACH CONTAINING 3 TEXT LINES OF 40 ($27) CHARACTERS EACH FOLLOWED BY 8
                                 BYTES WHICH ARE USED AS INPUT-OUTPUT PARAMETERS - ONE BYTE FOR EACH SLOT (0-7). LINES
                                 ARE INTERLACED DOWN PAGE: I.E. FIRST SUBPAGE CONTAINS LINES 1~9~17 & FIRST BLOCK OF
                                 I-O BYTES; SECOND SUBPAGE CONTAINS LINES 2~10~18 & SECOND BLOCK OF I-O BYTES; THIRD
                                 SUBPAGE CONTAINS LINES 3~11~19 & THIRD BLOCK OF I-O BYTES ETC.
$0400~$0477 (1024~1143) [(MACROLINE0)] \HB\TEXT VIDEO SCREEN DISPLAY PAGE 1 - MACROLINE ORSUBPAGE CONSISTING OF LINES 0
                                 ~ 8 & 16
$0400-$0427[(LO-RESLNS0/1)] \BB\  VIDEO SCREEN BUFFER LO-RES LINES 0 AND 1
$0400-$0427[(TEXTLN0)] \BB\      VIDEO SCREEN BUFFER TEXT LINE 0
$0428-$044F[(LO-RESLNS16/17)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 16 AND 17
$0428-$044F[(TEXTLN8)] \BB\      VIDEO SCREEN BUFFER TEXT LINE 8
$0450-$0477[(LO-RESLNS32/33)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 32 AND 33
$0450-$0477[(TEXTLN16)] \BB\     VIDEO SCREEN BUFFER TEXT LINE 16
$0478+S (1144+S) \P1\            SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #S
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

$0478+S (1144+S) [BRATE] \P1\      EXAMPLE: SERIAL INTERFACE BAUD QUANTUM RATE. $1= 19200 BAJD;$40=300 BAUD
$0478+S (1144+S) [DVOTRK] \P1\     EXAMPLE:'DRVOTRAK'= DISK DRIVE 0 CURRENT TRACK (VALUE = 2*TRACK#); DOS 3.2 PARAMETER
                                   FOR DISK IN SLOT #S
$0478 (1144) [CURTRK] \P1\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) PARAMETER CURRENT TRACK (LAST TRACK 'SEEK'-ED)
$0478~$047F (1144~1151)            SCRATCHPAD BYTES FOR I/O PERIPHERALS - ONE BYTE FOR EACH PERIPHERAL 'SLOT' 0 THRU 7
$0479 (1145) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1
$047A (1146) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$047B (1147) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$047C (1148) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$047D (1149) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$047E (1150) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$047F (1151) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
$0480-$04A7[(LO-RESLNS2/3)] \BB\   VIDEO SCREEN BUFFER LO-RES LINES 2 AND 3
$0480-$04A7[(TEXTLN1)] \BB\        VIDEO SCREEN BUFFER TEXT LINE 1
$0480~$04F7 (1152~1271) [(MACROLINE1)] \HB\TEXT PAGE 1 - MACROLINE OR SUBPAGE CONSISTING OF 3 TEXT LINES OF 40 BYTES
                                   (CHARACTERS) EACH PLUS A BLOCK OF 8 I-O PERIPHERAL BYTES. SUBSEQUENT
                                   MACROLINES WILL BE OMITTED FROM DATABASE
$04A8-$04CF[(LO-RESLNS18/19)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 18 AND 19
$04A8-$04CF[(TEXTLN9)] \BB\        VIDEO SCREEN BUFFER TEXT LINE 9
$04D0-$04F7[(LO-RESLNS34/35)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 34 AND 35
$04D0-$04F7[(TEXTLN17)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 17
$04F8+S (1272+S) \P1\              SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #S
$04F8+S (1272+S) [DRV1TRK] \P1\    EXAMPLE: 'DRV1TRK' = DISK DRIVE 1 CURRENT TRACK (VALUE = 2*TRACK#); DOS 3.2 PARAMETER
                                   FOR DISK IN SLOT #S
$04F8+S (1272+S) [STBITS] \P1\     EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S: CONTAIN NUMBER OF STOP BITS (INCLUDING 1
                                   PARITY BIT)
$04F8 (1272) \P1\                  SCRATCHPAD MEMORY BYTE USED BY DOS 3.2 (SHARED BY ALL PERIPHERAL CARDS)
$04F8~$04FF (1272~1279) \HB\       TEXT PAGE 1 - BLOCK OF 8 I-O PERIPHERAL BYTES ONE FOR EACH SLOT (0-7)
$04F9 (1273) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1
$04FA (1274) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$04FB (1275) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$04FB (1275) [SEEKCNT] \P1\        DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) SEEK COUNTER PARAMETER
$04FC (1276) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$04FD (1277) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$04FE (1278) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$04FF (1279) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
$0500-$0527[(LO-RESLNS4/5)] \BB\   VIDEO SCREEN BUFFER LO-RES LINES 4 AND 5
$0500-$0527[(TEXTLN2)] \BB\        VIDEO SCREEN BUFFER TEXT LINE 2
$0500~$0577 (1280~1399) [(TEXTMACROLINE2)] \HB\TEXTVIDEO DISPLAY - SUBPAGE 2. CONSISTS OF TEXT LINES 2~ 10 & 18
                                   FOLLOWED BY AN 8-BYTE BLOCK FOR I-O PERIPHERALS
$0528-$054F[(LO-RESLNS20/21)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 20 AND 21
$0528-$054F[(TEXTLN10)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 10
$0550-$0577[(LO-RESLNS36/37)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 36 AND 37
$0550-$0577[(TEXTLN18)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 18
$0578+S (1400+S) \P1\              EXAMPLE APPLE PARALLEL PRINTER INTERFACE IN SLOT #S: CARRIAGE WIDTH. E.G. POKE
                                   1400+S~80 FOR 80 COLUMN PRINT WIDTH
$0578+S (1400+S) \P1\              SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT#S
$0578+S (1400+S) [STATUS] \P1\     EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S: PARITY CHECKSUM OPTIONS (SEE MANUAL)
$0578 (1400) \P1\                  SCRATCHPAD MEMORY BYTE USED BY DOS 3.2 (SHARED BY ALL PERIPHERAL CARDS)
$0578~$057F (1400~1407) \HB\       BLOCK OF SCRATCH PAD BYTES FOR PERIPHERALS IN SLOTS 0-7
$0579 (1401) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1

-------------------------------------------------------------------------------------------------------------------------------
$0478 - $0579                    Prof. Luebbert's "What's Where in the Apple"                              NUMERIC ATLAS
```

```
$057A (1402) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$057B (1403) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$057C (1404) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$057D (1405) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$057E (1406) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$057F (1407) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
$0580-$05A7[(LO-RESLNS6/7)] \BB\   VIDEO SCREEN BUFFER LO-RES LINES 6 AND 7
$0580-$05A7[(TEXTLN3)] \BB\        VIDEO SCREEN BUFFER TEXT LINE 3
$05A8-$05CF[(LO-RESLNS22/23)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 22 AND 23
$05A8-$05CF[(TEXTLN11)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 11
$05D0-$05F7[(LO-RESLNS38/39)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 38 AND 39
$05D0-$05F7[(TEXTLN19)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 19
$05F8+S (1528+S) \P1\              EXAMPLE: APPLE PARALLEL PRINTER INTERFACE IN SLOT #S - CHARACTER COUNTER. E.G. POKE
                                   1528+S^0 TO RESET COUNT TO ZERO
$05F8+S (1528+S) \P1\              SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #S
$05F8+S (1528+S) [SLOT] \P1\       DOS READ-WRITE-TRACK-SECTOR (RWTS) 'SLOT' = HOLDS SLOT NUMBER USED
$05F8 (1528) [(BOOT DISK #)] \P1\ CONTAINS SLOT # OF DISK CONTROLLER CARD FROM WHICH ANY ACTIVE DOS 3.2 WAS BOOTED
$05F8 (1528) \P1\                  SCRATCHPAD MEMORY BYTE USED BY DOS 3.2 (SHARED BY ALL PERIPHERAL CARDS)
$05F9 (1529) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1
$05FA (1530) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$05FB (1531) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$05FC (1532) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$05FD (1533) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$05FE (1534) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$05FF (1535) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
$0600-$0627[(LO-RESLNS8/9)] \BB\   VIDEO SCREEN BUFFER LO-RES LINES 8 AND 9
$0600-$0627[(TEXTLN4)] \BB\        VIDEO SCREEN BUFFER TEXT LINE 4
$0628-$064F[(LO-RESLNS24/25)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 24 AND 25
$0628-$064F[(TEXTLN12)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 12
$0678+S (1656+S) \P1\              EXAMPLE: APPLE PARALLEL PRINTER INTERFACE CARD IN SLOT #S - CHARACTER COUNTER. E.G.
                                   POKE 1656+S^0 TO RESET CHARACTER COUNT TO ZERO
$0678+S (1656+S) \P1\              SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #S
$0678+S (1656+S) [BYTE]            EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S INPUT OUTPUT BUFFER
$0678 (1656) \P1\                  SCRATCHPAD MEMORY BYTE USED BY DOS 3.2 (SHARED BY ALL PERIPHERAL CARDS)
$0679 (1657) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1
$067A (1658) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$067B (1659) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$067C (1660) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$067D (1661) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$067E (1662) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$067F (1663) \P1\                  SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
$0680-$06A7[(LO-RESLNS10/11)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 10 AND 11
$0680-$06A7[(TEXTLN5)] \BB\        VIDEO SCREEN BUFFER TEXT LINE 5
$06=0-$0677[(LO-RESLNS40/41)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 40 AND 41
$06=0-$0677[(TEXTLN20)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 20
$06A8-$06CF[(LO-RESLNS26/27)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 26 AND 27
$06A8-$06CF[(TEXTLN13)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 13
$06D0-$06F7[(LO-RESLNS42/43)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 42 AND 43
$06D0-$06F7[(TEXTLN21)] \BB\       VIDEO SCREEN BUFFER TEXT LINE 21
$06F8+S (1784+S) [PWDTH] \P1\      EXAMPLE:APPLE SERIAL INTERFACE CARD IN SLOT #S - PRINTER WIDTH ('PWDTH')
$06F8+S (1784+S) \P1\              EXAMPLE:APPLE PARALLEL PRINTER INTERFACE CARD IN SLOT #S - COMMAND PREFIX. E.G. POKE
                                   1784+S^137 TO USE CTRL-L (ASCII 137) AS COMMAND PREFIX
```

```
$06F8+S (1784+S) \P1\       SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #S
$06F8 (1784) \P1\           SCRATCHPAD MEMORY BYTE USED BY DOS (SHARED BY ALL PERIPHERAL CARDS)
$06F9 (1785) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1
$06FA (1786) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$06FB (1787) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$06FC (1788) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$06FD (1789) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$06FE (1790) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$06FF (1791) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
$0700-$0727[(LO-RESLNS12/13)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 12 AND 13
$0700-$0727[(TEXTLN6)] \BB\VIDEO SCREEN BUFFER TEXT LINE 6
$0728-$074F[(LO-RESLNS28/29)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 28 AND 29
$0728-$074F[(TEXTLN14)] \BB\VIDEO SCREEN BUFFER TEXT LINE 14
$0750-$0777[(LO-RESLNS44/45)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 44 AND 45
$0750-$0777[(TEXTLN22)] \BB\VIDEO SCREEN BUFFER TEXT LINE 22
$0778+S (1912+S) \P1\       EXAMPLE: APPLE COMMUNICATIONS INTERFACE CARD IN SLOT #S - VIDEO ECHO (SEE ACIC MANUAL PAGE
                            17). E.G. POKE 1912+S^0 FOR NO VIDEO ECHO
$0778+S (1912+S) \P1\       EXAMPLE: APPLE PARALLEL PRINTER INTERFACE CARD IN SLOT #S - VIDEO & LINEFEED STATUS (HIGH
                            BIT CONTROLS VIDEO; LOW BIT CONTROLS L.F.). E.G. POKE 1912+S^1 FOR NO^VIDEO LF^ENABLE. POKE
                            1912+S^128 FOR VIDEO^ENABLE NO^LF. (CENTRONICS VERSION OF APPI DOES NOT HAVE LF OPTIONS
                            ACTIVATED)
$0778+S (1912+S) \P1\       SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #S
$0778+S (1912+S) [NBITS] \P1\EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S NUMBER OF DATA BITS PLUS 1 FOR START BIT
$0778 (1912) \P1\           SCRATCHPAD MEMORY BYTE USED BY DOS 3.2 (SHARED BY ALL PERIPHERAL CARDS)
$0779 (1913) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1
$077A (1914) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$077B (1915) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$077C (1916) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$077D (1917) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$077E (1918) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$077F (1919) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
$0780-$07A7[(LO-RESLNS14/15)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 14 AND 15
$0780-$07A7[(TEXTLN7)] \BB\VIDEO SCREEN BUFFER TEXT LINE 7
$07A8-$07CF[(LO-RESLNS30/31)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 30 AND 31
$07A8-$07CF[(TEXTLN15)] \BB\VIDEO SCREEN BUFFER TEXT LINE 15
$07D0-$07F7[(LO-RESLNS46/47)] \BB\VIDEO SCREEN BUFFER LO-RES LINES 46 AND 47
$07D0-$07F7[(TEXTLN23)] \BB\VIDEO SCREEN BUFFER TEXT LINE 23
$07F8+S (2040+S) \P1\       INTERRUPT RETURN MEMORY BYTE FOR PERIPHERAL IN SLOT #S (LOAD WITH $00CS)
$07F8+S (2040+S) [FLAGS] \P1\EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S OPERATION MODE
$07F8+S (2040+S) [STAT] \P1\APPLE COMMUNICATIONS INTERFACE CARD IN SLOT #S - STATUS (SEE ACIC MANUAL PG 17). E.G. POKE
                            2040+S^17
$07F8 (2040) [(SLOT #)]     CONTAINS SLOT NUMBER (IN THE FORMAT $CS) OF THE PERIPHERAL CARD CURRENTLY ACTIVE - PRINT
                            PEEK(2040)-192 YIELDS SLOT # IN DECIMAL FORMAT
$07F8 (2040) \P1\           SCRATCHPAD MEMORY BYTE USED BY DOS 3.2 (SHARED BY ALL PERIPHERAL CARDS)
$07F9 (2041) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #1
$07FA (2042) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #2
$07FB (2043) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #3
$07FC (2044) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #4
$07FD (2045) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #5
$07FE (2046) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #6
$07FF (2047) \P1\           SCRATCHPAD MEMORY BYTE FOR PERIPHERAL IN SLOT #7
```

```
$0800-$C000 (2048-16384)            RANGE OF POSSIBLE SETTINGS FOR HIMEM (DEPENDING UPON MEM SIZE- DOS 3.2 ETC.)
$0800-$3003 (2048-12291)            APPLESOFT - AREA OCCUPIED BY RAM VERSION (AS OPPOSED TO ROM OR LANGUAGE PACK
                                    VERSION) - MOST LOCATIONS IN ATLAS ARE GIVEN AT ROM LOCATIONS. USE OFFSET TO
                                    TRANSFER TO RAM LOCATIONS
$0800-$0BFF (2048-3071) [(LO-RES PAGE 2)] \HB\SECONDARY SCREEN BUFFER (TEXT & LOW-RES GRAPHICS PAGE 2)
$0800-$0BFF (2048-3071) \SB\        NORMAL LOCATION FOR HI-RES SUBROUTINES (INTEGER BASIC)
$0800-$09FF (2048-2559) \HB\        'NIBBLE' BUFFER AREA FOR PART 2 OF DOS 3.2 BOOT. CLOBBERED BY ANY DOS 3.2 BOOT.
$0800-LOMEM                         PROGRAM STORAGE FOR ROM VERSION OF  APPLESOFT
$0800 (2048)                        DEFAULT INTEGER BASIC LOMEM
$0801-$084C (2049-2124) \SB\        DOS 3.3 - PHASE 2 OF BOOT FROM SECTOR ZERO ON TRACK ZERO - FIRST RAM BOOTSTRAP
                                    LOADER (BOOT1). THIS ROUTINE LOADS THE SECOND RAM LOADER; BOOT 2 INCLUDING RWTS;
                                    INTO MEMORY AND JUMPS TO IT. USES $081F FOR SLOT#;$08FE FOR BOOT2 MEM PG;$08FE FOR
                                    BOOT2 LENGTH
$081F (2079) \SE\                   DOS 3.3 - PHASE 2 OF BOOT - FIRST RAM BOOTSTRAP LOADER. GETS SECTOR TO READ. IF
                                    ZERO GOTO $0839. TRANSLATES THEORETICAL SECTOR NUMBER INTO PHYSICAL SECTOR NUMBER
                                    BY INDEXING INTO SKEWING TABLE AT $084D. DECREMENTS $08FF (THEORETICAL SECTOR #).
                                    SETS UP PARAMETERS FOR ROM S/R $C65C ND JUMPS TO IT. (IT RETURNS TO $0801 WHEN
                                    SECTOR READ
$0839 (2105) \SE\                   DOS 3.3 - PHASE 2 OF BOOT - FIRST RAM BOOTSTRAP LOADER (BOOT1). ADJUSTS PAGE NUMBER
                                    AT $08FE TO LOCATE ENTRY POINT OF BOOT2. INITIALIZES MONITOR (TEXT MODE - STD
                                    WINDOW ETC.). GOTO BOOT2 ($3700 FOR A MASTER DISK;$B700 IN ITS FINAL RELOCATED
                                    LOCN)
$08A0 (2208) \SB\                   DURING DOS 3.2 BOOT AREA STARTING HERE HOLDS THE DISK ->NIBBLE TRANSLATE TABLE
$0C00-$1FFF (3072-8191) \HB\        OFTEN FREE SPACE UNLESS RAM\DISK APPLESOFT IN USE)
$0C00 (3072) \HB\                   DEFAULT LOCATION FOR START OF SHAPE TABLE AS SET BY HI-RES SHAPE LOAD S/R
$0C3C (3132) \SE\                   DOS 3.2\APPLESOFT TRANSFER POINT TO RAM APPLESOFT (DISK AS OPPOSED TO ROM OR
                                    LANGUAGE PACK VERSION) USED BY DOS 3.2 FOR SOFT ENTRY
$0CF2 (3314) \SE\                   APPLESOFT - SET (OR RESET) POINTERS & LINKAGES FOR RAM APPLESOFT STORED AT
                                    $0800-$3003 (2048-12291)
$0CF2 (3314) \SE\                   APPLESOFT - TO CNVRT A/S PROG FROM FIRMWARE (ROM OR LANGUAGE CARD) TO RAM (A/S
                                    STORED IN $0800-$3003): LOAD PROG- CALL 3314-LIST-SAVE
$1067 (4199) \SE\                   DOS 3.2\APPLESOFT TRANSFER POINT USED BY DOS 3.2 INTO RAM (DISK AS OPPOSED TO ROM
                                    OR LANGUAGE PACK) VERSION OF APPLESOFT WHEN PROCESSING ERRORS
$1B00-$3FFF (6912-16383) \SB\       THIS REGION OF MEMORY IS CLOBBERED BY A SLAVE DISKETTE BOOT
$1B00-$3FFF (6912-16383) \SB\       TEMP LOCATION OF RAWDOS 3.2 DURING DOS 3.2 BOOT
$1B00-$1CFF (6912-7423) [SB]        TEMPORARY LOCATION OF DOS 3.2 RELOCATION CODE DURING DOS 3.2 BOOT (SB)
$1DBF (7615) \SE\                   ROUTINE TO RECONNECT DOS 3.2 IF PAGE 3 MONITOR LINKAGES OVERWRITTEN (16K APPLE ONLY)
$2000-$3FFF (8192-16383) [(HI-RES P1)] \HB\HI-RES GRAPHICS PAGE 1
$2000-$2027 (8192-8231) [(HIRES P1L000)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #000
$2028-$204F (8232-8271) [(HIRES P1L064)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #064
$2050-$2077 (8272-8311) [(HIRES P1L128)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #128
$2080-$20A7 (8320-8359) [(HIRES P1L008)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #008
$20A8-$20CF (8360-8399) [(HIRES P1L072)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #072
$20D0-$20E7 (8400-8423) [(HIRES P1L136)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #136
$2100-$2127 (8448-8487) [(HIRES P1L016)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #016
$2128-$214F (8488-8527) [(HIRES P1L80)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #80
$2150-$217F (8528-8575) [(HIRES P1L144)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #144
$2180-$21A7 (8576-8615) [(HIRES P1L024)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #024
$21A8-$21CF (8616-8655) [(HIRES P1L088)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #088
$21D0-$21F7 (8656-8695) [(HIRES P1L152)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #152
$2200-$2227 (8704-8743) [(HIRES P1L032)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #032
```

$2228-$224F (8744-8783) [(HIRES P1L096)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #096
$2250-$2277 (8784-8823) [(HIRES P1L160)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #160
$2280-$22A7 (8832-8871) [(HIRES P1L040)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #040
$22A8-$22CF (8872-8911) [(HIRES P1L104)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #104
$22D0-$22F7 (8912-8951) [(HIRES P1L168)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #168
$2300-$2327 (8960-8999) [(HIRES P1L048)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #048
$2328-$234F (9000-9039) [(HIRES P1L112)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #112
$2350-$237F (9040-9087) [(HIRES P1L176)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #176
$2380-$23A7 (9088-9127) [(HIRES P1L056)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #056
$23A8-$23CF (9128-9167) [(HIRES P1L120)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #120
$23D0-$23F7 (9168-9207) [(HIRES P1L184)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #184
$2400-$2427 (9216-9255) [(HIRES P1L001)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #001
$2428-$244F (9256-9295) [(HIRES P1L065)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #065
$2450-$2477 (9296-9335) [(HIRES P1L129)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #129
$2480-$24A7 (9344-9383) [(HIRES P1L009)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #009
$24A8-$24CF (9384-9423) [(HIRES P1L073)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #073
$24D0-$24E7 (9424-9447) [(HIRES P1L137)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #137
$2500-$2527 (9472-9511) [(HIRES P1L017)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #017
$2528-$254F (9512-9551) [(HIRES P1L081)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #081
$2550-$257F (9552-9599) [(HIRES P1L145)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #145
$2580-$25A7 (9600-9639) [(HIRES P1L025)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #025
$25A8-$25CF (9640-9679) [(HIRES P1L089)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #089
$25D0-$25F7 (9680-9719) [(HIRES P1L153)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #153
$2600-$2627 (9728-9767) [(HIRES P1L033)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #033
$2628-$264F (9768-9807) [(HIRES P1L097)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #097
$2650-$2677 (9808-9847) [(HIRES P1L161)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #161
$2680-$26A7 (9856-9895) [(HIRES P1L041)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #041
$26A8-$26CF (9896-9935) [(HIRES P1L105)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #105
$26D0-$26F7 (9936-9975) [(HIRES P1L169)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #169
$2700-$2727 (9984-10023) [(HIRES P1L049)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #049
$2728-$274F (10024-10063) [(HIRES P1L113)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #113
$2750-$277F (10064-10111) [(HIRES P1L177)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #177
$2780-$27A7 (10112-10151) [(HIRES P1L057)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #057
$27A8-$27CF (10152-10191) [(HIRES P1L121)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #121
$27D0-$47F7 (10192-18423) [(HIRES P1L185)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #185
$2800-$2827 (10240-10279) [(HIRES P1L002)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #002
$2828-$284F (10280-10319) [(HIRES P1L066)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #066
$2850-$2877 (10320-10359) [(HIRES P1L130)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #130
$2880-$28A7 (10368-10407) [(HIRES P1L010)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #010
$28A8-$28CF (10408-10447) [(HIRES P1L074)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #074
$28D0-$28E7 (10448-10471) [(HIRES P1L138)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #138
$2900-$2927 (10496-10535) [(HIRES P1L018)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #018
$2928-$294F (10536-10575) [(HIRES P1L082)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #082
$2950-$297F (10576-10623) [(HIRES P1L146)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #146
$2980-$29A7 (10624-10663) [(HIRES P1L026)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #026
$29A8-$29CF (10664-10703) [(HIRES P1L090)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #090
$29D0-$29F7 (10704-10743) [(HIRES P1L154)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #154
$2A00-$2A27 (10752-10791) [(HIRES P1L034)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #034
$2A28-$2A4F (10792-10831) [(HIRES P1L098)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #098
$2A50-$2A77 (10832-10871) [(HIRES P1L162)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #162
$2A80-$2AA7 (10880-10919) [(HIRES P1L042)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #042

```
$2AA8~$2ACF (10920~10959) [(HIRES P1L106)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #106
$2AD0~$2AF7 (10960~10999) [(HIRES P1L170)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #170
$2B00~$2B27 (11008~11047) [(HIRES P1L050)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #050
$2B28~$362F (11048~13871) [(HIRES P1L114)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #114
$2B50~$2B7F (11088~11135) [(HIRES P1L178)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #178
$2B80~$2BA7 (11136~11175) [(HIRES P1L058)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #058
$2BA8~$2BCF (11176~11215) [(HIRES P1L122)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #122
$2BD0~$2BF7 (11216~11255) [(HIRES P1L186)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #186
$2C00~$2C27 (11264~11303) [(HIRES P1L003)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #003
$2C28~$2C4F (11304~11343) [(HIRES P1L067)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #067
$2C50~$2C77 (11344~11383) [(HIRES P1L131)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #131
$2C80~$2CA7 (11392~11431) [(HIRES P1L011)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #011
$2CA8~$2CCF (11432~11471) [(HIRES P1L075)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #075
$2CD0~$2CE7 (11472~11495) [(HIRES P1L139)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #139
$2D00~$2D27 (11520~11559) [(HIRES P1L019)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #019
$2D28~$2D4F (11560~11599) [(HIRES P1L083)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #083
$2D50~$2D7F (11600~11647) [(HIRES P1L147)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #147
$2D80~$2DA7 (11648~11687) [(HIRES P1L027)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #027
$2DA8~$2DCF (11688~11727) [(HIRES P1L091)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #091
$2DD0~$2DF7 (11728~11767) [(HIRES P1L155)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #155
$2E00~$2E27 (11776~11815) [(HIRES P1L035)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #035
$2E28~$2E4F (11816~11855) [(HIRES P1L099)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #099
$2E50~$2E77 (11856~11895) [(HIRES P1L163)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #163
$2E80~$2EA7 (11904~11943) [(HIRES P1L043)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #043
$2EA8~$2ECF (11944~11983) [(HIRES P1L107)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #107
$2ED0~$2EF7 (11984~12023) [(HIRES P1L171)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #171
$2F00~$2F27 (12032~12071) [(HIRES P1L051)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #051
$2F28~$2F4F (12072~12111) [(HIRES P1L115)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #115
$2F50~$2F7F (12112~12159) [(HIRES P1L179)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #179
$2F80~$2FA7 (12160~12199) [(HIRES P1L059)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #059
$2FA8~$2FCF (12200~12239) [(HIRES P1L123)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #123
$2FD0~$2FF7 (12240~12279) [(HIRES P1L187)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #187
$3000-LOMEM\SB\                  APPLESOFT - PROGRAM STORAGE FOR RAM VERSION
$3000~$3027 (12288~12327) [(HIRES P1L004)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #004
$3003 (12291)                    APPLESOFT - DISKETTE APPLESOFT FP SETS LOMEM TO THIS VALUE
$3028~$304F (12328~12367) [(HIRES P1L068)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #068
$3050~$3077 (12368~12407) [(HIRES P1L132)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #132
$3080~$30A7 (12416~12455) [(HIRES P1L012)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #012
$30A8~$30CF (12456~12495) [(HIRES P1L076)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #076
$30D0~$30E7 (12496~12519) [(HIRES P1L140)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #140
$3100~$3127 (12544~12583) [(HIRES P1L020)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #020
$3128~$314F (12584~12623) [(HIRES P1L084)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #084
$3150~$317F (12624~12671) [(HIRES P1L148)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #148
$3180~$31A7 (12672~12711) [(HIRES P1L028)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #028
$31A8~$31CF (12712~12751) [(HIRES P1L092)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #092
$31D0~$31F7 (12752~12791) [(HIRES P1L156)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #156
$3200~$3227 (12800~12839) [(HIRES P1L036)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #036
$3228~$324F (12840~12879) [(HIRES P1L100)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #100
$3250~$3277 (12880~12919) [(HIRES P1L164)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #164
$3280~$32A7 (12928~12967) [(HIRES P1L044)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #044
$32A8~$32CF (12968~13007) [(HIRES P1L108)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #108
```

```
$32D0-$32F7 (13008-13047) [(HIRES P1L172)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #172
$3300-$3327 (13056-13095) [(HIRES P1L045)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #045
$3328-$334F (13096-13135) [(HIRES P1L116)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #116
$3350-$337F (13136-13183) [(HIRES P1L180)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #180
$3380-$33A7 (13184-13223) [(HIRES P1L060)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #060
$33A8-$33CF (13224-13263) [(HIRES P1L124)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #124
$33D0-$33F7 (13264-13303) [(HIRES P1L188)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #188
$3400-$3427 (13312-13351) [(HIRES P1L005)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #005
$3428-$344F (13352-13391) [(HIRES P1L069)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #069
$3450-$3477 (13392-13431) [(HIRES P1L133)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #133
$3480-$34A7 (13440-13479) [(HIRES P1L013)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #013
$34A8-$34CF (13480-13519) [(HIRES P1L077)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #077
$34D0-$34E7 (13520-13543) [(HIRES P1L141)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #141
$3500-$3527 (13568-13607) [(HIRES P1L021)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #021
$3528-$354F (13608-13647) [(HIRES P1L085)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #085
$3550-$357F (13648-13695) [(HIRES P1L149)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #149
$3580-$35A7 (13696-13735) [(HIRES P1L029)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #029
$35A8-$35CF (13736-13775) [(HIRES P1L093)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #093
$35D0-$35F7 (13776-13815) [(HIRES P1L157)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #157
$3600-$3627 (13824-13863) [(HIRES P1L037)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #037
$3628-$364F (13864-13903) [(HIRES P1L101)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #101
$3650-$3677 (13904-13943) [(HIRES P1L165)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #165
$3680-$36A7 (13952-13991) [(HIRES P1L045)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #045
$36A8-$36CF (13992-14031) [(HIRES P1L109)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #109
$36D0-$36F7 (14032-14071) [(HIRES P1L173)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #173
$3700 (14080)                    DOS 3.3 - START OF BOOT2 AREA FOR A MASTER DISK
$3700-$3727 (14080-14119) [(HIRES P1L053)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #053
$3728-$374F (14120-14159) [(HIRES P1L117)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #117
$3750-$377F (14160-14207) [(HIRES P1L181)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #181
$3780-$37A7 (14208-14247) [(HIRES P1L061)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #061
$37A8-$37CF (14248-14287) [(HIRES P1L125)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #125
$37D0-$37F7 (14288-14327) [(HIRES P1L189)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #189
$3800-$3827 (14336-14375) [(HIRES P1L006)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #006
$3828-$384F (14376-14415) [(HIRES P1L070)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #070
$3850-$3877 (14416-14455) [(HIRES P1L134)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #134
$3880-$38A7 (14464-14503) [(HIRES P1L014)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #014
$38A8-$38CF (14504-14543) [(HIRES P1L078)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #078
$38D0-$38E7 (14544-14567) [(HIRES P1L142)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #142
$3900-$3927 (14592-14631) [(HIRES P1L022] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #022
$3928-$394F (14632-14671) [(HIRES P1L086)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #086
$3950-$397F (14672-14719) [(HIRES P1L150)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #150
$3980-$39A7 (14720-14759) [(HIRES P1L030)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #030
$39A8-$39CF (14760-14799) [(HIRES P1L094)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #094
$39D0-$39F7 (14800-14839) [(HIRES P1L158)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #158
$3A00-$3A27 (14848-14887) [(HIRES P1L038)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #038
$3A28-$3A4F (14888-14927) [(HIRES P1L102)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #102
$3A50-$3A77 (14928-14967) [(HIRES P1L166)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #166
$3A80-$3AA7 (14976-15015) [(HIRES P1L046)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #046
$3AA8-$3ACF (15016-15055) [(HIRES P1L110)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #110
$3AD0-$3AF7 (15056-15095) [(HIRES P1L174)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #174
$3B00-$3B27 (15104-15143) [(HIRES P1L054)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #054
```

$3B28~$362F (15144~13871) [(HIRES P1L118)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #118
$3B50~$3B7F (15184~15231) [(HIRES P1L182)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #182
$3B80~$3BA7 (15232~15271) [(HIRES P1L062)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #062
$3BA8~$3BCF (15272~15311) [(HIRES P1L126)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #126
$3BD0~$3BF7 (15312~15351) [(HIRES P1L190)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #190
$3C00~$3C27 (15360~15399) [(HIRES P1L007)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #007
$3C28~$3C4F (15400~15439) [(HIRES P1L071)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #071
$3C50~$3C77 (15440~15479) [(HIRES P1L135)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #135
$3C80~$3CA7 (15488~15527) [(HIRES P1L015)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #015
$3CA8~$3CCF (15528~15567) [(HIRES P1L079)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #079
$3CD0~$3CE7 (15568~15591) [(HIRES P1L143)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #143
$3D00~$3D27 (15616~15655) [(HIRES P1L023)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #023
$3D00~$3E93 (15616~16027) [RWTS] \SB\DOS 3.1/3.2 RWTS SUBROUTINE
$3D00 (15616) [RWTS] \SE\ DOS 3.1/3.2 READ\WRITE A TRACK & SECTOR. UPON ENTRY A- & Y-REGS POINT AT I/O CONTROL BLOCK (IOB)
$3D1E (15646) [STILLON] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL STARTS CODE WHICH SENSES IF MOTOR STILL ON
$3D23 (15651) [NOTSURE] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - AT THIS POINT PROGRAM NOT SURE WHETHER MOTOR IS RUNNING (STABLE LONG ENOUGH)
$3D28~$3D4F (15656~15695) [(HIRES P1L087)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #087
$3D2D (15661) [SAMESLOT] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE TO DETERMINE IF SAME SLOT BEING USED
$3D44 (15684) [PTRMOV] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE TO MOVE OUT ALL POINTERS FROM IOB (IN-OUT-BLOCK) TO ZERO PAGE
$3D50~$3D7F (15696~15743) [(HIRES P1L151)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #151
$3D5E (15710) [OK] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE THAT IT IS OKAY TO CONTINUE
$3D67 (15719) [DRVSEL] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL
$3D7D (15741) [MOTOF] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL - STARTSCODE TO DELAY UNTIL MOTOR UP TO SPEED
$3D7F (15743) [CONWAIT] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL - STARTS CONSTANT WAIT DELAY LOOP RETURN POINT
$3D80~$3DA7 (15744~15783) [(HIRES P1L031)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #031
$3D8A (15754) [TRYTRK] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL - TRY DISK TRACK AS PART OF LOCATING CORRECT SECTOR FOR READ
$3D9B (15771) [TRYTRK2] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL 'TRYTRK2'
$3DA0 (15776) [TRYADR] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK SECTOR) INTERIOR LABEL 'TRYADR'
$3DA8~$3DCF (15784~15823) [(HIRES P1L095)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #095
$3DA8 (15784) [TRYADR2] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK SECTOR) INTERIOR LABEL 'TRYADR2
$3DC1 (15809) [GOCAL] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - GO CALCULATE CORRECT TRACK
$3DC7 (15815) [RDRIGHT] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL WHICH STARTS CODE TO DETERMINE IF ONE IS READING CORRECT TRACK SECTOR AND VOLUME
$3DD0~$3DF7 (15824~15863) [(HIRES P1L159)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #159
$3DDE (15838) [DRVERR] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE FOR CLEANUP WHEN DRIVE ERROR DETECTED
$3DE1 (15841) [JMPTO1] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'JMPTO1'
$3DE2 (15842) [JMPTOERR] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'JMPTOERR' (JUMP TO ERROR HANDING ROUTINE HNDLERR)
$3DF0 (15856) [RTTRK] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL WHICH ASSUMES RIGHT TRACK SELECTED AND BEGINS CHECK OF CORRECT VOLUME NUMBER ON DISKETTE
$3E00~$3E27 (15872~15911) [(HIRES P1L039)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #039

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

$3E06 (15878) [CORRECTVOL] \SL\     DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL WHICH ASSUMES CORRECT VOLUME
                                    HAS BEEN DETECTED AND CHECKS FOR SECTOR SELECTION
$3E15 (15893) [JJTOER] \SL\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'JJTOER'
$3E17 (15895) [CORRECTSECT] \SL\    DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF CODE WHICH ASSUME
                                    SECTOR CORRECTLY CHOSEN AND JUMPS TO APPROPRIATE SUBROUTINE TO READ OR WRITE
$3E27 (15911) [ALLDONE] \SL\        DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'ALLDONE'
$3E28~$3E4F (15912~15951) [(HIRES P1L103)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #103
$3E29 (15913) [HNDLERR] \SL\        DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF ERROR HANDLING
                                    MODULE
$3E32 (15922) [WRIT] \SL\           DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF CODE TO WRITE
                                    NIBBLES TO DISK IF NOT WRITE PROTECTED
$3E3B (15931) [MYSEEK] \SE\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF ROUTINE WHICH
                                    SEEKS TRACK 'N' IN SLOT #X/$10. (IF DRIVENO IS - THEN DRIVE 0;IF DRIVENO IS + THEN
                                    DRIVE 1
$3E4C (15948) [SEEK] \SE\           DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT SOFT ENTRY POINT OF SEEK
                                    SUBROUTINE
$3E50~$3E77 (15952~15991) [(HIRES P1L167)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #167
$3E67 (15975) [ESDFO] \SL\          DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL 'WASDO'
$3E75 (15989) [ISDRVO] \DL\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'ISDRVO'
$3E78 (15992) [GOSEEK] \DL\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'GOSEEK'
$3E7B (15995) [XTOY] \DL\           DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'XTOY'
$3E80~$3EA7 (16000~16039) [(HIRES P1L047)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #047
$3E82 (16002) [SETTRK] \SM\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - CODE SETS THE
                                    SLOT-DEPENDENT TRACK LOCATION
$3E8F (16015) [SETTRK2] \SM\        DOS 3.2 RWTS (READ-WRITE INTERIOR LABEL 'SETTRK2'
$3E9B (16027) [ONDRVO] \DL\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'ONDRVO'
$3E9C~$3FD4 (16028~16340) [DSKFORM] \SB\DOS 3.2 DISK FORMATTER PACKAGE
$3E9C~$3ED9 (16028~16089) [DSKFORM] \SB\DOS 3.2 DISK FORMATTER MODULE TO FILL TRACK WITH SYNC
$3E9C (16028) [DSKFORM] \SE\        DOS 3.2 DISK FORMATTER ENTRY POINT - TURN MOTOR ON & FILL TRACK WITH SYNC
$3EA8~$3ECF (16040~16079) [(HIRES P1L111)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #111
$3EAB (16043) [DSKF2] \SL\          DOS 3.2 DISK FORMATTER LABEL AT POINT WHERE MOTOR IS RUNNING AND ON TRACK 0. BEGINS
                                    CODE WHICH FORMATS THIS TRACK
$3EAE (16046) [TRKFRM] \SL\         DOS 3.2 DISK FORMATTER LABEL AT POINT WHERE TRACK FORMATTING BEGINS
$3EC4 (16068) [WRTRK] \SL\          DOS 3.2 DISK FORMATTER - LABEL AT POINT WHERE WRITE OF FORMATTING INFO ONTO TRACK
                                    BEGINS -- A HIGHLY TIMING-SENSITIVE AREA OF CODE
$3ECA (16074) [CONSYNC] \SL\        DOS 3.2 DISK FORMATTER - LABEL AT POINT WHERE CONSTRUCTION OF SYNC BEGINS
$3ED0~$3EF7 (16080~16119) [(HIRES P1L175)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #175
$3ED6 (16086) [NXTPRT] \SL\         DOS 3.2 DISK FORMATTER - LABEL AT POINT WHERE CHECK IS MADE TO SEE IF TRACK DONE
$3EDA~$3F72 (16090~16242) \SB\      DOS 3.2 DISK FORMATTER BLOCK OF CODE TO DO SECTOR-BY-SECTORFORMATTING ON TRACK
                                    ALREADY FILLED WITH SELF-SYNC
$3EDE (16094) [RGTIM] \SL\          DOS 3.2 DISK FORMATTER INTERIOR LABEL 'RGTIM'
$3EE0 (16096) [FRMWSYNC] \SL\       DOS 3.2 DISK FORMATTER INTERIOR LABEL 'FRMWSYNC'
$3EE2 (16098) [WRIT2] \SL\          DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WRIT2'
$3EE6 (16102) [WRITSF] \SL\         DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WRITSF'
$3EE7 (16103) [WRIT3] \SL\          DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WRIT3'
$3F00~$3F27 (16128~16167) [(HIRES P1L055)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #055
$3F28~$3F4F (16168~16207) [(HIRES P1L119)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #119
$3F40 (16192) [FAKESCT] \SL\        DOS 3.2 DISK FORMATTER INTERIOR LABEL 'FAKESCT' AT BEGINNING OF CODE TO WRITE FAKE
                                    SECTOR
$3F46 (16198) [INTOIT] \SL\         DOS 3.2 DISK FORMATTER INTERIOR LABEL 'INTOIT'
$3F50~$3F7F (16208~16255) [(HIRES P1L183)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #183
-------------------------------------------------------------------------------------------------------------------------------
$3E06 - $3F50                    Prof. Luebbert's "What's Where in the Apple"                    NUMERIC ATLAS
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------

$3F50 (16208) [NXTTRY] \SL\          DOS 3.2 DISK FORMATTER INTERIOR LABEL 'NXTTRY'
$3F80~$3FA7 (16256~16295) [(HIRES P1L063)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #063
$3FC6 (16326) [CHGIT] \SL\           DOS 3.2 DISK FORMATTER INTERIOR LABEL 'CHGIT'
$3F73~$3FD4 (16243~16340) [TRKDON] \SB\DOS 3.2 DISK FORMATTER CHECK TRACK FORMATTING ROUTINE
$EF73 (-4237) [TRKDON] \SE\          DOS 3.2 DISK FORMATTER INTERIOR LABEL AT POINT WHERE TRACK FORMATTING IS DONE AND
                                     CHECKING OF THAT FORMATTING BEGINS
$3F80 (16256) [WLOOP] \SL\           DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF 26 MICROSECOND WAIT LOOP
$3F94 (16276) [NOGOOD] \SL\          DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF CLEAN UP IF NOGOOD CONDITION
                                     DETECTED
$3F9E (16286) [ITSGOOD] \SL\         DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF CONTINUATION IF GOOD
                                     CONDITION DETECTED
$3FA8~$3FCF (16296~16335) [(HIRES P1L127)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #127
$3FB3 (16307) [DRIVERR] \SL\         DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF CLEANUP IF DRIVE ERROR IS
                                     DETECTED
$3FB8 (16312) [DONEDSK] \SL\         DOS 3.2 DISK FORMATTER INTERIOR LABEL AT POINT WHERE DISK IS COMPLETED AND NO
                                     ERRORS HAVE BEEN DETECTED
$3FBB (16315) [WBYTE] \SL\           DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF TIGHT TIMING ROUTINE
$3FCA (16330) [WNIBLA] \SL\          DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WNIBLA'
$3FCB (16331) [WINBLB2] \SL\         DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WINBLB2'
$EFCD (-4147) [WINBLC] \SL\          DOS 3.2 DISK FORMAT INTERIOR LABEL 'WINBLC'
$EFCE (-4146) [WRNIBL] \SL\          DOS 3.2 DISK FORMAT INTERIOR LABEL 'WRNIBL'
$3FD0~$3FF7 (16336~16375) [(HIRES P1L191)] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #191
$4000~$5FFF (16384~24575) [(HI-RES PAGE 2)] \HB\HI-RES GRAPHICS PAGE 2
$4000~$4520 (16384~17696) \PB\       NORMAL LOCATION FOR MANY HI RES TEXT SETS - E.G. KAPOR'S
$4000~$4027 (16384~16423) [(HIRES P2L000)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #000
$4028~$404F (16424~16463) [(HIRES P2L064)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #064
$4050~$4077 (16464~16503) [(HIRES P2L128)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #128
$4080~$40A7 (16512~16551) [(HIRES P2L008)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #008
$40A8~$40CF (16552~16591) [(HIRES P2L072)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #072
$40D0~$40E7 (16592~16615) [(HIRES P2L136)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #136
$4100~$4127 (16640~16679) [(HIRES P2L016)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #016
$4128~$414F (16680~16719) [(HIRES P2L080)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #080
$4150~$417F (16720~16767) [(HIRES P2L144)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #144
$4180~$41A7 (16768~16807) [(HIRES P2L024)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #024
$41A8~$41CF (16808~16847) [(HIRES P2L088)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #088
$41D0~$41F7 (16848~16887) [(HIRES P2L152)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #152
$4200~$4227 (16896~16935) [(HIRES P2L032)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #032
$4228~$424F (16936~16975) [(HIRES P2L096)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #096
$4250~$4277 (16976~17015) [(HIRES P2L160)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #160
$4280~$42A7 (17024~17063) [(HIRES P2L040)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #040
$42A8~$42CF (17064~17103) [(HIRES P2L104)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #104
$42D0~$42F7 (17104~17143) [(HIRES P2L168)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #168
$4300~$4327 (17152~17191) [(HIRES P2L048)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #048
$4328~$434F (17192~17231) [(HIRES P2L112)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #112
$4350~$437F (17232~17279) [(HIRES P2L176)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #176
$4380~$43A7 (17280~17319) [(HIRES P2L056)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #056
$43A8~$43CF (17320~17359) [(HIRES P2L120)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #120
$43D0~$43F7 (17360~17399) [(HIRES P2L184)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #184
$4400~$4427 (17408~17447) [(HIRES P2L001)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #001
$4428~$444F (17448~17487) [(HIRES P2L065)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #065
$4450~$4477 (17488~17527) [(HIRES P2L129)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #129
```

```
$4480-$44A7 (17536-17575) [(HIRES P2L009)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #009
$44A8-$44CF (17576-17615) [(HIRES P2L073)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #073
$44D0-$44E7 (17616-17639) [(HIRES P2L137)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #137
$4500-$4527 (17664-17703) [(HIRES P2L017)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #017
$4528-$454F (17704-17743) [(HIRES P2L081)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #081
$4550-$457F (17744-17791) [(HIRES P2L145)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #145
$4580-$45A7 (17792-17831) [(HIRES P2L025)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #025
$45A8-$45CF (17832-17871) [(HIRES P2L089)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #089
$45D0-$45F7 (17872-17911) [(HIRES P2L153)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #153
$4600-$4627 (17920-17959) [(HIRES P2L033)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #033
$4628-$464F (17960-17999) [(HIRES P2L97)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #97
$4650-$4677 (18000-18039) [(HIRES P2L161)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #161
$4680-$46A7 (18048-18087) [(HIRES P2L041)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #041
$46A8-$46CF (18088-18127) [(HIRES P2L105)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #105
$46D0-$46F7 (18128-18167) [(HIRES P2L169)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #169
$4700-$4727 (18176-18215) [(HIRES P2L049)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #049
$4728-$474F (18216-18255) [(HIRES P2L113)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #113
$4750-$477F (18256-18303) [(HIRES P2L177)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #177
$4780-$47A7 (18304-18343) [(HIRES P2L057)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #057
$47A8-$47CF (18344-18383) [(HIRES P2L121)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #121
$47D0-$47F7 (18384-18423) [(HIRES P2L185)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #185
$4800-$4827 (18432-18471) [(HIRES P2L002)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #002
$4828-$484F (18472-18511) [(HIRES P2L066)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #066
$4850-$4377 (18512-18551) [(HIRES P2L130)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #130
$4880-$48A7 (18560-18599) [(HIRES P2L010)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #010
$48A8-$48CF (18600-18639) [(HIRES P2L074)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #074
$48D0-$48E7 (18640-18663) [(HIRES P2L138)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #138
$4900-$4927 (18688-18727) [(HIRES P2L018)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #018
$4928-$494F (18728-18767) [(HIRES P2L082)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #082
$4950-$497F (18768-18815) [(HIRES P2L146)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #146
$4980-$49A7 (18816-18855) [(HIRES P2L026)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #026
$49A8-$49CF (18856-18895) [(HIRES P2L090)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #090
$49D0-$49F7 (18896-18935) [(HIRES P2L154)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #154
$4A00-$4A27 (18944-18983) [(HIRES P2L034)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #034
$4A28-$4A4F (18984-19023) [(HIRES P2L098)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #098
$4A50-$4A77 (19024-19063) [(HIRES P2L162)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #162
$4A80-$4AA7 (19072-19111) [(HIRES P2L042)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #042
$4AA8-$4ACF (19112-19151) [(HIRES P2L106)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #106
$4AD0-$4AF7 (19152-19191) [(HIRES P2L170)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #170
$4B00-$4B27 (19200-19239) [(HIRES P2L050)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #050
$4B28-$562F (19240-22063) [(HIRES P2L114)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #114
$4B50-$4B7F (19280-19327) [(HIRES P2L178)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #178
$4B80-$4BA7 (19328-19367) [(HIRES P2L058)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #058
$4BA8-$4BCF (19368-19407) [(HIRES P2L122)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #122
$4BD0-$4BF7 (19408-19447) [(HIRES P2L186)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #186
$4C00-$4C27 (19456-19495) [(HIRES P2L003)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #003
$4C28-$4C4F (19496-19535) [(HIRES P2L067)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #067
$4C50-$4C77 (19536-19575) [(HIRES P2L131)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #131
$4C80-$4CA7 (19584-19623) [(HIRES P2L011)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #011
$4CA8-$4CCF (19624-19663) [(HIRES P2L075)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #075
$4CD0-$4CE7 (19664-19687) [(HIRES P2L139)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #139
```

HEX LOCN (DEC LOCN) [NAME] \USE-TYPE\ - DESCRIPTION

$4D00~$4D27 (19712~19751) [(HIRES P2L019)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #019
$4D28~$4D4F (19752~19791) [(HIRES P2L083)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #083
$4D50~$4D7F (19792~19839) [(HIRES P2L147)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #147
$4D80~$4DA7 (19840~19879) [(HIRES P2L027)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #027
$4DA8~$4DCF (19880~19919) [(HIRES P2L091)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #091
$4DD0~$4DF7 (19920~19959) [(HIRES P2L155)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #155
$4E00~$4E27 (19968~20007) [(HIRES P2L035)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #035
$4E28~$4E4F (20008~20047) [(HIRES P2L099)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #099
$4E50~$4E77 (20048~20087) [(HIRES P2L163)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #163
$4E80~$4EA7 (20096~20135) [(HIRES P2L043)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #043
$4EA8~$4ECF (20136~20175) [(HIRES P2L107)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #107
$4ED0~$4EF7 (20176~20215) [(HIRES P2L171)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #171
$4F00~$4F27 (20224~20263) [(HIRES P2L051)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #051
$4F28~$4F4F (20264~20303) [(HIRES P2L115)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #115
$4F50~$4F7F (20304~20351) [(HIRES P2L179)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #179
$4F80~$4FA7 (20352~20391) [(HIRES P2L059)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #059
$4FA8~$4FCF (20392~20431) [(HIRES P2L123)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #123
$4FD0~$4FF7 (20432~20471) [(HIRES P2L187)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #187
$5000~$5027 (20480~20519) [(HIRES P2L004)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #004
$5028~$504F (20520~20559) [(HIRES P2L068)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #068
$5050~$5077 (20560~20599) [(HIRES P2L132)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #132
$5080~$50A7 (20608~20647) [(HIRES P2L012)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #012
$50A8~$50CF (20648~20687) [(HIRES P2L076)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #076
$50D0~$50E7 (20688~20711) [(HIRES P2L140)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #140
$5100~$5127 (20736~20775) [(HIRES P2L020)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #020
$5128~$514F (20776~20815) [(HIRES P2L084)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #084
$5150~$517F (20816~20863) [(HIRES P2L148)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #148
$5180~$51A7 (20864~20903) [(HIRES P2L028)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #028
$51A8~$51CF (20904~20943) [(HIRES P2L092)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #092
$51D0~$51F7 (20944~20983) [(HIRES P2L156)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #156
$5200~$5227 (20992~21031) [(HIRES P2L036)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #036
$5228~$524F (21032~21071) [(HIRES P2L100)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #100
$5250~$5277 (21072~21111) [(HIRES P2L164)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #164
$5280~$52A7 (21120~21159) [(HIRES P2L044)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #044
$52A8~$52CF (21160~21199) [(HIRES P2L108)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #108
$52D0~$52F7 (21200~21239) [(HIRES P2L172)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #172
$5300~$5327 (21248~21287) [(HIRES P2L045)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #045
$5328~$534F (21288~21327) [(HIRES P2L116)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #116
$5350~$537F (21328~21375) [(HIRES P2L180)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #180
$5380~$53A7 (21376~21415) [(HIRES P2L060)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #060
$53A8~$53CF (21416~21455) [(HIRES P2L124)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #124
$53D0~$53F7 (21456~21495) [(HIRES P2L188)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #188
$5400~$5427 (21504~21543) [(HIRES P2L005)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #005
$5428~$544F (21544~21583) [(HIRES P2L069)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #069
$5450~$5477 (21584~21623) [(HIRES P2L133)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #133
$5480~$54A7 (21632~21671) [(HIRES P2L013)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #013
$54A8~$54CF (21672~21711) [(HIRES P2L077)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #077
$54D0~$54E7 (21712~21735) [(HIRES P2L141)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #141
$5500~$5527 (21760~21799) [(HIRES P2L021)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #021
$5528~$554F (21800~21839) [(HIRES P2L085)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #085
$5550~$557F (21840~21887) [(HIRES P2L149)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #149

```
$5580-$55A7 (21888-21927) [(HIRES P2L029)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #029
$55A8-$55CF (21928-21967) [(HIRES P2L093)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #093
$55D0-$55F7 (21968-22007) [(HIRES P2L157)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #157
$5600-$8000 (22016--32768) \SB\      DOS (32K APPLE ONLY) - DISK OPERATING SYSTEM
$5600-$5627 (22016-22055) [(HIRES P2L037)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #037
$5628-$564F (22056-22095) [(HIRES P2L101)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #101
$5650-$5677 (22096-22135) [(HIRES P2L165)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #165
$5680-$56A7 (22144-22183) [(HIRES P2L045)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #045
$56A8-$56CF (22184-22223) [(HIRES P2L109)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #109
$56D0-$56F7 (22224-22263) [(HIRES P2L173)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #173
$5700-$5727 (22272-22311) [(HIRES P2L053)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #053
$5728-$574F (22312-22351) [(HIRES P2L117)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #117
$5750-$577F (22352-22399) [(HIRES P2L181)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #181
$5780-$57A7 (22400-22439) [(HIRES P2L061)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #061
$57A8-$57CF (22440-22479) [(HIRES P2L125)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #125
$57D0-$57F7 (22480-22519) [(HIRES P2L189)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #189
$5800-$5827 (22528-22567) [(HIRES P2L006)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #006
$5828-$584F (22568-22607) [(HIRES P2L070)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #070
$5850-$5877 (22608-22647) [(HIRES P2L134)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #134
$5880-$58A7 (22656-22695) [(HIRES P2L014)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #014
$58A8-$58CF (22696-22735) [(HIRES P2L078)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #078
$58D0-$58E7 (22736-22759) [(HIRES P2L142)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #142
$5900-$5927 (22784-22823) [(HIRES P2L022] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #022
$5928-$594F (22824-22863) [(HIRES P2L086)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #086
$5950-$597F (22864-22911) [(HIRES P2L150)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #150
$5980-$59A7 (22912-22951) [(HIRES P2L030)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #030
$59A8-$59CF (22952-22991) [(HIRES P2L094)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #094
$59D0-$59F7 (22992-23031) [(HIRES P2L158)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #158
$5A00-$5A27 (23040-23079) [(HIRES P2L038)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #038
$5A28-$5A4F (23080-23119) [(HIRES P2L102)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #102
$5A50-$5A77 (23120-23159) [(HIRES P2L166)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #166
$5A80-$5AA7 (23168-23207) [(HIRES P2L046)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #046
$5AA8-$5ACF (23208-23247) [(HIRES P2L110)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #110
$5AD0-$5AF7 (23248-23287) [(HIRES P2L174)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #174
$5B00-$5B27 (23296-23335) [(HIRES P2L054)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #054
$5B28-$562F (23336-22063) [(HIRES P2L118)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #118
$5B50-$5B7F (23376-23423) [(HIRES P2L182)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #182
$5B80-$5BA7 (23424-23463) [(HIRES P2L062)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #062
$5BA8-$5BCF (23464-23503) [(HIRES P2L126)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #126
$5BD0-$5BF7 (23504-23543) [(HIRES P2L190)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #190
$5C00-$5C27 (23552-23591) [(HIRES P2L007)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #007
$5C28-$5C4F (23592-23631) [(HIRES P2L071)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #071
$5C50-$5C77 (23632-23671) [(HIRES P2L135)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #135
$5C80-$5CA7 (23680-23719) [(HIRES P2L015)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #015
$5CA8-$5CCF (23720-23759) [(HIRES P2L079)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #079
$5CD0-$5CE7 (23760-23783) [(HIRES P2L143)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #143
$5D00-$5D27 (23808-23847) [(HIRES P2L023)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #023
$5D28-$5D4F (23848-23887) [(HIRES P2L087)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #087
$5D50-$5D7F (23888-23935) [(HIRES P2L151)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #151
$5D80-$5DA7 (23936-23975) [(HIRES P2L031)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #031
$5DA8-$5DCF (23976-24015) [(HIRES P2L095)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #095
```

```
$5DD0~$5DF7 (24016~24055) [(HIRES P2L159)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #159
$5E00~$5E27 (24064~24103) [(HIRES P2L039)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #039
$5E28~$5E4F (24104~24143) [(HIRES P2L103)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #103
$5E50~$5E77 (24144~24183) [(HIRES P2L167)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #167
$5E80~$5EA7 (24192~24231) [(HIRES P2L047)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #047
$5EA8~$5ECF (24232~24271) [(HIRES P2L111)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #111
$5ED0~$5EF7 (24272~24311) [(HIRES P2L175)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #175
$5F00~$5F27 (24320~24359) [(HIRES P2L055)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #055
$5F28~$5F4F (24360~24399) [(HIRES P2L119)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #119
$5F50~$5F7F (24400~24447) [(HIRES P2L183)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #183
$5F80~$5FA7 (24448~24487) [(HIRES P2L063)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #063
$5FA8~$5FCF (24488~24527) [(HIRES P2L127)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #127
$5FD0~$5FF7 (24528~24567) [(HIRES P2L191)] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #191
$6884 (26756) [(COMMAND TBL)] \PB\   DOS 3.2 COMMAND TABLE (32K APPLE ONLY!)
$6974 (26996) [(DOS 3.2 ERR MSGS)] \PB\DOS 3.2 ERROR MESSAGES (32K APPLE ONLY!)
$6996~$6A53 (27030~27219) \PB\       DOS 3.2 COUT AND OTHER HOOKS (32K APPLE ONLY! -- SEE $A996~$AA53 FOR MORE
                                     DESCRIPTION BASED ON 48K APPLE)
$6A60 (27232) \P2\                   LENGTH OF MOST RECENTLY BLOADED PROGRAM OR DATA (32K APPLE ONLY)
$6A72 (27250) \P2\                   STARTING ADDRESS OF MOST RECENTLY BLOADED PROGRAM OR DATA (32K APPLE ONLY)
$8F57~$91B9 (-28841~-28231) \PB\      SPACE NORMALLY AVAILABLE FOR USER USE. HOWEVER IF DOS MAXFILES >=6 THIS AREA
                                      BECOMES DOS FILE BUFFER #6
$91B9~$940C (-28231~-27636) \PB\      SPACE NORMALLY AVAILABLE FOR USER USE. HOWEVER IF DOS MAXFILES >=5 THIS AREA
                                      BECOMES DOS FILE BUFFER #5
$940D~$95FF (-27635~-27137) \PB\      SPACE NORMALLY AVAILABLE FOR USER USE. HOWEVER IF DOS MAXFILES >=4 THIS AREA
                                      BECOMES DOS FILE BUFFER #4
$95FF (-27137)                        DEFAULT (MAXFILES = 3) END OF USER RAM WHEN DOS ACTIVE (HIMEM=49151)
$9600~$9CF8 (-27136~-25352) \HB\      3 DOS FILE BUFFERS (DEFAULT CASE) - APPLICABLE TO ALL VERSIONS
                                      (3.1~3.2~3.2.1~3.3) 48K
$9600~$9853 (-27136~-26541) \HB\      DOS FILE BUFFER #3. NOTE: THIS IS DEFAULT FIRST BUFFER USED BY DOS. IF MAXFILES>3
                                      ADDITIONAL BUFFERS WILL BE PLACED BELOW $9600 AND HIGHEST NUMBER BUFFER WILL BE
                                      USED DEFAULT FIRST
$9600~$9700 (-27136~-26880) \HB\      DOS FILE BUFFER #3 - SECTION 1: DATA BUFFER. RECEIVES CONTENTS OF CURRENT DATA
                                      SECTOR
$9600 (-27136)                        HIMEM VALUE (+1) SET HERE WHEN USING DOS 3.1~3.2~3.2.1~3.3 OR 3.2 IN DEFAULT CASE
                                      (MAXFILES=3)
$9701~$9800 (-26879~-26624) \PB\      DOS FILE BUFFER #3 - SECTION 2: TRACK & SECTOR BUFFER. RECEIVES THE CURRENT
                                      TRACK+SECTOR LIST (TSL) SECTOR
$9801~$9853 (-26623~-26541) \PB\      DOS FILE BUFFER #3 - FILE NAME & MISC DATA
$9CF8~$9CFF (-25352~-25345) \HB\      7-BYTE VACANT AREA NOT USED BY DOS 3.23.2
$982D (-26579)                        DOS FILE BUFFER #3 - START OF NAME OF FILE
$984B~$984C (-26549~-26548)           DOS FILE BUFFER #3 - ADDRESS OF START OF MISC INFO SECTION (SECTION 3)  ( DEFAULT
                                      CONTENTS =  $9800)
$984D~$984E (-26547~-26546)           DOS FILE BUFFER #3 - ADDRESS OF START OF TRACK & SECTOR SECTION (SECTION 2)
                                      (DEFAULT CONTENTS =  $9700)
$984F~$9850 (-26545~-26544)           DOS FILE BUFFER #3 - ADDRESS OF START OF DATA SECTION (SECTION 1)  (DEFAULT
                                      CONTENTS =  $9600)
$9851~$9852 (-26543~-26542)           DOS FILE BUFFER #3 - ADDRESS OF START OF NAME BUFFER FOR NEXT FILE ($0000 = NO
                                      MORE FILES)
$9853 (-26541)                        START OF DOS (=HIMEM+1) FOR MAXFILES=2
$9853~$9952 (-26541~-26286)           DOS FILE BUFFER #2 - SECTION 1: DATA BUFFER
$9953~$9A52 (-26285~-26030)           DOS FILE BUFFER #2 - SECTION 2: TRACK & SECTOR BUFFER
```

$9A53 (-26029)		DOS FILE BUFFER #2 - SECTION 3: START OF MISCELLANEOUS INFO BUFFER
$9A80 (-25984)		DOS FILE BUFFER #2 - NAME
$9A9E~$9A9F (-25954~-25953)		DOS FILE BUFFER #2 - ADDRESS OF START OF SECTION 3~ MISCELLANEOUS INFO BUFFER ($9A53)
$9AA0~$9AA1 (-25952~-25951)		DOS FILE BUFFER #2 - ADDRESS OF START OF SECTION 2~ TRACK AND SECTOR BUFFER ($9953)
$9AA2~$9AA3 (-25950~-25949)		DOS FILE BUFFER #2 - ADDRESS OF START OF SECTION 1~ DATA BUFFER ($9853)
$9AA4~$9AA5 (-25948~-25947)		DOS FILE BUFFER #2 - ADDRESS OF START OF NAME BUFFER OF NEXT FILE DOWN ($982D)
$9AA6 (-25946)		START OF DOS (=HIMEM+1) WHEN MAXFILES=1
$9AA6~$9BA5 (-25946~-25691)		DOS FILE BUFFER #1 - SECTION 1~ DATA BUFFER
$9BA6~$9CA5 (-25690~-25435)		DOS FILE BUFFER #1 - SECTION 2~ & SECTOR BUFFER
$9CA6 (-25434)		DOS FILE BUFFER #1 - SECTION 3~ START OF MISC INFO BUFFER ($53 BYTES)
$9CD3 (-25389)		DOS FILE BUFFER #1 - NAME
$9CF1~$9CF2 (-25359~-25358)		DOS FILE BUFFER #1 - ADDRESS OF SECTION 3~ OF MISC INFO BUFFER ($9CA6)
$9CF3~$9CF4 (-25357~-25356)		DOS FILE BUFFER #1 - ADDRESS OF START OF SECTION 2~ TRACK & SECTOR BUFFER ($9BA6)
$9CF5~$9CF6 (-25355~-25354)		DOS FILE BUFFER #1 - ADDRESS OF START OF SECTION 1~ DATA BUFFER ($9AA6)
$9CF7~$9CF8 (-25353~-25352)		DOS FILE BUFFER #1 - ADDRESS OF START OF NAME BUFFER OF NEXT FILE DOWN ($9A80)
$9CF9~$9CFF (-25351~-25345)		DOS 3.2 UNUSED
$9D00~$BFFF (-25344~-16385)	\SB\	DOS 3.2/3.3 (NOT INCLUDING ANY BUFFERS)
$9D00~$9D83 (-25344~-25213)	\SB\	DOS 3.2/3.3 ADDRESS TABLE (LIST OF TWO-BYTE ADDRESS CONSTANTS USED BY DOS)
$9D00~$9D01 (-25344~-25343)	\P2\	ADDRESS OF DOS 3.2/3.3 FILE BUFFER #1 AT ITS FILE NAME FIELD ($9CD3)
$9D02~$9D03 (-25342~-25341)	\P2\	ADDRESS OF DOS 3.2/3.3 INPUT CHARACTER (KEYBOARD INTERCEPT) ROUTINE ($9E81)
$9D04~$9D05 (-25340~-25339)	\P2\	ADDRESS OF DOS 3.2/3.3 OUTPUT CHARACTER (VIDEO INTERCEPT) ROUTINE ($9EBD)
$9D06~$9D07 (-25338~-25337)	\P2\	ADDRESS OF DOS 3.2/3.3 FILE NAME FOR BUFFER #1 (PRIMARY FILE NAME) ($AA75)
$9D08~$9D09 (-25336~-25335)	\P2\	ADDRESS OF DOS 3.2/3.3 FILE NAME FOR BUFFER #2 (SECONDARY OR 'RENAME' FILE NAME) ($AA93)
$9D0A~$9D0B (-25334~-25333)	\P2\	ADDRESS POINTS TO PARAMETER SECTION FOR FIRST LEVEL OF DOS 3.2/3.3 - SEE NEXT ITEM FOR FIRST ENTRY IN SECTION
$9D0A~$9D0B (-25334~-25333)	\P2\	ADDRESS OF DOS 3.2/3.3 LENGTH OF LOAD ($AA60)
$9D0C~$9D0D (-25332~-25331)	\P2\	ADDRESS OF DOS 3.2/3.3 LOAD ADDRESS - I.E. BEGINNING OF DOS ($9D00)
$9D0E~$9D0F (-25330~-25329)	\P2\	DOS 3.2/3.3 ADDRESS POINTS TO PARAMETER SECTION FOR FILE MANAGER - I.E. SECOND (I/O ROUTINE) LEVEL OF DOS
$9D0E~$9D0F (-25330~-25329)	\P2\	ADDRESS OF DOS 3.2/3.3 END OF SYSTEM BUFFERS ($B5BB)
$9D10~$9D1C (-25328~-25316)	\SB\	DOS VIDEO (CSWL) INTERCEPT'S STATE HANDLER ADDRESS TABLE; I.E. TABLE OF ADDRESSES USED IN STATE MACHINE THAT ROUTES OUTPUT CHARACTERS. USED FROM $9EC0 TO $9ED0. $AA52 IS USED TO CHOOSE WHICH ONE
$9D10~$9D11 (-25328~-25327)	\P2\	ADDRESS OF DOS 3.2/3.3 STATE MACHINE CONDITION #0 CODE ($9EEB-1)
$9D12~$9D13 (-25326~-25325)	\P2\	ADDRESS OF DOS 3.2/3.3 STATE MACHINE CONDITION #1CODE ($9F12-1)
$9D14~$9D15 (-25324~-25323)	\P2\	ADDRESS OF DOS 3.2/3.3 STATE MACHINE CONDITION #2 CODE ($9F23-1)
$9D16~$9D17 (-25322~-25321)	\P2\	ADDRESS OF DOS 3.2/3.3 STATE MACHINE CONDITION #3 CODE ($9F2F-1)
$9D18~$9D19 (-25320~-25319)	\P2\	ADDRESS OF DOS 3.2/3.3 STATE MACHINE CONDITION #4 CODE ($9F52-1)
$9D1A~$9D1B (-25318~-25317)	\P2\	ADDRESS OF DOS 3.2/3.3 STATE MACHINE CONDITION #5 CODE ($9F61-1)
$9D1C~$9D1D (-25316~-25315)	\P2\	ADDRESS OF DOS 3.2/3.3 STATE MACHINE CONDITION #6 CODE ($9F71-1)
$9D1E~$9D55 (-25314~-25259)	\SB\	DOS 3.2/3.3 COMMAND DECODER TABLE OF SUBROUTINE ADDRESSES (EXPRESSED IN VALUE-1 FORM TO SIMPLIFY CALLING)
$9D1E~$9D1F (-25314~-25313)	\P2\	DOS 3.2/3.3/3.3 ADDRESS-1 OF CODE FOR 'INIT' COMMAND ($A54F-1)
$9D20~$9D21 (-25312~-25311)	\P2\	DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'LOAD' COMMAND ($A413-1)
$9D22~$9D23 (-25310~-25309)	\P2\	DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'SAVE' COMMAND ($A397-1)
$9D24~$9D25 (-25308~-25307)	\P2\	DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'RUN' COMMAND ($A4D1-1)
$9D26~$9D27 (-25306~-25305)	\P2\	DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'CHAIN' COMMAND ($A4F0-1)
$9D28~$9D29 (-25304~-25303)	\P2\	DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'DELETE' COMMAND ($A263-1)

$9D2A~$9D2B (-25302~-25301) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'LOCK' COMMAND ($A271-1)
$9D2C~$9D2D (-25300~-25299) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'UNLOCK' COMMAND ($A275-1)
$9D2E~$9D2F (-25298~-25297) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'CLOSE' COMMAND ($A2EA-1)
$9D30~$9D31 (-25296~-25295) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'READ' COMMAND ($A51B-1)
$9D32~$9D33 (-25294~-25293) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'EXEC' COMMAND ($A5C6-1)
$9D34~$9D35 (-25292~-25291) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'WRITE' COMMAND ($A510-1)
$9D36~$9D37 (-25290~-25289) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'POSITION' COMMAND ($A5DD-1)
$9D38~$9D39 (-25288~-25287) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'OPEN' COMMAND ($A2A3-1)
$9D3A~$9D3B (-25286~-25285) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'APPEND' COMMAND ($A298-1)
$9D3C~$9D3D (-25284~-25283) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'RENAME' COMMAND ($A281-1)
$9D3E~$9D3F (-25282~-25281) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'CATALOG' COMMAND ($A56E-1)
$9D40~$9D41 (-25280~-25279) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'MON' COMMAND ($A233-1)
$9D42~$9D43 (-25278~-25277) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'NOMON' COMMAND ($A23D-1)
$9D44~$9D45 (-25276~-25275) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'PR#' COMMAND ($A229-1)
$9D46~$9D47 (-25274~-25273) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'IN#' COMMAND ($A22E-1)
$9D48~$9D49 (-25272~-25271) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'MAXFILES' COMMAND ($A251-1)
$9D4A~$9D4B (-25270~-25269) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'FP' COMMAND ($A57A-1)
$9D4C~$9D4D (-25268~-25267) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'INT' COMMAND ($A59E-1)
$9D4E~$9D4F (-25266~-25265) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'BSAVE' COMMAND ($A331-1)
$9D50~$9D51 (-25264~-25263) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'BLOAD' COMMAND ($A35D-1)
$9D52~$9D53 (-25262~-25261) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'BRUN' COMMAND ($A38E-1)
$9D54~$9D55 (-25260~-25259) \P2\ DOS 3.2/3.3 ADDRESS-1 OF CODE FOR 'VERIFY' COMMAND ($A27D-1)
$9D56~$9D83 (-25258~-25213) \SB\ FOUR TABLES OF VECTORS USED BY DOS 3.2/3.3 TO INTERFACE WITH THE VARIOUS SUPPORTED LANGUAGES. DOS USES THESE ADDRESSES TO JUMP INTO THE LANGUAGE WHEN RUNNING (OR CHAINING IN THE CASE OF INTEGER BASIC) A NEW PROGRAM OR WHEN PROCESSING ERRORS
$9D56~$9D61 (-25258~-25247) \SB\ TABLE OF VECTORS USED BY DOS 3.2/3.3 TO INTERFACE WITH CURRENT LANGUAGE
$9D56~$9D57 (-25258~-25257) \P2\ DOS 3.2/3.3 CURRENT LANGUAGE ENTRY-VECTOR TO 'CHAIN'
$9D58~$9D59 (-25256~-25255) \P2\ DOS 3.2/3.3 CURRENT LANGUAGE ENTRY-VECTOR TO 'RUN'
$9D5A~$9D5B (-25254~-25253) \P2\ DOS 3.2/3.3 CURRENT LANGUAGE ENTRY-VECTOR TO 'ERROR'
$9D5C~$9D5D (-25252~-25251) \P2\ DOS 3.2/3.3 CURRENT LANGUAGE ENTRY-VECTOR TO 'HARD ENTRY'
$9D5E~$9D5F (-25250~-25249) \P2\ DOS 3.2/3.3 CURRENT LANGUAGE ENTRY-VECTOR TO 'SOFT ENTRY'
$9D60~$9D61 (-25248~-25247) \P2\ DOS 3.2/3.3 CURRENT LANGUAGE ENTRY-VECTOR TO 'RECOMPUTE LINKS' FOR APPROPRIATE LOCATION OF APPLESOFT BASIC (APPLESOFT ONLY)
$9D62~$9D6B (-25246~-25237) \SB\ IMAGE OF THE ENTRY POINT VECTOR FOR INTEGER BASIC; I.E. TABLE OF VECTORS USED BY DOS 3.2/3.3 TO INTERFACE WITH INTEGER BASIC. MOVED INTO $9D56~$9D59 WHEN INTEGER BASIC IS CURRENT LANGUAGE
$9D62~$9D63 (-25246~-25245) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO INTEGER BASIC 'CHAIN' ($E839)
$9D64~$9D65 (-25244~-25243) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO INTEGER BASIC 'RUN' ($A4E5)
$9D66~$9D67 (-25242~-25241) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO INTEGER BASIC 'ERROR' ($E3E3)
$9D68~$9D69 (-25240~-25239) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO INTEGER BASIC - 'CONTROL-B' OR 'COLD' OR 'HARD' ENTRY ($E000)
$9D6A~$9D6B (-25238~-25237) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO INTEGER BASIC 'SOFT ENTRY' ($E003)
$9D6C~$9D6D (-25236~-25235) NOT USED
$9D6C~$9D77 (-25236~-25225) \PB\ DOS 3.2/3.3 - IMAGE OF THE ENTRY POINT VECTOR FOR APPLESOFT (ROM VERSION) I.E. TABLE OF INTERFACE VECTORS MOVED INTO $9D56~$9D61 WHEN ROM APPLESOFT IS CURRENT LANGUAGE
$9D6C~9D6D\P2\ DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (ROM VERSION) 'CHAIN' (REALLY RUN) ($A4FC)
$9D6E~$9D6F (-25234~-25233) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (ROM VERSION) 'RUN' ($A4FC)
$9D70~$9D71 (-25232~-25231) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (ROM VERSION) 'ERROR' ($D865)
$9D72~$9D73 (-25230~-25229) \P2\ DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (ROM VERSION) - 'CONTROL-B' OR 'COLD' OR 'HARD' ENTRY ($E000)

$9D73~$A7DF (-25229~-22561) \SB\	SYSTEM SECTION OF DOS 3.1 (48K APPLE)
$9D74~$9D75 (-25228~-25227) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (ROM VERSION) 'SOFT ENTRY' ($D43C)
$9D76~$9D77 (-25226~-25225) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (ROM VERSION) 'RECOMPUTE LINKS' ($D4F2)
$9D78~$9D83 (-25224~-25213) \PB\	DOS 3.2/3.3 APPLESOFT (RAM OR DISK VERSION) INTERFACE VECTORS (MOVED INTO $9D56~$9D61 WHEN RAM OR DISK APPLESOFT IS CURRENT LANGUAGE)
$9D84~$A883 (-25212~-22397)	DOS 3.2/3.3 (48K) SYSTEM CODE SECTION
$9D84~$9DBE (-25212~-25154) \SB\	DOS 3.2/3.3 COLDSTART ENTRY ROUTINE
$9D84 (-25212) \SE\	LOCATION TO WHICH DOS 3.2/3.3 JUMPS (ON A CTRL-B OR 3D3G) FOR CODE TO IMPLEMENT A HARD ENTRY TO RAM (DISK AS OPPOSED TO ROM OR LANGUAGE PACK) VERSION OF APPLESOFT
$9D78~$9D79 (-25224~-25223) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (RAM OR DISK) 'CHAIN' ($A506)
$9D7A~$9D7B (-25222~-25221) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (RAM OR DISK) 'RUN' ($A506)
$9D7C~$9D7D (-25220~-25219) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (RAM OR DISK) 'ERROR' ($1067)
$9D7E~$9D7F (-25218~-25217) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (RAM OR DISK) 'HARD ENTRY' ($9D84)
$9D80~$9D81 (-25216~-25215) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (RAM OR DISK) 'SOFT ENTRY' ($0C3C)
$9D82~$9D83 (-25214~-25213) \P2\	DOS 3.2/3.3 ENTRY-VECTOR TO APPLESOFT (RAM OR DISK) 'RECOMPUTE LINKS' ($0CF2)
$9D84 (-25212) \SE\	DOS 3.2/3.3 HARD ENTRY POINT. BOOTSTRAP ROUTINE AT $B700 AND $03D3G BOTH JUMP HERE
$9DB9 (-25159) \SE\	INITIALIZE OR RE-INITIALIZE DOS 3.2/3.3 IF PAGE 3 LINKAGES DESTROYED. OBSOLETE (DOS 3.1 ONLY?)
$9DBF-$9DE9\SB\	DOS 3.2/3.3 WARMSTART ENTRY ROUTINE. GETS REMEMBERED BASIC TYPE AND SETS ROM CARD AS NECESSARY CALLING $A5B2
$9DBF (-25153) \SE\	DOS 3.2/3.3 (48K) SOFT ENTRY POINT. $03D0G AND RESET WITH AUTOSTART ROM BOTH JUMP HERE. (RECONNECTS DOS 3.2 IF PAGE 3 MONITOR LINKAGES OVERWRITTEN)
$9DD1 (-25135)	DOS 3.2/3.3 PARAMETER TO REMEMBER WHETHER ENTRY IS COLDSTART OR WARMSTART
$9DEA (-25110) \SE\	DOS 3.2/3.3 (48K) BLOCK OF CODE WHICH INITS DOS BUFFERS AND SETS VECTORS FOR RAM APPLESOFT. RESTORES $03D0~$03FF FROM $9E51~$9E80. CALLED BY KEYIN IF APPLESOFT MUST COME FROM DISK
$9DEA-$9E50\SB\	DOS 3.2/3.3 FIRST ENTRY PROCESSING ROUTINE CALLED BY KEYBOARD INTERCEPT HANDLER WHEN FIRST KEYBOARD INPUT REQUEST MADE BY BASIC AFTER A DOS COLDSTART
$9E4D (-25011) \SE\	ROUTINE WHICH HANDLES DOS 3.1 INPUT HOOK
$9E50~$9E80 (-25008~-24960) \SB\	BLOCK OF COMMANDS ETC. COPIED INTO $03D0~$03E0 ON DOS 3.2/3.3 BOOT TO CONTROL TRANSFERS TO SOFT ENTRY~ HARD ENTRY~ I-O PKG~ RWTS AND TO GET END OF SYSTEM BUFFER~ IOB ADDRESS~ AND TO UPDATE I-O HOOKS~ AND DO JUMP TRANSFERS TO AUTO BRK ENTRY~ CTRL~Y ENTRY~ NMI ENTRY AND PROVIDE IRQ ADDRESS
$9E51~$9E7F (-25007~-24961)	DOS 3.3 IMAGE OF 3-PAGE JUMP VECTOR WHICH ROUTINE AT $9DEA COPIES TO $3D0-$3FF
$9E7E (-24962) \SE\	ROUTINE WHICH HANDLES DOS 3.1 OUTPUT HOOK
$9E81~$9EB9 (-24959~-24903) \SE\	DOS 3.2/3.3 (48K) KEYBOARD INTERCEPT (INPUT CHARACTER) ROUTINE. CALLS $9ED1 AND MAY CALL $9E9E~$A626 AND~OR $9DEA. DOS COMES HERE FOR EVERY BASIC INPUT STATEMENT OR EVERY LINE TYPED TO THE BASIC PROMPT (E.G.] OR >) OR EVERY TIME PROGRAM USES JSR $FD18 OR $FD0C
$9EBA~$9EBC (-24902~-24900)	DOS 3.2/3.3 JUMP TO THE TRUE KSWL HANDLER ROUTINE
$9EBD~$9EEA (-24899~-24854) \SB\	DOS 3.2/3.3 (48K) DOS COMMAND DECODER -- PART 3. OUTPUT STATE MACHINE AND DEVICE SELECTION CODE
$9EBD~$9ED0 (-24899~-24880)	DOS 3.2/3.3 DOS VIDEO INTERCEPT ROUTINE. CALLS $9ED1 TO SAVE REGISTERS AT ENTRY TO DOS. GETS VIDEO INTERCEPT STATE AND USING IT AS INDEX TO STATE HANDLER TABLE ($9D10) GOES TO PROPER HANDLER ROUTINE & PASSES IT THE CHARACTER TO BE PRINTED
$9EBD (-24899) \SE\	DOS 3.2/3.3 OUTPUT ROUTINE. IF DOS ACTIVE OUTPUT HOOK POINTS HERE & EVERY CHAR TO BE OUTPUTTED PUT INTO ACCUMULATOR FOR DISPOSAL BY CALLING THIS ROUTINE. IT PUSHES ADDRESS FROM STATE MACHINE TABLE ONTO STACK AND THEN RTS'S TO JUMP TO THAT ADDRESS+1
$9ED1-$9EEA	DOS 3.2/3.3 COMMON INTERCEPT SAVE REGISTERS ROUTINE. SAVES A~X~Y AND S-REGISTERS AT $AA59-$AA5C. WHILE IN DOS RESTOR TRUE I/O HANDLERS TO $0036~$0039

$9ED1 (-24879) \SE\	ENTRY POINT TO ABOVE ROUTINE WHICH RESTORES KEYBOARD AND PRINT HOOKS
$9EEB~$9F11 (-24853~-24815) \SB\	DOS 3.2/3.3 STATE 0 OUTPUT HANDLER
$9EEB (-24853) \SE\	DOS 3.2/3.3 STATE MACHINE ENTRY DOS#0 ($AA52=0). DEFAULT VALUE ON DOS ENTRY (SET AT $9DDA) AND ALSO USED AT FRONT OF LINE OUTPUTTED FROM A PROGRAM. CHECKS FOR A VARIETY OF SPECIAL CASES
$9F12~$9F22 (-24814~-24798) \SB\	DOS 3.2/3.3 STATE 1 OUTPUT HANDLER. FUNCTION: COLLECT DOS COMMAND
$9F12 (-24814) \SE\	DOS 3.2/3.3 (48K) STATE MACHINE ENTRY DOS#1 ($AA52=1). USED WHEN OUTPUTTING CTRL-D LINE (DOS COMMAND) FROM PROGRAM SO DOS MUST COLLECT THE LINE FOR DECODING
$9F23~$9F2E (-24797~-24786) \SB\	DOS 3.2/3.3 STATE 2 OUTPUT HANDLER. FUNCTION: NON-DOS COMMAND TO BE IGNORED
$9F23 (-24797) \SE\	DOS 3.2/3.3 (48K) STATE MACHINE ENTRY DOS#2 ($AA52=2). USED FOR OUTPUTTING NORMAL LINE FROM PROGRAM SO DOS MUST ROUTE TO OUTPUT DEVICE
$9F2F (-24785) \SB\	DOS 3.2/3.3 (48K) STATE 3 OUTPUT HANDLER. FUNCTION: INPUT STATEMENT HANDLER
$9F2F (-24785) \SE\	DOS 3.2/3.3 (48K) STATE MACHINE ENTRY DOS#3 ($AA52=3). COME HERE TO OUTPUT A CHARACTER BEING ECHOED FROM THE INPUT ROUTINE (KEYBOARD OR EXEC FILE)
$9F52~$9F60 (-24750~-24736) \SB\	DOS 3.2/3.3 (48K) STATE 4 OUTPUT HANDLER. FUNCTION: WRITE DATA TO A FILE
$9F52 (-24750) \SE\	DOS 3.2/3.3 (48K) STATE MACHINE ENTRY DOS#4 ($AA52=4). STATES DOS#4 & DOS#5 WORK TOGETHER TO OUTPUT TO THE DISK UNTIL A LINE COMES ALONG WITH A CTRL-D ON THE FRONT. DOS#4 - WRITE IS ACTIVE~ MIDDLE OF LINE
$9F61 (-24735) \SB\	DOS 3.2/3.3 (48K) STATE 5 OUTPUT HANDLER. FUNCTION: START OF WRITE DATA LINE
$9F61 (-24735) \SE\	DOS 3.2/3.3 (48K) STATE MACHINE ENTRY DOS#5 ($AA52=5). SEE $9F52 FOR EXPLANATION. DOS#5 - WRITE IS ACTIVE~ FRONT OF LINE
$9F71~$9F77 (-24719~-24713) \SB\	DOS 3.2/3.3 (48K) STATE 6 OUTPUT HANDLER. FUNCTION: SKIP PROMPT CHARACTER. SETS STATE TO 0 AND EXITS VIA $9F9D (ECHO IF MON I)
$9F71 (-24719) \SE\	DOS 3.2/3.3 (48K) STATE MACHINE ENTRY DOS#6 ($AA52=6). CONDITION WHEN ECHOING INPUT FROM 'READ' FILE. DOS IGNORES CHARACTERS FOR DOS COMMAND PURPOSES. USED BY THE EXEC COMMAND
$9F78~$9F82 (-24712~-24702)	DOS 3.2/3.3 (48K) FINISHES RUN COMMAND INTERRUPTED BY APPLESOFT RAM LOAD. RESETS 'RUN INTERRUPTED' FLAG; CALLS $A851 TO REPLACE DOS CSWL/KSWL INTERCEPTS AND GOES TO $A4DC TO COMPLETE THE RUN COMMAND
$9F83~$9F94 (-24701~-24684)	DOS 3.2/3.3 (48K) COMMAND SCANNER EXIT TO BASIC ROUTINE. IF 1ST CHAR OF COMMAND LINE IS CONTROL-D GO TO ECHO EXIT ($9F75); OTHERWISE SET THINGS UP SO BASIC WON'T SEE THE DOS COMMAND (BY PASSING A ZERO-LENGTH LINE I.E. ONLY A CARRIAGE RETURN) AND FALL THRU TO ECHO EXIT
$9F95~$9FB0 (-24683~-24656)	DOS 3.2/3.3 (48K) ROUTINE TO ECHO CHARACTER ON SCREEN (CONDITIONALLY) AND EXIT DOS. ($9F95 ECHO ONLY IF MON C SET; OTHERWISE GOTO $9FBE. $9F99 ECHO ONLY IF MON O SET; OTHERWISE GO TO $9FB3. $9F9D ECHO ONLY IF MON I SET; OTHERWISE GOTO $9FB3. $9FA4 ALWAYS ECHO CHARACTER.) CALLS $9FBA EXIT DOS $9FC5
$9FB3~$9FC4 (-24653~-24636)	DOS 3.2/3.3 (48K) EXIT ROUTINE AND REGISTER RESTORE. CALLS $A851 TO PUT BACK DOS KSWL/CSWL INTERCEPTS. RESTORES S-REGISTER FROM ENTRY TO DOS.
$9FBA (-24646)	DOS 3.2/3.3 (48K) DOS REGISTER RESTORE SUBROUTINE. RESTORES REGISTERS FROM FIRST ENTRY TO DOS AND RETURNS TO CALLER
$9FC5-$9FC7	DOS 3.2/3.3 (48K) JUMP TO THE TRUE CSWL ROUTINE
$9FC8~$9FCC (-24632~-24628)	DOS 3.2/3.3 (48K) SKIP A LINE ON THE SCREEN BY LOADING A CARRIAGE RETURN INTO THE A REGISTER AND CALLING $9FC5 TO PRINT IT
$9FCD~$A179 (-24627~-24199) \SB\	DOS 3.2/3.3 (48K) DOS COMMAND PARSE ROUTINE
$9FCD (-24627)	START OF SECTION OF CODE THAT ATTEMPTS TO MATCH TO A COMMAND AND GET ALL INFO NEEDED & ALL OPERATIONAL INFO GIVEN. CHECKS SYNTAX AND RANGES BEFORE EXECUTION
$A095 (-24427)	DOS 3.2/3.3 (48K) SUBROUTINE TO BLANK BOTH FILENAME BUFFERS
$A0D1 (-24367)	DOS 3.2/3.3 (48K) SETS DEFAULTS FOR THE KEYWORD OPERANDS (V=0~L=0~B=0)
$A0E8 (-24344)	DOS 3.2/3.3 (48K) GET THE LINE OFFSET INDEX AND FLUSH TO THE NEXT NON-BLANK SKIPPING ANY COMMAS FOUND. IF NOT YET TO END OF LINE GOTO $A10C. CHECK TO SEE IF ANY KEYWORDS WERE GIVEN WHICH WERE NOT ALLOWED BY THIS COMMAND

$A10C (-24308)	DOS 3.2/3.3 (48K) LOOKUP THE KEYWORD FOUND ON THE COMMAND LINE IN THE TABLE OF VALID KEYWORDS ($A941). SAVE VALUE OF KEYWORD IN KEYWORD VALUES TABLE STARTING AT $AA66. GO PARSE NEXT KEYWORD. GOTO $A0E8
$A164 (-24220)	DOS 3.2/3.3 (48K) INDICATE C~I OR O KEYWORDS WERE PARSED. UPDATE MONN VALUE IN KEYWORD VALUE TABLE APPROPRIATELY. GOT PARSE THE NEXT KEYWORD. GOTO $A0E8
$A180~$A192 (-24192~-24174)	DOS 3.2/3.3 (48K) DO COMMAND. RESET VIDEO INTERCEPT STATE TO ZERO;CLEAR FILE MANAGER PARAMETER LIST. USING COMMAND INDEX GET ADDRESS OF THE COMMAND HANDLING ROUTINE FROM THE COMMAND HANDLER ROUTINE TABLE AT $9D1E AND GO TO IT. COMMAND HANDLER WILL EXIT TO CALLER OF THIS ROUTINE
$A193~$A1A3 (-24173~-24157)	DOS 3.2/3.3 (48K) GET NEXT CHARACTER ON COMMAND LINE AND CHECK TO SEE IF IT IS A C/R OR A COMMA
$A1A4~$A1AD (-24156~-24147)	DOS 3.2/3.3 (48K) FLUSHES COMMAND LINE CHARACTERS UNTIL A NON-BLANK IS FOUND
$A1AE~$A1B8 (-24146~-24136)	DOS 3.2/3.3 (48K) CLEAR FILE MANAGER PARAMETER LIST AT $B5BB TO ZEROS
$A1B4 (-24140)	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'PR#' COMMAND
$A1B9 (-24135) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'IN#' COMMAND
$A1BE (-24130) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'MON' COMMAND
$A1DC (-24100) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'MAXFILES' COMMAND
$A1EE (-24082) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'DELETE' COMMAND
$A1FC (-24068) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'LOCK' COMMAND
$A200 (-24064) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'BSAVE' COMMAND
$A200 (-24064) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'UNLOCK' COMMAND
$A208 (-24056) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'VERIFY' COMMAND
$A20C (-24052) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'RENAME' COMMAND
$A223 (-24029) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'APPEND' COMMAND
$A229~$A60D (-24023~-23027) \SB\	DOS 3.2/3.3 (48K) - BLOCK OF CODE TO HANDLE INDIVIDUAL DOS COMMANDS
$A229 (-24023) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'PR#' COMMAND
$A22E (-24018) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'IN#' COMMAND
$A233 (-24013) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'MON' COMMAND
$A236 (-24010) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'OPEN' COMMAND
$A23D (-24003) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'NOMON' COMMAND
$A251 (-23983) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'MAXFILES' COMMAND
$A263 (-23965) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'DELETE' COMMAND
$A271 (-23951) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'LOCK' COMMAND
$A275 (-23947) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'UNLOCK' COMMAND
$A278 (-23944) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'CLOSE' COMMAND
$A27D (-23939) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'VERIFY' COMMAND
$A281 (-23935) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'RENAME' COMMAND
$A298 (-23912) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'APPEND' COMMAND
$A2A3 (-23901) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'OPEN' COMMAND
$A2EA (-23830) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'CLOSE' COMMAND
$A2EC (-23828) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'BLOAD' COMMAND
$A327 (-23769) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'BRUN' COMMAND
$A330 (-23760) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'SAVE' COMMAND
$A331 (-23759) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'BSAVE' COMMAND
$A35D (-23715) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'BLOAD' COMMAND
$A38E (-23666) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'BRUN' COMMAND
$A397 (-23657) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'SAVE' COMMAND
$A3A5 (-23643) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'LOAD' COMMAND
$A413 (-23533) \SE\	DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'LOAD' COMMAND
$A476 (-23434) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'RUN' COMMAND
$A48D (-23411) \SE\	DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'CHAIN' COMMAND

$A4A5 (-23387) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'WRITE' COMMAND
$A4B0 (-23376) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'READ' COMMAND
$A4D1 (-23343) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'RUN' COMMAND
$A4E4 (-23324) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'INIT' COMMAND
$A4E5 (-23323) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT TO WHICH DOS TRANSFERS TO RUN A NEW INTEGER BASIC PROGRAM
$A4F0 (-23312) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'CHAIN' COMMAND
$A4FC (-23300) \SE\ DOS 3.2/3.3\APPLESOFT TRANSFER POINT USED BY DOS 3.2 TO JUMP INTO EITHER CHAIN OR RUN OF AN APPLESOFT (ROM) PROGRAM
$A501 (-23295) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'NOMON' COMMAND
$A506 (-23290) \SE\ DOS 3.2/3.3\APPLESOFT TRANSFER POINT USED BY DOS 3.2 TO JUMP INTO EITHER CHAIN OR RUN OF AN APPLESOFT (RAM OR DISK VERSION) PROGRAM
$A50D (-23283) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'FP' COMMAND
$A510 (-23280) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'WRITE' COMMAND
$A51B (-23269) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'READ' COMMAND
$A531 (-23247) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'INT' COMMAND
$A54F (-23217) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'INIT' COMMAND
$A54F (-23217) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'EXEC' COMMAND
$A566 (-23194) \SE\ DOS 3.1 (48K) ENTRY POINT FOR CODE TO IMPLEMENT 'POSITION' COMMAND
$A56E (-23186) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'CATALOG' COMMAND
$A57A (-23174) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'FP' COMMAND
$A59E (-23138) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'INT' COMMAND
$A5C6 (-23098) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'EXEC' COMMAND
$A5DD (-23075) \SE\ DOS 3.2/3.3 (48K) ENTRY POINT OF CODE TO IMPLEMENT 'POSITION' COMMAND
$A60E (-23026) DOS 3.2/3.3 - CODE WHICH STARTS THE READ PROCESS
$A626 (-23002) DOS 3.2/3.3 - CODE WHICH STARTS THE WRITE PROCESS
$A644 (-22972) DOS 3.2/3.3 - CODE WHICH STORES DATA COMING FROM TEXT FILE INTO KEYBOARD BUFFER. USED BY EXEC COMMAND
$A679 (-22919) DOS 3.2/3.3 - CODE TO CLOSE FILES AND EXIT DOS
$A69D (-22883) DOS 3.2/3.3 - CODE TO SET UP ADDRESS OF NAME SECTION OF NEXT FILE
$A6AB (-22869) DOS 3.2/3.3 - CODE TO CLOSE THE BUFFER LAST USED
$A6C4 (-22844) DOS 3.2/3.3 - PRINTS 'SYNTAX ERROR'
$A6C8 (-22840) DOS 3.2/3.3 - PRINT 'NO BUFFERS AVAILABLE'
$A6CC (-22836) DOS 3.2/3.3 - PRINTS 'PROGRAM TOO LARGE'
$A6D0 (-22832) DOS 3.2/3.3 - PRINTS 'FILE TYPE MISMATCH'
$A6D2 (-22830) DOS 3.2/3.3 - START OF ERROR PROCESSING ROUTINE. ENTER WITH ERROR NUMBER IN A-REG
$A6D5 (-22827) DOS 3.2/3.3 - PRINTS OTHER ERROR MESSAGES BY MESSAGE NUMBER CONTAINED IN $AA5C
$A702 (-22782) DOS 3.2/3.3 - ANOTHER PART OF ROUTINE THAT PRINTS APPROPRIATE DOS ERROR MESSAGES (?)
$A743 (-22717) DOS 3.2/3.3 - MOVES NAME FROM THE NAME BUFFER TO THE NAME SECTION OF THE FILE BUFFER
$A764 (-22684) DOS 3.2/3.3 - ATTEMPTS TO FIND A FILE BUFFER ALREADY IN USE BY THE NAME GIVEN
$A7C4 (-22588) DOS 3.2/3.3 - CHECKS FILE TYPE
$A7D4 (-22572) DOS 3.2/3.3 - SETS UP FILE BUFFERS AND ADDRESSES (USED BY MAXFILES)
$A7E0~$A863 (-22560~-22429) [(DOS 3.1 COMMAND TBL)] \PB\DOS 3.1 COMMAND TABLE (DOS 3.1 - 48K APPLE ONLY!)
$A851 (-22447) DOS 3.2/3.3 - RESTORES DOS HOOKS (SAVE ADDRESSES OF CHARACTER INPUT AND OUTPUT ROUTINES CURRENTLY IN USE AND RECONNECT DOS)
$A884~$AAFC (-22396~-21764) DOS 3.2/3.3 (48K) PARAMETER AREA FOR SYSTEM SECTION
$A884~$A908 (-22396~-22264) [(DOS 3.2/3.3 COMMAND TBL)] \PB\DOS 3.2 (48K) COMMAND NAME TABLE OF DOS COMMAND DECODER (TABLE-DRIVEN COMMAND PARSER). CONTAINS NAMES OF DOS COMMANDS WITH LAST BYTE OF EACH NAME HAVING HIGH (7TH) BIT SET; OTHER BYTES HAVE IT CLEAR. THIS PERMITS CLOSE PACKING FOR SEQUENTIAL SEARCH. EOT IS $00 BYTE

$A909~$A970 (-22263~-22160) \PB\ DOS 3.2/3.3 (48K) PARAMETER VALIDITY TABLE OF DOS COMMAND DECODER. USED TO CHECK VALIDITY OF VARIOUS PARAMETERS AGAINST USABILITY WITH VARIOUS COMMANDS. USES 2-BYTE MASKS. ONE BYTE USED TO DETERMINED WHAT TYPE(S) OF EXTRA DATA ARE NEEDED BY A COMMAND; THE OTHER FOR WHAT FILE TYPE TO CREATE OR LOOK FOR

$A941 (-22207) DOS 3.2/3.3 -TABLE CONTAINING THE LETTERS V~D~S~L~R~B~A~C. THESE ARE USED AS SINGLE-CHARACTER KEYWORDS WHICH MAY APPEAR ON DOS COMMANDS. USED WHEN CHECKING FOR THIS OPTIONAL DATA

$A94B~$A954 (-22197~-22188) DOS 3.2/3.3 -TABLE OF BYTES FOR. TABLE CONTAINS OPERAND MASKS ASSOCIATED WITH EACH OPERAND. IF HIGH ORDER BIT IS CLEAR IT INDICATES A NUMERIC ASSOCIATED WITH IT DETERMINING WHAT TYPE OF OPTIONAL DATA TO LOOK FOR. TABLE CONTAINS OPERAND MASKS ASSOCIATED WITH EACH OPERAND. IF HIGH ORDER BIT IS CLEAR IT INDICATES A NUMERIC ASSOCIATED WITH IT

$A955~$A970 (-22187~-22160) DOS 3.2/3.3 -TABLE OF MINIMUM AND MAXIMUM RANGES FOR V~D~S~L~R~B~A

$A971~$AA3E (-22159~-21954) \PB\ DOS 3.2/3.3 (48K) ERROR MESSAGE TABLE (TEXT OF MESSAGES) NOTE: $AA3F~$AA4F IS INDEX TABLE FOR SELECTION OF SPECIFIC MESSAGE FROM THIS BLOCK

$AA3F~$AA4F (-21953~-21937) \PB\ DOS 3.2/3.3 (48K) INDEX TABLE FOR ERROR MESSAGES AT $A971

$A8CD~$A980 (-22323~-22144) [(DOS 3.1 ERROR MSGS)] \PB\DOS 3.1 ERROR MSG TABLE (DOS 3.1 - 48K APPLE ONLY!)

$A971~$AA3E (-22159~-21954) [(DOS 3.2/3.3 ERROR MSGS)] \PB\DOS 3.2/3.3 ERROR MESSAGES (DOS 3.2/3.3 - 48K APPLE ONLY!)

$A996~$A997 (-22122~-22121) [COUT] \P2\DOS 3.1 INTERNAL HOOK ENTRY ADDRESS TO OUTPUT A CHARACTER

$A998~$A999 (-22120~-22119) [CIN] \P2\DOS 3.1 INTERNAL HOOK ENTRY ADDRESS TO INPUT A CHARACTER

$A9A3~$A9A4 (-22109~-22108) \P2\ LENGTH OF MOST RECENTLY BLOADED FILE (DOS 3.1 ONLY - 48K)

$A9B5~$A9B6 (-22091~-22090) \P2\ STARTING ENTRY ADDRESS OF BLOADED FILE (DOS 3.1 ONLY - 48K)

$AA0B (-22005) START OF LIST OF POINTERS TO SECTIONS OF DOS 3.1 I/O PACKAGES

$AA3F~$B2CE (-21953~-19762) \SB\ DOS 3.1 I/O PACKAGE (48K APPLE) (SEE $AAFD FOR CORRESPONDING PKG DOS 3.2~3.2.1~3.3)

$AA42~$AAC8 (-21950~-21816) \PB\ DOS 3.2/3.3 (48K) BLOCK OF IMPORTANT VARIABLES (PARAMETERS)

$AA4F~$AA50 (-21937~-21936) \P2\ DOS 3.2/3.3 (48K) CURRENT FILE BUFFER POINTER

$AA51 (-21935) \P1\ DOS 3.2/3.3 STATE-MACHINE INPUT-STATE CONTROL PARAMETER

$AA52 (-21934) \P1\ DOS 3.2/3.3 (48K) STATE-MACHINE OUTPUT-STATE-CONTROL PARAMETER (0-7)

$AA53~$AA54 (-21933~-21932) \P2\ DOS 3.2/3.3 (48K) OUTPUT HOOK - I.E. ADDRESS OF CHARACTER OUTPUT ROUTINE WHICH WAS IN CONTROL WHEN DOS WAS RECONNECTED (DEFAULT $FDF0)

$AA55~$AA56 (-21931~-21930) \P2\ DOS 3.2/3.3 (48K) INPUT HOOK - I.E. ADDRESS OF CHARACTER INPUT ROUTINE WHICH WAS IN CONTROL WHEN DOS WAS RECONNECTED (DEFAULT $FD1B)

$AA57 (-21929) \P1\ DOS 3.2/3.3 (48K) CURRENT NUMBER OF DOS BUFFERS. DEFAULT=3; CHANGED BY SETTING MAXFILES

$AA59 (-21927) \P1\ DOS 3.2/3.3 (48K) TEMPORARY DOS STORAGE FOR S-REGISTER

$AA5A (-21926) \P1\ DOS 3.2/3.3 (48K) TEMPORARY DOS STORAGE FOR X-REGISTER

$AA5B (-21925) \P1\ DOS 3.2/3.3 (48K) TEMPORARY DOS STORAGE FOR Y-REGISTER

$AA5C (-21924) \P1\ DOS 3.2/3.3 (48K) TEMPORARY DOS STORAGE FOR A-REGISTER

$AA5C (-21924) \P1\ DOS 3.2/3.3 (48K) UPON ENCOUNTERING A DOS ERROR CONTAINS DOS ERROR CODE USED AS INDEX INTO TABLE AT $AA3F OUTPUT OF WHICH IS USED AS INDEX TO ERROR TEXT TABLE AT $A971

$AA5D (-21923) \P1\ DOS 3.2/3.3 (48K) LINE BUFFER INDEX (DISPLACEMENT)

$AA5E (-21922) \P1\ DOS 3.2/3.3 (48K) MON-NOMON STATUS PARAMETERS MASK

$AA5F (-21921) \P1\ DOS 3.2/3.3 (48K) COMMAND NUMBER

$AA60~$AA61 (-21920~-21919) \P2\ DOS 3.2/3.3 (48K) BLOCK LENGTH (FOUND L$ FROM A 'BLOAD')

$AA60~$AA61 (-21920~-21919) \P2\ DOS 3.2/3.3 (48K) - CONTAINS LENGTH OF LOADED BASIC PROGRAM

$AA62 (-21918) \P1\ DOS 3.2/3.3 (48K) STORES COMMAND NUMBER

$AA63 (-21917) \P1\ DOS 3.2/3.3 (48K) TEMP 1A

$AA64 (-21916) \P1\ DOS 3.2/3.3 (48K) TEMP 2A

$AA65 (-21915) \P1\ DOS 3.2/3.3 (48K) COMMAND INPUT OPTIONS

$AA66~$AA74 (-21914~-21900) DOS 3.2/3.3 (48K) KEYWORD VALUES PARSED FROM COMMAND AND/OR DEFAULTED

$AA66~$AA67 (-21914~-21913) \P2\ DOS 3.2/3.3 (48K) COMMAND (OR DEFAULT) VOLUME

$AA68~$AA69 (-21912~-21911) \P2\ DOS 3.2/3.3 (48K) COMMAND (OR DEFAULT) DRIVE

$AA6A~$AA6B (-21910~-21909) \P2\ DOS 3.2/3.3 (48K) COMMAND (OR DEFAULT) SLOT

$AA6C~$AA6D (-21908~-21907) \P2\ DOS 3.2/3.3 (48K) COMMAND L-VALUE (LENGTH)

$AA6E~$AA6F (-21906~-21905) \P2\ DOS 3.2/3.3 (48K) COMMAND R-VALUE (RECORD NUMBER)

$AA70~$AA71 (-21904~-21903) \P2\ DOS 3.2/3.3 (48K) COMMAND B-VALUE (BYTE NUMBER)

$AA72~$AA73 (-21902~-21901) \P2\ DOS 3.2/3.3 (48K) COMMAND A-VALUE (ADDRESS)

$AA72~$AA73 (-21902~-21901) \P2\ CONTAINS START ADDRESS OF MOST RECENTLY BLOAD-ED PROGRAM OR DATA (DOS 3.2/3.3 - 48K APPLE)

$AA74 (-21900) \P1\ DOS 3.2/3.3 (48K) DOS 'C' 'I' & 'O' BITS

$AA75~$AA92 (-21899~-21870) \PB\ DOS 3.2/3.3 (48K) START OF LAST FILE NAME USED IN A DOS COMMAND. THIS IS NORMALLY FILE NAME OF BUFFER #3. IF RUN COMMAND USED WITHOUT FILE NAME THIS FIELD IS SET TO BLANKS. AT BOOT THIS AREA CONTAINS THE NAME OF THE GREETING PROGRAM

$AA93~$AAB0 (-21869~-21840) \PB\ DOS 3.2/3.3 (48K) START OF FILE NAME - BUFFER #2

$AAB1 (-21839) \P1\ DOS 3.2/3.3 (48K) DEFAULT NUMBER OF FILE BUFFERS (3)

$AAB2 (-21838) \P1\ DOS 3.2/3.3 (48K) COMMAND CHARACTER (CTRL-D)

$AAB3 (-21837) \P1\ DOS 3.2/3.3 (48K) EXEC FILE STATE (DIRECT~ DEFERRED ETC.)

$AAB4~$AAB5 (-21836~-21835) \P2\ DOS 3.2/3.3 (48K) EXEC FILE BUFFER POINTER

$AAB6 (-21834) \P1\ DOS 3.2/3.3 (48K) APPLESOFT-INTEGER BASIC SWITCH ($00=INTEGER BASIC;$40=ROM APPLESOFT;$80=RAM APPLESOFT)

$AAB7 (-21833) \P1\ DOS 3.2/3.3 (48K) APPLESOFT - BEGIN RUN SWITCH ($00=NO;$40 OR $80=YES)

$AAB8~$AAC0 (-21832~-21824) \PB\ TEXT WORD 'APPLESOFT' (NAME OF DOS 3.2/3.3 FP FILE USED TO GET DISK APPLESOFT)

$AAC1~$AAC2 (-21823~-21822) \P2\ DOS 3.2/3.3 (48K) ADDRESS POINTER TO IOB (RWTS BUFFER) NOTE: THIS IS LOADED INTO Y & A-REGS WHEN $03E3 IS BRANCHED TO

$AAC3~$AAC4 (-21821~-21820) \P2\ DOS 3.2/3.3 (48K) ADDRESS POINTER TO VTOC BUFFER (BUFFER FOR TRACK/SECTOR LIST - USED BY RWTS

$AAC5~$AAC6 (-21819~-21818) \P2\ DOS 3.2/3.3 (48K) ADDRESS POINTER TO SYS BUFFER (BUFFER FOR DATA - USED BY RWTS)

$AAC7~$AAC8 (-21817~-21816) \P2\ DOS 3.2/3.3 (48K) ADDRESS POINTER TO TOP OF RAM+1

$AAC9~$AAFC (-21815~-21764) \SB\ DOS 3.2/3.3 (48K) I/O PACKAGE COMMANDS FUNCTIONAL-CODE LOOK-UP TABLE. THIS TABLE IS USED AT $AB14 TO $AB1E TO JUMP TO CORRECT I-O ROUTINE. $B5BB IS USED TO CHOOSE WHICH I-O ROUTINE WILL BE CALLED

$AAC9~$AACA (-21815~-21814) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'GOOD RETURN' ($B37F-1)

$AACB~$AACC (-21813~-21812) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'OPEN FILE' ($AB22-1)

$AACD~$AACE (-21811~-21810) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'CLOSE FILE' ($AC06-1)

$AACF~$AAD0 (-21809~-21808) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'READ FROM FILE' ($$AC58-1)

$AAD1~$AAD2 (-21807~-21806) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'WRITE TO FILE' ($AC70-1)

$AAD3~$AAD4 (-21805~-21804) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'DELETE FILE' ($AD2B-1)

$AAD5~$AAD6 (-21803~-21802) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'PRINT CATALOG' ($AD98-1)

$AAD7~$AAD8 (-21801~-21800) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'LOCK A FILE' ($ACEF-1)

$AAD9~$AADA (-21799~-21798) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'UNLOCK A FILE' ($ACF6-1)

$AADB~$AADC (-21797~-21796) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'RENAME FILE' ($AC3A-1)

$AADD~$AADE (-21795~-21794) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'POSITION FILE' ($AD12-1)

$AADF~$AAE0 (-21793~-21792) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'FORMAT DISK (INIT)' ($AE8E-1)

$AAE1~$AAE2 (-21791~-21790) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR 'VERIFY FILE' ($AD18)

$AAE3~$AAE4 (-21789~-21788) \P2\ DOS 3.2/3.3 (48K) I-O PKG ADDRESS FOR GOOD RETURN ($B37F-1) [DUMMY ENTRY IN TABLE?]

$AAE5~$AAF0 (-21787~-21776) \PB\ DOS 3.2/3.3 (48K) I-O PKG READ COMMAND ENTRY-VECTOR TABLE. THIS TABLE USED AT $AC58 TO $AC69 TO JUMP TO CORRECT READ ROUTINE. THE VALUE OF $B5BC IS USED TO GET THE CORRECT ENTRY AND A JUMP IS MADE TO THERE

$AAE5~$AAE6 (-21787~-21786) \P2\ DOS 3.2/3.3 (48K) I-O PKG READ COMMAND ENTRY-VECTOR FOR 'GOOD RETURN' ($B37F-1)

$AAE7~$AAE8 (-21785~-21784) \P2\ DOS 3.2/3.3 (48K) I-O PKG READ COMMAND ENTRY-VECTOR FOR 'READ NEXT BYTE' ($AC8A-1)

$AAE9~$AAEA (-21783~-21782) \P2\ DOS 3.2/3.3 (48K) I-O PKG READ COMMAND ENTRY-VECTOR FOR 'READ NEXT BLOCK' ($AC96-1)

$AAEB~$AAEC (-21781~-21780) \P2\ DOS 3.2/3.3 (48K) I-O PKG READ COMMAND ENTRY-VECTOR FOR 'READ SPECIFIC BYTE' ($AC93-1)

$AAED~$AAEE (-21779~-21778) \P2\ DOS 3.2/3.3 (48K) I-O PKG READ COMMAND ENTRY-VECTOR FOR 'READ SPECIFIC BLOCK' ($AC93-1)

$AAEF~$AAF0 (-21777~-21776) \P2\ DOS 3.2/3.3 (48K) I-O PKG READ COMMAND ENTRY-VECTOR FOR 'GOOD RETURN' (DUMMY??)($B37F-1)

$AAF1~$AAFC (-21775~-21764) \PB\ DOS 3.2/3.3 (48K) I-O PKG WRITE COMMAND ENTRY-VECTOR TABLE. THIS TABLE IS USED AT $AC70 TO $AC86 TO JUMP TO THE CORRECT WRITE ROUTINE. THE VALUE OF $B5BC IS USED TO SPECIFY WHICH ROUTINE WILL BE JUMPED TO

$AAF1~$AAF2 (-21775~-21774) \P2\ DOS 3.2/3.3 (48K) I-O PKG WRITE COMMAND ENTRY-VECTOR FOR 'GOOD RETURN' ($B37F-1)

$AAF3~$AAF4 (-21773~-21772) \P2\ DOS 3.2/3.3 (48K) I-O PKG WRITE COMMAND ENTRY-VECTOR FOR 'WRITE NEXT BYTE' ($ACBE-1)

$AAF5~$AAF6 (-21771~-21770) \P2\ DOS 3.2/3.3 (48K) I-O PKG WRITE COMMAND ENTRY-VECTOR FOR 'WRITE NEXT BLOCK' ($ACCA-1)

$AAF7~$AAF8 (-21769~-21768) \P2\ DOS 3.2/3.3 (48K) I-O PKG WRITE COMMAND ENTRY-VECTOR FOR 'WRITE SPECIFIC BYTE' ($ACBB-1)

$AAF9~$AAFA (-21767~-21766) \P2\ DOS 3.2/3.3 (48K) I-O PKG WRITE COMMAND ENTRY-VECTOR FOR 'WRITE SPECIFIC BLOCK' ($ACC7-1)

$AAFB~$AAFC (-21765~-21764) \P2\ DOS 3.2/3.3 (48K) I-O PKG WRITE COMMAND ENTRY-VECTOR FOR 'GOOD RETURN' ($BE7F-1) [DUMMY ENTRY IN TABLE?]

$AAFD~$B5FF (-21763~-18945) \SB\ DOS 3.2/3.3 (48K) FILE MANAGER I-O PACKAGE (INCLUDING PARAMETER & SYSTEM BUFFER AREAS). CONTAINS CODE TO PERFORM FUNCTIONS LIKE OPEN~ CLOSE~ RENAME~ DELETE~ WRITE BYTES TO A FILE~ READ BYTES FROM A FILE ETC. NOTE: REFERENCED FROM PAGE 3 BY BRANCH FROM $03D6

$AAFD~$B396 (-21763~-19562) \SB\ DOS 3.2/3.3 (48K) FILE MANAGER (I-O PACKAGE) CODE (LESS PARAMETER & SYSTEM BUFFER AREAS)

$AAFD (-21763) \SE\ DOS 3.2/3.3 FILE MANAGER (I-O PACKAGE) ENTRY POINT. REFERENCED FROM PAGE 3 BY A BRANCH FROM $03D6

$AB22 (-21726) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO OPEN FILE

$AB28 (-21720) DOS 3.2/3.3 (48K). READS VTOC & DIRECTORY ATTEMPTING TO FIND AN ENTRY WITH SAME NAME AS THAT GIVEN. IF NOT FOUND CHECKS TABLE OF MASKS TO SEE IF IT IS ALLOWED TO CREATE A FILE. IF IT IS ALLOWED IT DOES SO; IF NOT EXITS WITH 'FILE NOT FOUND' OR 'LANGUAGE NOT AVAILABLE'

$ABDC (-21540) DOS 3.2/3.3 (48K) - CLEARS MISCELLANEOUS INFO HARDWARE BUFFER;SETS VOLUME NUMBER DRIVE NUMBER AND SLOT NUMBER

$AC06 (-21498) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO CLOSE FILE (UPDATES VTOC~ TRACK BIT MAP AND SECTOR COUNT OF DIRECTORY ENTRY AS NEEDED)

$AC3A (-21446) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO RENAME FILE (FINDS DIRECTORY ENTRY~ STORES NEW NAME IN ENTRY~ THEN WRITES THAT DIRECTORY SECTOR BACK TO DISK)

$AC58 (-21416) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO READ FROM FILE

$AC70 (-21392) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO WRITE TO FILE

$AC87 (-21369) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO READ SPECIFIC BYTE

$AC8A (-21366) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO READ NEXT BYTE

$AC93 (-21357) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO READ SPECIFIC BLOCK

$ACBB (-21317) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO WRITE SPECIFIC BYTE

$AC96 (-21354) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO READ NEXT BLOCK

$ACBE (-21314) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO WRITE NEXT BYTE

$ACC7 (-21305) \SE\ DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO WRITE SPECIFIC BLOCK

$ACCA (-21302) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO WRITE NEXT BLOCK
$ACEF (-21265) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO LOCK A FILE
$ACF6 (-21258) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO UNLOCK A FILE
$AD12 (-21230) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO POSITION FILE
$AD18 (-21224) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO VERIFY FILE
$AD2B (-21205) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO DELETE FILE
$AD54 (-21164)	DOS 3.2/3.3 (48K) PART OF DELETE ROUTINE WHICH FREES SECTORS USED BY DELETED FILE
$AD98 (-21096) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO PRINT CATALOG
$AE39 (-20935)	DOS 3.2/3.3 (48K) PART OF CATALOG ROUTINE RESPONSIBLE FOR PAUSING DURING A CATALOG LISTING. TO DISABLE THIS INSTRUCTION SIMPLY PATCH OVER IT WITH 3 NOP'S
$AE42 (-20926)	DOS 3.2/3.3 (48K) PART OF CATALOG ROUTINE WHICH PRINTS THE NUMBER IN $0044 AS 3 DIGIT ASCII
$AE6A (-20886)	DOS 3.2/3.3 (48K) MOVES MISCELLANEOUS INFO FROM THE FILE BUFFER TO THE I-O PKG BUFFER
$AE7E (-20866)	DOS 3.2/3.3 (48K) MOVES MISCELLANEOUS INFO FROM THE FILE BUFFER TO THE I-O PKG BUFFER
$AE8E (-20850) \SE\	DOS 3.2/3.3 (48K) I-O PKG ROUTINE TO FORMAT A DISK (INIT)
$AF1D (-20707)	DOS 3.2/3.3 (48K) WRITES DATA SECTION OF FILE BUFFER TO DISK
$AF34 (-20684)	DOS 3.2/3.3 (48K) WRITES TRACK/SECTOR LIST SECTION OF FILE BUFFER TO DISK
$AF4B (-20661)	DOS 3.2/3.3 (48K) SETS HARDWARE POINTER TO THE TRACK AND SECTOR LIST SECTION OF THE FILE BUFFER BEING USED
$AF5E (-20642)	DOS 3.2/3.3 (48K) CHECKS POSITION IN FILE. IF OUT OF CURRENT SECTOR READS/WRITES NEXT SECTOR~ UPDATES VTOC BUFFER~ UPDATES TRACK/SECTOR LIST SECTION OF FILE BUFFER IF IN WRITE MODE
$AFDC (-20516)	DOS 3.2/3.3 (48K) READS FROM DISK INTO DATA SECTION OF FILE BUFFER
$AFE4 (-20508)	DOS 3.2/3.3 (48K) SETS HARDWARE POINTERS TO DATA SECTION OF FILE BUFFER BEING USED
$AFF7 (-20489)	DOS 3.2/3.3 (48K) READS VTOC TO ITS BUFFER ($B3BB~$B4BA)
$AFFB (-20485)	DOS 3.2/3.3 (48K) WRITES VOTC FROM ITS BUFFER ($B3BB~$B4BA)
$B011 (-20463)	DOS 3.2/3.3 (48K) READS A DIRECTORY SECTOR INTO ITS BUFFER ($B4BB~$B5BA). INITIALLY READS SECTOR A. SUCCESSIVE ENTRIES INTO THIS SUBROUTINE READ SUCCESSIVE SECTORS FROM DISK. WHEN ALL SECTORS READ AND SUBROUTINE CALLED AGAIN IT MAY EXIT WITH CARRY SET
$B037 (-20425)	DOS 3.2/3.3 (48K) WRITES CURRENT DIRECTORY SECTOR FROM BUFFER TO DISK
$B052 (-20398)	DOS 3.2/3.3 (48K) SETS UP IOB FOR DIRECTORY SECTORS; GOES TO RWTS
$B0A0 (-20320)	DOS 3.2/3.3 (48K) NO ERROR EXIT TO $B052
$B0A1 (-20319)	DOS 3.2/3.3 (48K) START OF ERROR-HANDING ROUTINE FOR $B052
$B0B6 (-20298)	DOS 3.2/3.3 (48K) CHECKS POSITION IN FILE; READS/WRITES NEXT SECTOR AS NEEDED
$B134 (-20172)	DOS 3.2/3.3 (48K) INITIALIZES DATA SECTION OF FILE BUFFER TO ALL ZEROES
$B15B (-20133)	DOS 3.2/3.3 (48K) SETS NEXT POSITION IN FILE
$B194 (-20076)	DOS 3.2/3.3 (48K) INCREMENTS POSITION IN FILE
$B1A2 (-20062)	DOS 3.2/3.3 SETS NEXT RAM ADDRESS
$B1B5 (-20043)	DOS 3.2/3.3 CALCULATES HOW MUCH RAM IS LEFT
$B1C9~$B213 (-20023~-19941)	DOS 3.3 - LOCATE OR ALLOCATE A DIRECTORY ENTRY IN THE CATALOG; READ THE VTOC SECTOR ($AFF7); SET $0042~$0043 TO POINT TO FILE NAME BEING LOOKED FOR; SET PAGE NUMBER TO 1 (LOCATE FILE).
$B1C9 (-20023)	DOS 3.2/3.3 READS VTOC (VOLUME TABLE OF CONTENTS) AND SUCCESSIVE ENTRIES ATTEMPTING TO FIND SPECIFIED FILE NAME
$B21C~$B22F (-19940~-19921)	DOS 3.3 - COPY FILE NAME TO DIRECTORY ENTRY. ADVANCE INDEX TO FILE NAME FIELD IN DIRECTORY ENTRY; COPY 30 BYTE FILENAME TO DIRECTORY ENTRY; RELOAD DIRECTORY INDEX AND RETURN TO CALLER

$B21E (-19938) DOS 3.2/3.3 PUTS NAME OF FILE INTO DIRECTORY
$B224 (-19932) DOS 3.2/3.3 SETS NEXT SECTOR; UPDATES VTOC BUFFER
$B230 (-19920) DOS 3.3 ADVANCE INDEX TO NEXT DIRECTORY ENTRY IN SECTOR; ADD 35 (LENGTH OF ENTRY) TO INDEX; TEST FOR END OF SECTOR AND RETURN TO CALLER
$B23A~$B243 (-19910~-19901) DOS 3.3 SWITCH TO SECOND PASS IN DIRECTORY SCAN. IF ON PASS ONE~ SWITHC TO PASS 2 AND GOTO $B1D8; IF ON PASS TWO EXIT FILE MANAGER WITH 'DISK FULL' ERROR
$B244~$B2C2 (-19900~-19774) DOS 3.3 ALLOCATE A DISK SECTOR
$B2C3~$B2DC (-19773~-19748) DOS 3.3 RELEASE PRE-ALLOCATED SECTORS IN CURRENT TRACK AND CHECKPOINT THE VTOC.
$B2C3 (-19773) DOS 3.2/3.3 UPDATES VTOC
$B2DD~$B2FF (-19747~-19713) DOS 3.3 (48K) - FREE ONE OR MORE SECTORS BY SHIFTING MASK IN FILE MANAGER'S ALLOACTION AREA BACK INTO VTOC BIT MAP
$B2DD (-19747) DOS 3.2/3.3 CALCULATES TRACK BIT MAP FOR VTOC
$B300~$B35E (-19712~-19618) DOS 3.3 (48K) - CALCULATE FILE POSITION
$B35F~$B37D (-19617~-19587) DOS 3.3 (48K) - ERROR EXISTS
$B35F (-19617) DOS 3.3 (48K) - RC=1 "LANGUAGE NOT AVAILABLE"
$B363 (-19613) DOS 3.3 (48K) - RC=2 "RANGE ERROR" (BAD OPCODE)
$B367 (-19609) DOS 3.3 (48K) - RC=3 "RANGE ERROR" (BAD SUBCODE)
$B36B (-19605) DOS 3.3 (48K) - RC=4 "WRITE PROTECTED"
$B36F (-19601) DOS 3.3 (48K) - RC=5 "END OF DATA"
$B373 (-19597) DOS 3.3 (48K) - RC=6 "FILE NOT FOUND"
$B37B (-19589) DOS 3.3 (48K) - RC=A "FILE LOCKED"
$B37F~$B396 (-19585~-19562) EXIT FILE MANAGER
$B37F (-19585) \SE\ DOS 3.2/3.3 (48K) FILE MANAGER (I-O) PKG GOOD RETURN (RETURN CODE =0; CLEAR CARRY FLAG AND GO TO $B386)
$B385 (-19579) DOS 3.3 (48K) - EXIT SETTING CARRY FLAG TO INDICATE ERROR
$B386 (-19578) DOS 3.3 (48K) - SAVE RETURN CODE IN PARAMLIST; CLEAR MONITOR STATUS REGISTER; SAVE FILE MANAGER WORKAREA TO FILE BUFFER ($AE7E); RESTORE PROCESSOR STATUS AND STACK REGISTER; EXIT TO ORIG CALLER OF FILE MANAGER
$B397~$B6FF (-19561~-18689) DOS 3.2/3.3 (48K) FILE MANAGER (I-O PACKAGE) DATA AREA (PARAMETERS & SYSTEM BUFFER)
$B397~$B3A3 (-19561~-19549) DOS 3.3 FILE MANAGER (I-O PACKAGE) SCRATCH SPACE
$B397 (-19561) \P2\ DOS 3.2/3.3 (48K) CONTAINS TRACK AND SECTOR ADDRESS OF MOST RECENTLY READ DIRECTORY (CATALOG) SECTOR
$B39B (-19557) DOS 3.3 FILE MANAGER S-REGISTER SAVE AREA
$B39C (-19556) DOS 3.3 FILE MANAGER DIRECTORY INDEX
$B39D (-19555) DOS 3.3 FILE MANAGER CATALOG LINE COUNTER~ DIRECTOR LOOKUP FLAG ~ ETC
$B39E (-19554) DOS 3.3 (48K) LOCK/UNLOCK MASK~ ALLOCATION FLAG ETC.
$B3A0 (-19552) DOS 3.3 (48K) FOUR BYTE MASK USED BY INIT TO FREE AN ENTIRE TRACK IN THE VTOC BIT MAP
$B3A4~$B3A6 (-19548~-19546) DOS 3.3 (48K) DECIMAL CONVERSION TABLE (1~10~100)
$B3A7~$B3AE (-19545~-19538) DOS 3.3 (48K) FILE TYPE NAME TABLE USED BY CATALOG. FILE TYPES ARE T~I~A~B~S~R~A~B CORRESPONDING TO HEX VALUES OF $00~$02$04~$08~$10~$20 AND $40 RESPECTIVELY
$B3A7 (-19545) DOS 3.2 (48K) CONTAINS 4 FILETYPE CHARACTERS T~ I~ A~ AND B
$B3AF~$B3BA (-19537~-19526) DOS 3.2/3.3 (48K) CONTAINS CHARACTER STRING 'DISK VOLUME' FOR CATALOG COMMAND (IN REVERSE ORDER)
$B3BB~$B4BA (-19525~-19270) DOS 3.2/3.3 (48K) VTOC SECTOR BUFFER PART OF SYSTEM BUFFER - CONTAINS THE MASTER TRACK/SECTOR BIT MAP SECTOR OR VOLUME TABLE OF CONTENTS
$B3BC (-19524) TRACK~SECTOR OF FIRST DIRECTOR SECTOR
$B3BE (-19522) DOS RELEASE NUMBER (1~2 OR 3 FOR 3.1~3.2 OR 3.3)
$B3C1 (-19519) VOLUME NUMBER OF DISKETTE

$B3E2 (-19486)	NUMBER OF ENTRIES IN EACH TRACK~SECTOR LIST SECTOR
$B3EB (-19477)	TRACK TO ALLOCATE NEXT
$B3EC (-19476)	DIRECTION OF TRACK ALLOCATION (+1 OR -1)
$B3EF (-19473) \P1\	NUMBER OF TRACKS ON A DISK
$BEF0 (-16656) \P1\	NUMBER OF SECTORS ON A DISK
$BEF1 (-16655) \P2\	SECTOR SIZE IN BYTES
$B3F3~$B47B (-19469~-19333)	ARRAY OF 34 TRACK BIT MAPS
$B3F3 (-19469)	TRACK 0 BIT MAP
$B3F4 (-19468)	TRACK 1 BIT MAP
$B3F5 (-19467)	TRACK 2 BIT MAP
$B3F6 (-19466)	TRACK 3 BIT MAP
$B47A (-19334)	TRACK 33 BIT MAP
$B47B (-19333)	TRACK 34 BIT MAP
$B3EF~$B642 (-19473~-18878) \HB\	DOS 3.1 (48K) SYSTEM BUFFER (FOR CATALOG ETC.)(SEE $B4BB FOR DOS 3.2~3.2.1~3.3)
$B4BB~$B5BA (-19269~-19014)	DOS 3.2/3.3 DIRECTORY SECTOR BUFFER PART OF SYSTEM BUFFER. CONTAINS LAST ACCESSED DIRECTORY SECTORY SECTOR (ACCESS BY A CATALOG COMMAND OR ANY OTHER DOS COMMAND REQUIRING A DIRECTORY SECTORY SEARCH)
$B4BC (-19268)	TRACK~SECTOR OF NEXT DIRECTORY SECTOR
$B4C6 (-19258)	FIRST DIRECTORY ENTRY AND TRACK OF TRACK-SECTOR LIST
$B4C7 (-19257)	SECTOR OF TRACK-SECTOR LIST
$B4C8 (-19256)	FILE TYPE AND LOCK BIT
$B4C9 (-19255)	FILENAME FIELD (30 BYTES)
$B4E7 (-19225)	SIZE OF FILE IN SECTORS (INCLUDING TRACK~SECTOR LIST(S))
$B5BB~$B5D0 (-19013~-18992)	DOS 3.2/3.3 (48K) FILE MANAGER PARAMETER LIST
$B5BB (-19013)	1ST BYTE BEYOND SYSTEM BUFFER. PAGE 3 ROUTINE AT $03DC LOADS Y-REG & A-REG TO POINT HERE
$B5BB (-19013) \P\	DOS 3.2/3.3 (48K) I-O PKG 'OPCODE' PARAMETER USED TO CHOOSE WHICH I-O PKG 'OPCODE' ROUTINE WILL BE CALLED
$B5BC (-19012)	DOS 3.2/3.3 (48K) I-O PKG 'SUBCODE' PARAMETER USED TO CHOOSE WHICH READ OR WRITE OPTION IS TO BE USED
$B5BD~$B5C4 (-19011~-19004)	DOS 3.2/3.3 (48K) EIGHT BYTES OF PARAMETERS. PARAMETERS VARY ACCORDING TO 'OPCODE' PARAMETER IN $B5BB
$B5C5 (-19003)	DOS 3.2/3.3 FILE MANAAGER PARAMETER LIST RETURN CODE
$B5C7 (-19001)	DOS 3.2/3.3 ADDRESS OF FILE MANAGER WORK AREA BUFFER
$B5C9 (-18999)	DOS 3.2/3.3 ADDRESS OF TRACK/SECTOR LIST SECTOR BUFFER
$B5CB (-18997)	DOS 3.2/3.3 (48K) ADDRESS OF DATA SECTOR BUFFER
$B5CD (-18995)	DOS 3.2/3.3 (48K) ADDRESS OF NEXT DOS BUFFER ON CHAIN (NOT USED)
$B5D1~$B5FD (-18991~-18947)	DOS 3.2/3.3 FILE MANAGER WORK AREA
$B5D1 (-18991)	FIRST TRACK-SECTOR LIST SECTOR'S TRACK & SECTOR
$B5D3 (-18989)	CURRENT TRACK-SECTOR LIST SECTOR'S TRACK & SECTOR
$B5D5 (-18987)	FLAGE: 80=T~S LIST NEEDS CHECKPOINT;40=DATA SECTOR NEEDS CHECKPOINT;20=VTOC SECTOR NEEDS CHECKPOINT;02 LAST OPERATION WAS WRITE
$B5D6 (-18986)	CURRENT DATA SECTOR'S TRACK~SECTOR
$B5D8 (-18984)	DIRECTORY SECTOR INDEX FOR FILE ENTRY
$B5D9 (-18983)	INDEX INTO DIRECTORY SECTOR TO DIRECTORY ENTRY FOR FILE
$B5DA (-18982)	NUMBER OF SECTORS DESCRIBED BY ON TRACK~SECTOR LIST
$B5DC (-18980)	RELATIVE SECTOR NUMBER OF FIRST SECTOR IN LIST
$B5DE (-18978)	RELATIVE SECTOR NUMBER +1 OF LAST SECTOR IN LIST
$B5E0 (-18976)	RELATIVE SECTOR NUMBER OF LAST SECTOR READ
$B5E2 (-18974)	SECTOR LENGTH IN BYTES
$B5E4 (-18972)	FILE POSITION (3 BYTES); SECTOR OFFSET;BYTE OFFSET INTO THAT SECTOR

$B5E8 (-18968)		RECORD LENGTH FROM 'OPEN'
$B5EA (-18966)		RECORD NUMBER
$B5EC (-18964)		BYTE OFFSET INTO RECORD.
$B5EE (-18962)		NUMBER OF SECTORS IN FILE
$B5F0 (-18960)		SECTOR ALLOCATION AREA (6 BYTES). NEXT SECTOR TO ALLOCATE (SHIFT COUNT);TRACK BEING ALLOCATED;FOUR BYTE BIT MAP OF TRACK BEING ALLOCATED~ ROTATED TO NEXT SECTOR TO ALLOCATE
$B5F6 (-18954)		FILE TYPE
$B5F7 (-18953)		SLOT NUMBER *16
$B5F8 (-18952)		DRIVE NUMBER
$B5F9 (-18951)		VOLUME NUMBER (IN COMPLEMENT FORM)
$B5FA (-18950)		TRACK NUMBER
$B5FE~$B5FF (-18946~-18945)		DOS 3.2/3.3 NOT USED
$B600~$B6FF (-18944~-18689)		DOS 3.2/3.3 BOOT SECTOR BUFFER; I.E. BOOT 2 RWTS (READ-WRITE TRACK-SECTOR) IMAGE
$B600 (-18944)		DOS 3.3 (48K) START OF PHASE 2 (BOOT 1) IMAGE WHICH CAN BE WRITTEN TO INIT'ED DISKS ON TRACK 0 SECTOR 0
$B65D (-18851)		DOS 3.3 (48K) PATCH AREA STARTS HERE WITH APPEND PATCH
$B65E (-18850)		DOS 3.3 (48K) ANOTHER APPEND PATCH STARTS HERE
$B686 (-18810)		DOS 3.3 (48K) VERIFY PATCH
$B692 (-18798)		DOS 3.3 (48K) ANOTHER APPEND PATCH STARTS HERE
$B6FE (-18690)		DOS 3.3 (48K) PAGE ADDRESS OF FIRST PAGE IN PHASE 3 (BOOT 2)
$B6FF (-18689)		DOS 3.3 (48K) NUMBER OF SECTORS (PAGES) IN PHASE 3 (BOOT 2)
$B700 (-18688)		DOS 3.2 BOOTSTRAP LOADER FOR PHASE 3 (BOOT 2) OF DOS BOOT (PHASE 1 IN DISK CONTROLLER ROM; PHASE 2 IN PAGE 3 [$300-$3FF]). READS DRIVE1 CURRENT SLOT $B1 SECTORS AND TRACK 0 SECTOR A INTO RAM STARTING AT $1B00
$B700~$B749 (-18688~-18615)		DOS 3.3 BOOTSTRAP LOADER FOR PHASE 3 (BOOT 2) OF DOS BOOT (PHASE 2 IN $0800-$08FF). SETS RWTS PARAMETER LIST TO READ DOS FROM DISK; CALLS READ~WRITE GROUPP OF PAGES ($B793) & CREATES NEW STACK. ALSO CALLS $FE93 (SETVID) AND $FE89 (SETKBD) AND EXITS TO COLDSTART AT $9D84
$B74A~$B78C (-18614~-18548)		DOS 3.3 (48K) - WRITES DOS ONTO TRACKS 0-2. SETS RWTS PARAMETER LIST TO READ DOS FROM DISK; CALLS READ/WRITE GROUP OF PAGES ($B793); EXITS TO CALLER
$B74A (-18614)		DOS 3.2 (48K) - WRITES $0A SECTORS STARTING FROM $B600~ THEN $1B SECTORS STARTING AT $1B00 BEGINNING AT TRACK 0 SECTOR 0
$B78D~$B792 (-18547~-18542)		DOS 3.3 (48K) UNUSED
$B793~$B7B4 (-18541~-18508)	\SB\	DOS 3.3 (48K) READ/WRITE A GROUP OF PAGES. CALLS RWTS THROUGH EXTERNAL ENTRY POINT $B7B5 & EXITS TO CALLER
$B793 (-18541)		DOS 3.2~3.2.1~3.3 ROUTINE TO STORE A BLOCK OF CONSECUTIVE SECTORS. LOADS COMMAND BYTE ($B7FC);NUMBER OF SECTORS IN $B7E1; DOS 3.2 INCREMENTS - SET UP IOBLK TO 1ST SECTOR; DOS 3.3 DECREMENTS - SET UP IOBLK TO LAST SECTOR
$B793 (-18541)		DOS 3.2 (48K) - INCREMENTS OR DECREMENTS TRACK/SECTOR AS NEEDED AND DATA ADDRESS FOR $B700 & $B793 ROUTINES
$B7B5~$B7C1 (-18507~-18495)		DOS 3.3 (48K) DISABLE INTERRUPTS AND CALL RWTS
$B7B5 (-18507)		DOS 3.2 (48K) START OF RWTS-IN-ENVIRONMENT ROUTINE. DISABLES INTERRUPTS; CALLS RWTS (LOCATED AT $BD00); RE-ENABLES INTERRUPS AND PASSES BACK RETURN CODE FROM RWTS IN FORM OF CARRY FLAG
$B7C2~$B7D5 (-18494~-18475)		DOS 3.3 (48K) SET RWTS PARAMETERS FOR WRITING DOS
$B7C2 (-18494)		DOS 3.2 (48K) SETS ADDRESS OF DATA BUFFER AND SETS EXPECTED VOLUME NUMBER
$B7D6~$B7DE (-18474~-18466)		DOS 3.3 (48K) ZERO CURRENT BUFFER (256 BYTES POINTED TO BY $0042~$0043) AND EXIT TO CALLER
$B7DB (-18469)		DOS 3.2 (48K) STORES ZEROES IN ONE PAGE STARTING AT ADDRESS IN $0042~$0043
$B7DF~$B7E7 (-18465~-18457)		DOS 3.3 (48K) DOS PHASE 3 (BOOT 2) BOOT LOADER PARAMETER LIST

$B7DF (-18465) DOS 3.3 (48K) UNUSED
$B7E0 (-18464) DOS 3.3 (48K) NUMBER OF PAGES IN 2ND DOS LOAD (PHASE 3)
$B7E1 (-18463) DOS 3.3 NUMBER OF SECTORS TO READ/WRITE
$B7E2 (-18462) DOS 3.3 NUMBER OF PPAGES IN 1ST DOS LOAD (PHASE 2)
$B7E3 (-18461) DOS 3.3 INIT DOS PAGE COUNTER
$B7E4~$B7E5 (-18460~-18459) DOS 3.3 POINTER TO RWTS PARAMETER LIST
$B7E6~$B7E7 (-18458~-18457) DOS 3.3 POINTER TO 1ST STAGE BOOT LOCATION
$B7E8~$B7F8 (-18456~-18440) DOS 3.2/3.3 (48K) RWTS PARAMETER LIST OR SYSTEM IOB. THIS IOB SET UP ACCORDING TO LAST DOS OPERATION THAT OCCURED
$B7E8 (-18456) DOS 3.2/3.3 (48K) - TABLE TYPE. MUST BE $01
$B7E9 (-18455) DOS 3.2/3.3 (48K) - SLOT NUMBER * 16
$B7EA (-18454) DOS 3.2/3.3 (48K) - DRIVE NUMBER ($01 OR $02)
$B7EB (-18453) DOS 3.2/3.3 (48K) - VOLUME NUMBER (NOTE: 0 MATCHES ANY VOLUME.)
$B7EC (-18452) DOS 3.2/3.3 (48K) - TRACK NUMBER ($00~$22)
$B7ED (-18451) DOS 3.2/3.3 (48K) - SECTOR NUMBER (DOS 3.2 $0~$C;DOS 3.3 $0~$F)
$B7EE~$B7EF (-18450~-18449) DOS 3.2/3.3 - POINTER TO DCT (DEVICE CHARACTERISTICS TABLE)
$B7F0~$BFF1 (-18448~-16399) DOS 3.2/3.3 - POINTER TO USER DATA BUFFER FOR READ/WRITE
$B7F2 (-18446) DOS 3.2/3.3 - UNUSED
$B7F3 (-18445) DOS 3.2/3.3 - BYTE COUNTER FOR PARTIAL SECTOR; USE $00 FOR FULL 256 BYTES
$B7F4 (-18444) DOS 3.2/3.3 - COMMAND CODE: 0=SEEK; 1=READ; 2=WRITE; 4=FORMAT.
$B7F5 (-18443) DOS 3.2/3.3 - ERROR CODE (VALID IF CARRY SET): $10=WRITE PROTECT;$20=VOLUME MISMATCH; $40=DRIVE ERROR; $08=INIT ERROR.
$B7F6 (-18442) DOS 3.2/3.3 - VOLUME NUMBER FOUND
$B7F7 (-18441) DOS 3.2/3.3 - SLOT NUMBER FOUND
$B7F8 (-18440) DOS 3.2/3.3 - DRIVE NUMBER FOUND
$B7F9~B7FA DOS 3.3 (48K) UNUSED
$B7FB~$B7FE (-18437~-18434) DOS 3.2/3.3 (48K) - DEVICE CHARACTERISTICS TABLE (DCT) ASSOCIATED WITH SYSTEM IOB. NOTE: IOB CONTAINS DETAILED DEVICE CHARACTERISTICS TABLE AS DOCUMENTED FOR RWTS IN WOZPACK
$B7FB (-18437) DOS 3.2/3.3 (48K) - DEVICE TYPE (SHOULD BE $00)
$B7FC (-18436) DOS 3.3 (48K) - PHASES PER TRACK (SHOULD BE $01)
$B7FD~$B7FE (-18435~-18434) DOS 3.3 (48K) - MOTOR ON TIME COUNT (SHOULD BE $EF & $D8)
$B7FF (-18433) DOS 3.3 (48K) - UNUSED.
$B800~$B869 (-18432~-18327) [PRENIBL-PRENIB16] \SB\DOS 3.1~3.2~3.3 RWTS (READ-WRITE TRACK-SECTOR) PRENIBL MODULE. CONVERTS A PAGE OF 256 OF REAL BYTES TO A SECTOR OF 410 ($19A) RIGHT JUSTIFIED 5 BIT NIBBLES (EXCEPT DOS 3.3 CONVERTS TO 342 6 BIT NIBBLES OF THE FORM 00XXXXXX). POINTER TO PAGE TO CONVERT AT $003E~$003F; DATA STORED AT PRIMARY XXX) SECONDARY BUFFERS; ON EXIT X-REG XXX) Y-REG CONTAIN $FF & CARRY SET.
$B82A~$B8B7 (-18390~-18249) [WRITE16 (DOS 3.3)] \SB\DOS 3.3 'WRITE'. WRITES PRENIBBILIZED DATA FROM PRIMARY & SECONDARY BUFFERS TO DISK; CALLS WRITE-A-BYTE S~R; WRITES 5 BYTES AUTSYNC~ STARTING DATA MARKS ($D5~$AA~$AD)~ 342 BYTES DATA~ ONE BYTE CHECKSUM~ AND CLOSING DATA MARKS ($DE~$AA~$EB). USES WRITE TRANSLATE TABLE ($BA29). ON ENTRY X-REG CONTAINS SLOT#*16. ON EXIT X-REG UNCHANGED;Y-REG $00; CARRY CLEAR. USES $0026~ $0027~ $678
$B86A~$B8FC (-18326~-18180) [WRITE] \SB\DOS 3.1~3.2~3.2.1 (SEE $B82A FOR DOS 3.3 'WRITE') RWTS (READ-WRITE TRACK-SECTOR) WRITE MODULE. WRITES A BUFFER OF 410 ($19A) 5-BIT RIGHT-JUSTIFIED NIBBLES ONTO THE DISK SURFACE AS A SECTOR CONVERTING THEM TO A 8-BIT 'DISK BYTE' FORMAT FIRST
$B8B8~$B8C1 (-18248~-18239) DOS 3.3 WRITE-A-BYTE S/R. THIS IS TIMING-CRITCAL CODE USED TO WRITE BYTES AT 32 CYCLE INTERVALS. EXITS TO CALLER

HEX LOCN (DEC LOCN) [NAME] \USE-TYPE\ - DESCRIPTION

$B8C2~$B8DB (-18238~-18213) [POSTNB16] \SB\DOS 3.3 POSTNIBBLE ROUTINE. CONVERTS 342 6-BIT NIBBLES OF FORM 00XXXXXX TO 256 8-BIT BYTES. NIBLES STORED AT PRIMARY ($BB00-$BBFF) AND SECONDARY ($BC00~$BC55) BUFFERS. POINTER TO DATA PARGE STORED AT 'BUFPTR' ($003E~$003F). ON ENTRY X-REG= SLOT*16; CSW ($0036~$0037) POINTS TO USER DATA; $0026= BYTE COUNT IN SECONDARY BUFFER. ON EXIT CARRY SET 'BUFPTR' Y-REG CONTAINS BYTE COUNT IN SECONDARY BUFFER

$B8C2 (-18238) [POSTNIBL (DOS 3.3)] \SB\DOS 3.3 'POSTNIBL'

$B8DC~$B943 (-18212~-18109) [READ16] \SB\DOS 3.3 'READ' IN RWTS (READ-WRITE TRACK-SECTOR). READS A SECTOR OFF THE DISK INTO SECONDARY BUFFER ($BC00-$BC55) HIGH TO LOW THEN INTO PRIMARY ($BB00-$BBFF) LOW TO HIGH EN ROUTE TO OVERALL PROCESS OF FORMING $153 RIGHT-JUSTIFIED 6-BIT NIBBLES

$B8FD~$B964 (-18179~-18C76) [READ] \SB\DOS 3.1~3.2~3.2.1 (SEE $B8DC FOR DOS 3.3 'READ') RWTS (READ-WRITE TRACK-SECTOR READ MODULE. READS A SECTOR OFF THE DISK FORMING 410 ($19A) 5-BIT RIGHT-JUSTIFIED NIBBLES

$B944~$B99F (-18108~-18017) [READADR-RDADR16 (DOS 3.3)] \SB\DOS 3.3 READADR. FUNCTION SAME AS READADR-RDADR16 (DOS 3.2)

$B944 (-18108) [.RDADR16] DOS 3.3 SYNONYM FOR READADR

$B965~$B9C0 (-18075~-17984) [READADR (DOS 3.2)] \SB\DOS 3.1~3.2~3.2.1 (SEE $B944 FOR DOS 3.3 'READADR (DOS 3.2)') RWTS (READ-WRITE TRACK SECTOR) READ ADDRESS MODULE. READS ADDRESSES ON THE SECTORS OF CURRENT TRACK UNTIL IT FINDS A SECTOR. THEN IT RETURNS PUTTING CHECKSUM INTO $002C;SECTOR INTO $002D;TRACK INTO $002E; AND VOLUME INTO $002F. CARRY IS SET ON ERROR

$B9A0~$B9FC (-18016~-17924) [SEEKABS (DOS 3.3)] \SB\DOS 3.3 - MOVES DISK AREM TO DESIRED TRACK. CALLS ARM MOVE DELAY SUBROUTINE ($B9FD). ON ENTRY $0478 CONTAINS CURRENT TRACK; X-REG CONTAINS SLOT*16; A-REG DESIRED TRACK. ON EXIT X-REG UNCHANGED; A-REG Y-REG CLOBBERED; $0478 &$002A: FINAL TRACK;$27 PRIOR TRACK (IF SEEK NEEDED). USES $0026;$0027;$002A;$002B. EXITS TO CALLER

$B9C1~BA1D[POSTNIBL (DOS 3.2)] \SB\ DOS 3.1~3.2~3.2.1 (SEE $B8C2 FOR DOS 3.3) RWTS (READ-WRITE TRACK SECTOR) POSTNIBL (DOS 3.2) MODULE. CONVERTS A BUFFER OF 410 ($19A) LEFT-JUSTIFIED 5-BIT NIBBLES TO 256 ($100) REAL BYTES. $003E~$003F POINTS TO BUFFER TO PUT THEM INTO

$B9A0 (-18016) [SEEKABS (DOS 3.2)] \SB\DOS 3.2 'SEEKABS'

$B9EC (-17940) DOS 3.2 CODE TO IMPLEMENT INITIALIZATION WITH VOLUNME NUMBER TO BE INITIALIZED IN $002F

$B9FD~$BA1D (-17923~-17904) \SB\ DOS 3.3 ARM MOVE DELAY SUBROUTINE. DELAYS SPECIFIED NUMBER OF 100 MICROSEC INTERVALS. ON ENTRY A-REG CONTAINS NUMBER OF INTERVALS; 'MONTIME' ($0046) SHOULD CONTAIN MOTOR-ON TIME ($EF~$D8) FROM DCT;$0478 CONTAIN CURRENT TRACK; ON EXIT A-REG CURRENT X-REG CONTAIN $00; Y-REG UNCHANGED CARRY SET. EXIT TO CALLER

$BA00 (-17920) [MSWAIT] \SB\ DOS 3.3 RWTS OPERATION TIMER ROUTINE

$BA11 (-17903) \SB\ DOS 3.3 RWTS OPERATION TIMER TABLE1

$BA1E~$BA8F (-17890~-17777) [SEEKABS] \SB\DOS 3.1~3.2~3.2.1 (SEE $B9A0 FOR DOS 3.3) RWTS (READ-WRITE TRACK SECTOR) SEEKABS MODULE. MOVES HEAD TO TRACK SPECIFIED BY A-REG. $0478 IS CURRENT. RWTS DOES PHASE OFF FOR ALL FOUR BEFORE CALL

$BA29 (-17879) DOS 3.3 (48K) ROUTINE TO ENCODE NIBBLES(6 DATA BITS PER NIBBLE INSTEAD OF 5 AS IN DOS 3.1~3.2)

$BA11~BA28\SB\ DOS 3.3 ARM MOVE DELAY TABLE. CONTAINS VALUES OF 100 MICROSEC INTERVALUS USED DURING PHASE-ON AND PHASE-OFF OF STEPPER MOTOR

$BA29~$BA68 (-17879~-17816) DOS 3.3 WRITE TRANSLATE TABLE. CONTAINS 6-BIT NIBBLES USED TO CONVERT 8-BIT BYTES. VALUES RANGE FROM $96 TO $FF AND CODES WITH MORE THAN ONE PAIR OF ADJACENT ZEROS OR NO ADJACENT ONES ARE EXCLUDED

$BA69 (-17815) DOS 3.3 - UNUSED

$BA96~$BAFF (-17770~-17665) DOS 3.3 READ TRANSLATE TABLE. CONTAINS 8 BIT BYTES USED TO CONVERT 6-BIT NIBBLES. VALUES RANGE FROM $96 TO $FF. CODES WITH MORE THAN ONE PAIR OF ADJACENT ZEROS OR WITH NO ADJACENT ONES ARE EXCLUDED.

$BCC4~$BCDE (-17212~-17186) \SB\ DOS 3.3 WRITE DOUBLE BYTE SUBROUTINE. THIS IS TIMING CRITICAL CODE THAT ENCODES ADDRESS INFO INTO EVEN AND ODD BITS AND WRITES IT AT 32 CYCLE INTERVALS. EXIT TO CALLER

$BCDF~$BCFF (-17185~-17153) DOS 3.2 UNUSED

$BD00~$BD18 (-17152~-17128) DOS 3.3 MAIN ENTRY TO RWTS (READ-WRITE TRACK-SECTOR). UPON ENTRY STORE Y-REG AND A-REG AT $0048 AND $0049 AS POINTERS TO IOB. INITIALIZE NUMBER OF RECALS AT 1 AND SEEKS AT 4. IF SLOT # HAS NOT CHANGED BRANCH TO 'SAMESLOT' ($BD34)

$BD19~$BD33 (-17127~-17101) DOS 3.3 - UPDATE SLOT NUMBER IN IOB AND WAIT FOR OLD DRIVE TO TURN OFF

$BEAE (-16722) DOS 3.2 START OF CODE TO INITIALIZE A SINGLE TRACK

$BFA2 (-16478) DOS 3.2 - TESTS TO SEE IF ALL $22 TRACKS HAVE BEEN INITIALIZED YET; IF SO EXITS RWTS AT $BFB8

$BA7B (-17797) [(TIMER DOS 3.2.1)] \SB\DOS 3.2.1 RWTS OPERATION TIMER ROUTINE

$BA7F (-17793) [(TIMER DOS 3.1~3.2)] \SB\DOS 3.1~3.2 RWTS OPERATION TIMER ROUTINE

$BA8C (-17780) [(TABLE1 DOS 3.2.1)] \SB\DOS 3.2.1 RWTS OPERATION TIMER ROUTINE TABLE1

$BA90~$BA93 (-17776~-17765) \SB\ DOS 3.1~3.2 RWTS (READ-WRITE TRACK-SECTOR) TABLE OF PHASE-ON TIMES TO WAIT (LOCATED AT $BA8C IN DOS 3.2.1 & AT $BA11 IN DOS 3.3)

$BA96~$BAFF (-1777C~-17665) DOS 3.3 NIBBLE ENCODE/DECODE TABLE

$BA9C~$BAA7 (-17764~-17753) \SB\ DOS 3.1~3.2 RWTS (READ-WRITE TRACK-SECTOR) TABLE OF PHASE-OFF TIMES TO WAIT (LOCATED AT $BA98 IN DOS 3.2.1 & AT $BA1D IN DOS 3.3)

$BAA8~$BAFF (-17752~-17665) \SB\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) TABLE OF NIBBLES IN POSITION OF CORRESPONDING DISK BYTE OFFSET FROM $BA00 USED FOR CONVERSION DISK BYTES->NIBBLES

$BB00~$BC99 (-17664~-17255) \BB\ BUFFER TO HOLD 410 5-BYTE NIBBLES CREATED FROM A PAGE OF 256 BYTES BY PRENIBL ROUTINE IN DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR)

$BB00~$BBFF (-17664~-17409) DOS 3.3 RWTS PRIMARY BUFFER

$BC00~$BC55 (-17408~-17323) DOS 3.3 RWTS SECONDARY BUFFER

$BC56~$BCC3 (-17322~-17213) DOS 3.3 WRITE ADDRESS FIELD DURING INITIALIZATION. CALLS WRITE DOUBLE BYE SUBROUTINE. WRITES # OF BYTES CONTAINED IN Y-REG; STARTING ADDRESS MARKS ($D5/$AA/$96); ADDRESS INFO (VOL/TRACK/SECTOR/CHECKSUM);CLOSING ADDRESS MARKS ($DE~$AA~$EB). ON ENTRY X-REG CONTAINS SLOT*16;Y-REGCONTAINS NUMBER OF AUTOSYNC TO WRITE;$3E:$AA;$3F:SECTOR#;$41:VOL#;$44:TRACK#. ON EXIT: A-REG ?;X-REG UNCHANGED;Y-REG $00;CARRY SET. EXIT TO CALLER

$BC9A~BCB9\PB\ CONVERSION TABLE TO CONVERT 5-BIT NIBBLES TO 8-BIT 'DISK BYTES' USED BY DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) PACKAGE WRITE SUBROUTINE

$BCD0~BCDC DOS 3.2 (48K) - 13 BYTES CONTAINING PERMUTATIONS OF 5 MOD 13 USED IN NYBBLE CONVERSION. NO SUCH TABLE IN DOS 3.3

$BD00~$BFFF (-17152~-16385) DOS 3.2 (48K) - MAINLINE READ-WRITE TRACK-SECTOR (RWTS) CODE

$BD00 (-17152) \SE\ DOS 3.2 (48K) - ROUTINE WHICH READS IN DIRECTORY OFF DISK

$BD34~$BD53 (-17100~-17069) [SAMESLOT] \SB\ENTER READ MODE AND READ WITH DELAYS TO SEE IF DISK IS SPINNING. SAVE RESULTS OF TEST AND TURN ON MOTOR ANYHOW

$BD44 (-17084) ADDRESS OF DEVICE CHARACTERISTICS TABLE (DCT) AND BUFFER ARE MOVED FROM THE IOB INTO LOCNS $003C~$003D & $003E~$003F

$BD54~$BD73 (-17068~-17037) \SB\ DOS 3.3 - MOVE POINTER IN IOB TO ZERO PAGE. (SET DEVCTBL ($003C~$003D) AND BUFPTR ($003E~$003F) AND $0047 WITH $00D8 FROM DCT). CHECK IF DRIVE # HAS CHANGED. IF NOT BRANCH TO $BD74

$BD74~$BD8F (-17036~-17009) \SB\ DOS 3.3 - SELECT APPROPRIATE DRIVE AND SAVE DRIVE BEING USED AS HIGH BIT OF 'DRIVENO' ($0035). 1=DRIVE 1;0=DRIVE 2. IF DRIVE WAS ON BRANCH TO $BD90. IF NOT CALL 'MSWAIT' AT $BA00

$BD90~$BDAA (-17008~-16982) \SB\ DOS 3.3 - GET DESTINATION TRACK AND GO TO IT USING 'MYSEEK'($BE5A)

$BD90~$BDAA (-17008~-16982) \SB\ DOS 3.3 - GET DESTINATION TRACK AND GO TO IT VIA 'MYSEEK'($BE5A). CHECK TEST RESULT AGAIN AND IF DRIVE ON BRANCH TO 'TRYTRK' ($BDAB)

```
$BDAB~$BDB3 (-16981~-16965) [TRYTRK] \SB\DOS 3.3 - GET COMMAND CODE.  IF NULL EXIT VIA 'ALLDONE' ($BE46) TURNING OFF
                                     DRIVE & RETURNING TO CALLER. IF COMMAND CODE=4 BRANCH TO 'FORMDSK' ($BE0D);
                                     OTHERWISE MOVE LOW BIT INTO CARRY (SET=READ;CLEAR=WRITE) AND SAVE VALUE ON
                                     STATUS REG. IF WRITE OPN DATA IS PRENIBBILIZED VIA 'PRENIB16' ($B800)
$BDBC~$BDEC (-16964~-16916) [RDRIGHT] \SB\DOS 3.3 - INITIALIZE MAX RETRIES AT 48. READ ADDRESS FIELD VIA 'RDADR16'
                                     ($B944). IF GOOD READ BRANCH TO 'RDRIGHT' ($BDED). IF BAD TRY AGAIN
                                     DECREMENTING RETRIES. IF NONE LEFT PREPARE TO RECALIBRATE. DECREMENT RECAL
                                     COUNT. IF NO MORE THEN 'DRVERR' ($BE04). OTHERWISE RESET RESEEKS AT 4 AND
                                     RECALIBRATE ARM. TRY AGAIN
$BDED~$BE03 (-16915~-16893) [RDRIGHT] \SE\DOS 3.3 - VERIFY TRACK. IF CORRECT BRANCH TO 'RTTRK' ($BE10) OTHERWISE GOTO
                                     'SETTRK' ($BE95) AND DECREMENT RESEEK COUNT. IF ZERO RECAL OTHERWISE RESEEK TRACK
$BE04~$BE0A (-16892~-16886) [DRVERR] \SE\DOS 3.3 - CLEAN UP STACK & STATUS REG; LOAD A-REG WITH $40 (DRIVE ERROR) AND
                                     GOTO 'HNDLERR' ($BE48)
$BE0B~$BE0C (-16885~-16884)          DOS 3.3 - BRANCH TO 'ALLDONE' ($BE46)
$BE0D~$BF0F (-16883~-16625) [FORMDSK] DOS 3.3 - JUMP TO 'DSKFORM' ($BEAF)
$BE10~$BE25 (-16880~-16859) [RTTRK]  DOS 3.3 -CHECK VOL# FOUND VS VOL# WANTED. IF NO VOL SPECIFIED NO ERROR OTHERWISE
                                     IF MISMATCH LOAD A-REG WITH $20 (VOLUME MISMATCH ERROR) AND EXIT VIA 'HNDLERR'
                                     ($BE48)
$BE26~$BE45 (-16858~-16827) [CRCTVOL] DOS 3.3 - CHECK TO SEE IF SECTOR CORRECT. USE 'ILEAV' TABLE ($BFB8) FOR SOFTWARE
                                     SECTOR INTERLEAVING. IF WRONG SECTOR TRY AGAIN AT 'TRYADR ($BDC1). IF WRITE
                                     BRANCH TO 'WRIT' ($BE51). OTHERWISE GOTO 'READ16' ($B8DC). IF GOOD READ CALL
                                     'POSTN316' ($B8C2) AND RETURN TO CALLER WITH NO ERROR
$BE46~$BE47 (-16826~-16825) [ALLDONE] DOS 3.3 - SKIP OVER SET CARRY INSTRUCTION IN 'HNDLERR'
$BE48~$BE50 (-16824~-16816) [HNDLERR] DOS 3.3 - SET CARRY; STORE A-REG IN IOB AS RETURN CODE. TURN OFF MOTOR. RETURN TO
                                     CALLER
$BE51~$BE59 (-16815~-16807) [WRIT]   DOS 3.3 - WRITE A SECTOR USING 'WRITE16' ($B82A); IF GOOD WRITE EXIT VIA
                                     'ALLDONE' ($BE46) OTHERWISE LOAD A-REG WITH $10 (WRITE PROTECT ERROR) AND EXIT
                                     VIA 'HNDLERR' ($BE48)
$BE5A~$BE8D (-16806~-16755) [MYSEEK] DOS 3.3 - HOUSEKEEPING BEFORE 'SEEKABS'. DETERMINES NUMBER OF PHASES PER TRACK &
                                     STORES TRACK INFO IN APPROPRIATE SLOT-DEPENDENT LOCN
$BE8E~$BE94 (-16754~-16748) [XTOY]   DOS 3.3 - X-REG/16 =>Y-REG. USED TO PUT SLOT INTO Y-REG
$BE95~$BEAE (-16747~-16722) [SETTRK] DOS 3.3 - SET TRACK #
$BEAF~$BF0C (-16721~-16628) [DSKFORM] \SB\DOS 3.3 - INIT COMMAND HANDLER
$BF0D~$BF61 (-16627~-16543) \SB\     DOS 3.3 - TRACK WRITE ROUTINE
$BF62~$BF87 (-16542~-16505) \SB\     DOS 3.3 - VERIFY TRACK ROUTINE
$BF88~$BFA7 (-16504~-16473) \SB\     DOS 3.3 - SECTOR MAP ROUTINE - MARKS SECTOR INITIALIZATION MAP AS EACH SECTOR
                                     VERIFIED
$BFA8~BFB7                           DOS 3.3 - SECTOR INITIALIZATION MAP. CONTAINS $30 PRIOR TO INITIALIZATION OF
                                     TRACK. VALUE CHANGED TO $FF AS EACH SECTOR COMPLETED
$BFB8~$BFC7 (-16456~-16441) [ILEAV]  DOS 3.3 - SECTOR TRANSLATE TABLE. SECTOR INTERLEAVING DONE WITH SOFTWARE
$BFC8~$BFFF (-16440~-16385)          DOS 3.3 PATCH AREA
$BFC8~$BFD8 (-16440~-16424)          DOS 3.3 PATCH TO ZERO LANGUAGE CARD DURING BOOT
$BFD9~$BFDB (-16423~-16421)          DOS 3.3 - UNUSED
$BFDC~$BFE5 (-16420~-16411)          DOS 3.3 PATCH CALLED FROM $A032 TO SET ADDITIONAL DEFAULTS
$BFE6~$BFEC (-16410~-16404)          DOS 3.3 PATCH CALLED FROM ERROR HANDLER AT $A6D5. CALLS $A75B TO RESET STATE 0
                                     AND SET WARMSTART FLAG. MARK RUN NOT INTERRUPTED. RETURN TO CALLER
$BFED~$BFFF (-16403~-16385)          DOS 3.3 PATCH CALLED FROM DISK FULL ERROR EXIT ($B377). CALLS $AE7E TO SAVE FILE
                                     MANAGER WORK AREA; RESTORES STACK; CLOSES ALL OPEN FILES; SAVES STACK AGAIN;
                                     EXITS THRU $B385 ("DISK FULL ERROR")
$BFFF (-16385) \H\                   HIGHEST RAM MEMORY ADDRESS (FULL 48K APPLE) - NOTE: WITH LANGUAGE CARD SPECIAL
                                     RAM EXISTS HIGHER
```

```
$BFFF (-16385)                              DEFAULT INTEGER BASIC HIMEM (W/O DOS 3.2~ 48K MACHINE)
$C000~$CFFF (-16384~-12289) \HB\            ENTRY ADDRESSES DEDICATED TO I/O FUNCTIONS
$C000~$C00F (-16384~-16369) [M] \H1\        EQUIVALENT ADDRESSES - ALL FOR KEYBOARD INPUT BYTE. WHEN KEY PRESSED ASCII VALUE
                                            GOES THERE AND HIGH BIT SET
$C000~[KBD ~ IOADR] \H1\                    MONITOR I/O - PEEK TO READ KEYBOARD. IF VAL>127 KEY HAS BEEN PRESSED SINCE LAST
                                            STROBED AT $C010.
$C010~$C01F (-16368~-16353) \H1\            EQUIVALENT ADDRESSES - ALL CLEAR KEYBOARD STROBE  I.E. SET FLAG (HIGH) BET OF
                                            $C000 TO 0 (VAL<128) AND REACTIVATE KEYBOARD
$C010 (-16368) [KBDSTB] \H1\                KEYBOARD STROBE- REACTIVATES KEYBOARD SO THAT VALUE OF PRESSED KEY GOES TO $C000.
                                            SETS HIGH BITTO ZERO..-4
$C020~$C02F (-16352~-16337) [TAPEOUT] \H1\CASSETTE OUTPUT TOGGLE FLIP FLOP. READ ONLY DO NOT WRITE TO THESE ADDRESSES
                                            WHICH ARE DECODED AS SAME SINGLE BIT LOCN
$C020 (-16352) [TAPEOUT] \H1\               PEEK TO TOGGLE CASSETTE OUTPUT (CREATE A 'CLICK' ON RECORDING)
$C030~$C03F (-16336~-16321) [SPKR] \H1\SPEAKER TOGGLE FLIP FLOP. READ ONLY - DO NOT WRITE TO THES ADDRESSES WHICH ARE
                                            DECODED AS SAME SINGLE BIT LOCN
$C030 (-16336) [SPKR] \H1\                  PEEK TO TOGGLE SPEAKER (PRODUCES A 'CLICK')
$C040~$C04F (-16320~-16305) \H1\            UTILITY STROBE. IF READ PIN 5 ON GAME I/O CONNECTOR DROPS FROM 5 V TO 0 V FOR 1
                                            MICROSECOND
$C040~$C04F (-16320~-16305) \H1\            ANY ONE OF THESE 16 LOCATIONS HAS SAME EFFECT IF POKED. IT OUTPUTS STROBE TO GAME
                                            I/O CONNECTOR
$C050 (-16304) [TXTCLR] \H1\                POKE TO 0 TO SET FROM TEXT TO GRAPHICS MODE W/O CLEARING SCREEN
$C051 (-16303) [TXTSET] \H1\                POKE TO 0 TO SET FROM GRAPHICS TO TEXT MODE W/O RESETTING SCROLLING WINDOW
$C052 (-16302) [MIXCLR] \H1\                POKE TO 0 TO RESET FROM MIXED GRAPHICS (W/4 LINES TEXT) TO FULL-SCREEN GRAPHICS
$C053 (-16301) [MIXSET] \H1\                POKE=0 TO SET TEXT/GRAPHICS MIX (BOTTOM 4 LINES TEXT)
$C054 (-16300) [LOWSCR] \H1\                POKE TO 0 TO DISPLAY PAGE 1 (DOES NOT CLEAR SCREEN)
$C055 (-16299) [HISCR] \H1\                 POKE TO 0 TO DISPLAY PAGE 2  (DOES NOT CLEAR SCREEN)
$C056 (-16298) [LO-RES] \H1\                POKE TO 0 TO SET FROM HI-RES TO SAME PAGE # OF LO-RES OR TEXT
$C057 (-16297) [HI-RES] \H1\                POKE TO 0 TO SET TO HI-RES GRAPHICS FROM LO-RES OR TEXT (SAME PAGE)
$C058 (-16296) [SETAN0] \FF\                VALUE<>0 WHEN GAME AN0 IS SET. POKE 0 TO CLEAR GAME I/O OUTPUT AN0 (3.5V AT PIN
                                            15)
$C059 (-16295) [CLRAN0] \FF\                VALUE <>0 WHEN GAME AN0 IS RESET (CLEARED). POKE 0 TO SET GAME I/O OUTPUT AN0
                                            (0.3V AT PIN 15)
$C05A (-16294) [SETAN1] \FF\                POKE 0 TO CLEAR GAME I/O OUTPUT AN1 (3.5V AT PIN 14)
$C05B (-16293) [CLRAN1] \FF\                POKE 0 TO SET GAME I/O OUTPUT AN1 (0.3V AT PIN 14)
$C05C (-16292) [SETAN2] \FF\                POKE 0 TO CLEAR GAME I/O OUTPUT AN2 (3.5V AT PIN 13)
$C05D (-16291) [CLRAN2] \FF\                POKE 0 TO SET GAME I/O OUTPUT AN2 (0.3V AT PIN 13)
$C05E (-16290) [SETAN3] \FF\                POKE 0 TO CLEAR GAME I/O OUTPUT AN3 (3.5V AT PIN 12)
$C05F (-16289) [CLRAN3] \FF\                POKE 0 TO SET GAME I/O OUTPUT AN3 0.3V AT PIN 12)
$C060 (-16288) [TAPEIN]                     MONITOR MEMORY LOCATION 'TAPEIN'
$C060/8 (-16288) [TAPEIN] \H1\              STATE OF 'CASSETE DATA IN' APPEARS IN BIT 7
$C061 (-16287) \H1\                         PEEK TO READ PDL(0) PUSH BUTTON SWITCH. IF >127 SWITCH ON
$C062 (-16286) \H1\                         PEEK TO READ PDL(1) PUSH BUTTON SWITCH. IF >127 SWITCH ON
$C063 (-16285) \H1\                         PEEK TO READ PDL(2) PUSH BUTTON SWITCH. IF >127 SWITCH ON
$C064 (-16284) [PADDL0] \H1\                MONITOR MEMORY LOCATION PADDL0; HARDWARE INDISTINGUISHABLE FROM $C06C; STATE OF
                                            TIMER OUTPUT FOR PADDLE 0 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
$C065 (-16283) [PADDL1] \H1\                MONITOR MEMORY LOCATION PADDL1; HARDWARE INDISTINGUISHABLE FROM $C06D; STATE OF
                                            TIMER OUTPUT FOR PADDLE 1 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
$C066 (-16282) [PADDL2] \H1\                MONITOR MEMORY LOCATION PADDL2; HARDWARE INDISTINGUISHABLE FROM $C06E; STATE OF
                                            TIMER OUTPUT FOR PADDLE 2 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
$C067 (-16281) [PADDL3] \H1\                MONITOR MEMORY LOCATION PADDL3; HARDWARE INDISTINGUISHABLE FROM $C06F; STATE OF
                                            TIMER OUTPUT FOR PADDLE 3 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

$C06C (-16276) [PADDL0] \H1\          MONITOR MEMORY LOCATION PADDL0; HARDWARE INDISTINGUISHABLE FROM $C064; STATE OF
                                      TIMER OUTPUT FOR PADDLE 0 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
$C06D (-16275) [PADDL1] \H1\          MONITOR MEMORY LOCATION PADDL1; HARDWARE INDISTINGUISHABLE FROM $C065; STATE OF
                                      TIMER OUTPUT FOR PADDLE 1 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
$C06E (-16274) [PADDL2] \H1\          MONITOR MEMORY LOCATION PADDL2; HARDWARE INDISTINGUISHABLE FROM $C066; STATE OF
                                      TIMER OUTPUT FOR PADDLE 2 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
$C06F (-16273) [PADDL3] \H1\          MONITOR MEMORY LOCATION PADDL3; HARDWARE INDISTINGUISHABLE FROM $C067; STATE OF
                                      TIMER OUTPUT FOR PADDLE 3 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)
$C070~$C07F (-16272~-16257) [PTRIG] \H1\GAME CONTROLLER STROBE. WHEN READ CAUSES FALG INPUTS OF GAME CONTROLLERS TO GO
                                      OFF & TIMING LOOPS RESTARTED
$C070~$C07F (-16272~-16257) [PTRIG] \H1\ALL 16 ADDRESSES DECODE TO SINGLE SWITCH WHICH TRIGGERS PADDLE TIMERS DURING
                                      PHI-2
$C080~$C08F (-16256~-16241) [(DEV SELECT 0)] 16 MEMORY LOCNS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #0. WHEN
                                      ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED. SINCE SLOT #0 IS COMMON AREA
                                      USED IN COMMON FOR PARAMETERS OF INTEREST TO ALL SLOTS
$C080 (-16256) \H1\                   SELECT 2ND BANK OF$D000~$DFFF RAM IN LANGUAGE CARD. WRITE PROTECT RAM ($C084
                                      DECODES TO SAME ADDRESS & EFFECT)
$C080~$C081 (-16256~-16255) [PHSOFF~PHSON] \P4\DOS 3.2 READ\WRITE TRACK\SECTOR PACKAGE PARAMETER STATEMACHINE CONTROLS
                                      TABLE: LO LO=READ;HI LO=SENSE WRITE PROTECT;LO HI=WRITE;HI HI=WRITE LOAD
$C081 (-16255) \H1\                   READ-DESELECT 2ND BANK OF $D000~$DFFF RAM IN LANG. CARD (ENABLE ROM). TWO
                                      SUCCESSIVE READS WRITE-ENABLES RAM
$C081 (-16255) [PHASON] \P1\          DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'PHASON'
$C082 (-16254) \H1\                   READ-DESELECT 2ND BANK OF $D000~$DFFF RAM IN LANGUAGE CARD (ENABLE ROM). WRITE
                                      PROTECT RAM
$C082 (-16254) [PHSOFF] \P1\          DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'PHSOFF'
$C083 (-16253) \H1\                   SELECT 2ND BANK OF $D000~$DFFF RAM IN LANG. CARD. TWO SUCCESSIVE READS TO THIS
                                      ADDR WRITE-ENABLES RAM
$C084 (-16252) \H1\                   SELECT 2ND BANK OF$D000~$DFFF RAM IN LANGUAGE CARD. WRITE PROTECT RAM ($C080
                                      DECODES TO SAME ADDRESS & EFFECT)
$C085 (-16251) \H1\                   READ-DESELECT 2ND BANK OF $D000~$DFFF RAM IN LANG. CARD (ENABLE ROM). TWO
                                      SUCCESSIVE READS WRITE-ENABLES RAM
$C086 (-16250) \H1\                   READ-DESELECT 2ND BANK OF $D000~$DFFF RAM IN LANGUAGE CARD (ENABLE ROM). WRITE
                                      PROTECT RAM
$C087 (-16249) \H1\                   SELECT 2ND BANK OF $D000~$DFFF RAM IN LANG. CARD. TWO SUCCESSIVE READS TO THIS
                                      ADDR WRITE-ENABLES RAM
$C088 (-16248) \H1\                   SELECT 1ST BANK OF$D000~$DFFF RAM IN LANGUAGE CARD. WRITE PROTECT RAM
$C088 (-16248) [MOTOROFF] \P1\        DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'MOTOROFF'
$C089 (-16247) \H1\                   READ-DESELECT 1ST BANK OF $D000~$DFFF RAM IN LANG. CARD (ENABLE ROM). TWO
                                      SUCCESSIVE READS WRITE-ENABLES RAM
$C089 (-16247) [MOTORON] \P1\         DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'MOTORON'
$C08A (-16246) \H1\                   READ-DESELECT 1ST BANK OF $D000~$DFFF RAM IN LANGUAGE CARD (ENABLE ROM). WRITE
                                      PROTECT RAM
$C08A (-16246) [DRVOEN] \P1\          DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'DRVOEN' (DRIVE 0 ENABLE)
$C08B (-16245) \H1\                   SELECT 1ST BANK OF $D000~$DFFF RAM IN LANGUAGE CARD. TWO SUCCESSIVE READS TO THIS
                                      ADD WRITE-ENABLES RAM
$C08B (-16245) [DRV1EN] \P1\          DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'DRV1EN' (DRIVE 1 ENABLE)
$C08C~$C08D (-16244~-16243) [Q6L\Q6H] \P2\DOS 3.2 READ~WRITE TRACK\SECTOR PACKAGE PARAMETER 'Q6L~Q6H' (Q6 LOW CAUSES
                                      DOS 3.2 TO READ A BYTE)
$C08C (-16244) \H1\                   SELECT 1ST BANK OF$D000~$DFFF RAM IN LANGUAGE CARD. WRITE PROTECT RAM
$C08D (-16243) \H1\                   READ-DESELECT 1ST BANK OF $D000~$DFFF RAM IN LANG. CARD (ENABLE ROM). TWO
                                      SUCCESSIVE READS WRITE-ENABLES RAM
-------------------------------------------------------------------------------------------------------------------------------
$C06C - $C08D                    Prof. Luebbert's "What's Where in the Apple"                    NUMERIC ATLAS
```

$C08E~$C08F (-16242~-16241) [Q7L\Q7H] \P2\DOS 3.2 READ~WRITE TRACK\SECTOR PACKAGE PARAMETER 'Q7L~Q7H' (Q7 LOW SETS DOS 3.2 FOR READ MODE)

$C08E (-16242) \H1\ READ-DESELECT 1ST BANK OF $D000~$DFFF RAM IN LANGUAGE CARD (ENABLE ROM). WRITE PROTECT RAM

$C08F (-16241) \H1\ SELECT 1ST BANK OF $D000~$DFFF RAM IN LANGUAGE CARD. TWO SUCCESSIVE READS TO THIS ADD WRITE-ENABLES RAM

$C090~$C09F (-16240~-16225) [(DEV SELECT 1)] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #1. WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED

$C0A0~$C0AF (-16224~-16209) [(DEV SELECT 2)] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #2. WHEN ADDRESS PIN 41 TELLS DEVICE IT IS SELECTED

$C0B0~$C0BF (-16208~-16193) [(DEV SELECT 3)] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #3. WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED

$C0C0~$C0CF (-16192~-16177) [(DEV SELECT 4)] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #4. WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED

$C0D0~$C0DF (-16176~-16161) [(DEV SELECT 5)] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #5. WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED

$C0E0~$C0EF (-16160~-16145) [(DEV SELECT 6)] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #6. WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED

$C0E0~C0E7 EXAMPLE:DISK CONTROLLER IN SLOT 6 - ADDRESSES USED TO PULSE THE HEAD STEPPING MOTOR. (SEE PGS 145-146 IN DOS MANUAL FOR INFO ABOUT 4 CONTROL LINES TO STEPPING MOTOR) THESE ADDRESSES APPEAR IN 4 PAIRS WITH ODD ADDRESSES APPLYING VOLTAGE TO A LINE AND EVEN TURNING IT OFF AGAIN. IF REFERENCED IN DESCENDING ORDER HEAD STEPS TO A LOWER TRACK AND VICE-VERSA

$C0E8 (-16152) EXAMPLE: DISK CONTROLLER IN SLOT 6 - ENTRY ADDRESS TO POWER DOWN DISK

$C0E9 (-16151) EXAMPLE: DISK. NOTE: BASIC PROGRAMS CAN POKE TO THIS ADDRESS TO START THE MOTOR BEFORE ISSUING A DOS COMMAND AND GAIN A SLIGHT DECREASE IN ACCESS TIME CONTROLLER IN SLOT 6 - ENTRY ADDRESS TO POWER UP DISK. NOTE: BASIC PROGRAMS CAN POKE TO THIS ADDRESS TO START THE MOTOR BEFORE ISSUING A DOS COMMAND AND GAIN A SLIGHT DECREASE IN ACCESS TIME

$C0EA (-16150) EXAMPLE: DISK CONTROLLER IN SLOT 6 - SELECT DISK DRIVE #1

$C0EB (-16149) EXAMPLE: DISK CONTROLLER IN SLOT 6 - SELECT DISK DRIVE #2

$C0EC~$C0EF (-16148~-16145) EXAMPLE: DISK CONTROLLER IN SLOT 6 - 4 BYTES TO DETERMINE WHETHER DISK CONTROLLER IS TO READ~ WRITE OR RETURN THE STATUS OF THE WRITE PROTECT MICROSWITCH. ALSO USED FOR PASSING READ/WRITE DATA IN GROUPS OF 4-BIT NIBBLES

$C0F0~$C0FF (-16144~-16129) [(DEV SELECT 7)] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #7. WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED

$CS00~$CSFF 256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE IN SLOT #S. PIN 1 DROPS WHEN ADDRESS SELECTED

$CS00\SE\ EXAMPLE: CALL -16384+256*S TO TRANSMIT ASCII CHAR IN ACCUMULATOR OUT VIA APPLE SERIAL INTERFACE IN SLOT S

$C100~$C1FF (-16128~-15873) 256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE #1. PIN 1 DROPS WHEN ADDRESS SELECTED

$C100 (-16128) \SE\ STANDARD CHARACTER I/O SUBROUTINE ENTRY POINT FOR SLOT #1

$C100 (-16128) \SE\ EXAMPLE: JSR $C100 OR CALL -16128 IS EQUIVALENT TO PR#1 FOR INITIALIZING APPLE SERIAL INTERFACE IN SLOT #1

$C200~$C2FF (-15872~-15617) 256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE #2. PIN 1 DROPS WHEN ADDRESS SELECTED

$C200 (-15872) \SE\ STANDARD CHARACTER I/O SUBROUTINE ENTRY POINT FOR SLOT #2

$C300~$C3FF (-15616~-15361) 256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE #3. PIN 1 DROPS WHEN ADDRESS SELECTED

$C300 (-15616) \SE\ STANDARD CHARACTER I/O SUBROUTINE ENTRY POINT FOR SLOT #3

$C400~$C4FF (-15360~-15105)	256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE #4. PIN 1 DROPS WHEN ADDRESS SELECTED
$C400 (-15360) \SE\	STANDARD CHARACTER I/O SUBROUTINE ENTRY POINT FOR SLOT #4
$C500~$C5FF (-15104~-14849)	256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE #5. PIN 1 DROPS WHEN ADDRESS SELECTED
$C500 (-15104) \SE\	STANDARD CHARACTER I/O SUBROUTINE ENTRY POINT FOR SLOT #5
$C600~$C6FF (-14848~-14593)	256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE #6. PIN 1 DROPS WHEN ADDRESS SELECTED
$C600~$C6FF (-14848~-14593)	256 BYTE PAGE OF MEMORY USED BY DOS 3.2/3.3 IF DISK CONTROLLING IN STANDARD SLOT #6 (MEMORY PHYSICALLY ON CONTROLLER BOARD). PART OF THIS INFO IS TRANSFERED TO PAGE 3 ($300) ON BOOTING
$C600~$C65B (-14848~-14757)	DOS 3.2/3.3 - THIS CODE FROM DISK II CONTROLLER ROM IS FIRST CODE EXECUTED WHEN A DISK IS TO BE BOOTED. DYNAMICALLY BUILDS A TRANSLATE TABLE FOR CONVERTING DISK CODES TO 6 BIT HEX AT $0356~$03FF AND DOES INITIAL HOUSEKEEPING AND SETS UP TO READ SECTOR ZERO TRACK ZERO TO $0800 THEN FALLS THRU TO GENERAL SECTOR READ SR AT $C65C
$C600 (-14848) \SE\	STANDARD CHARACTER I/O SUBROUTINE ENTRY POINT FOR SLOT #6
$C65C~$C6FA (-14756~-14598)	DOS 3.3 GENERAL SECTOR READ ROUTINE. USES SECTOR # AT $3D ON THE TRACK INDICATED BY $0041. READS TO ADDRESS SPECIFIED AT $0026~$0027. IF D5/AA/AD FOUND ON SECTOR ADDRESS HEADER & SECTOR DATA WANTED GOTO $C6A6
$C683 (-14717)	DOS 3.3 S/R TO HANDLE SECTOR ADDRESS BLOCK. READS 3 DOUBLE BYTES AND COMBINE TO FORM VOLUME~ TRACK & SECTOR. STORE TRACK AT $0040. IF DESIRED SECTOR FOUND GOTO $C65D TO GET SECTOR DATA; OTHERWISE RETURN TO $C65C
$C6A6 (-14682)	DOS 3.3 SECTOR DATA HANDLING BLOCK. READS 85 BYTES OF SECONDARY DATA TO $0300-$0355 AND READS 256 BYTES OF PRIMARY DATA TO ADDRESS SPECIFIED BY $0026~$0027 & 'NIBBLIZE'. INCREMENT $0027 & $003D AND CHECK AGAINST $0800 TO SEE IF ADDITIONAL SECTORS TO BE READ
$C700~$C7FF (-14592~-14337)	256 BYTE PAGE OF MEMORY (USUALLY ROM) ALLOCATED TO PERIPHERAL DEVICE #7. PIN 1 DROPS WHEN ADDRESS SELECTED
$C700 (-14592) \SE\	STANDARD CHARACTER I/O SUBROUTINE ENTRY POINT FOR SLOT #7
$C800~$CFFF (-14336~-12289)	EXPANSION ROM MEMORY SPACE. RESERVED FOR 2K ROMS ON PERIPHERAL CARDS. ROM IS ACTIVE (ADRESSABLE) ONLY WHEN SLOT IS ACTIVE
$C800~$CFFF (-14336~-12289)	PIN 20 ON ALL PERIPH CONCTRS GOES LOW DURING PHI0 ON READ OR WRITE TO THIS GP
$C93D (-14019) \SE\	SERIAL INTERFACE BATCH INPUT ROUTINE. A1&A2 SPECIFY MEMORY RANGE
$C941 (-14015) \SE\	SERIAL INTERFACE BATCH OUTPUT ROUTINE - A1 & A2 SPECIFY MEMORY RANGE
$CFFF (-12289) [CLRROM] \H1\	SPECIAL LOCATION RECOGNIZED BY PERIPHERAL CARDS AS SIGNAL TO TURN OFF FLIP FLOPS WHICH DISABLE EXPANSION ROM
$D000~$DFFF (-12288~-8193) \HB\	LANGUAGE CARD CONTAINS TWO SWITCHABLE BANKS OF RAM MEMORY WHICH SHARE THIS ADDRESS SPACE
$D000~$D7FF (-12288~-10241) \HB\	ROM SOCKET D0
$D000~$D3FF (-12288~-11265) \HB\	PROGRAMMERS AID #1 (HI-RES GRAPHICS ROM)
$D000 (-12288) [SETHRL] \SE\	HI-RES GRAPHICS INIT S/R CALL (ROM VERSION)
$D00E (-12274) [HCLR] \SE\	HI-RES GRAPHICS CLEAR S/R CALL
$D010 (-12272) [BKGND0]	HI-RES GRAPHICS 'BKGND0 (HCOLOR1 SET FOR BLACK BKGND)
$D012 (-12270) [BKGND] \P1\	HI-RES GRAPHICS MEMORY LOCATION 'BKGND' (ROM)
$D1FC (-11780) [HFIND] \SE\	HI-RES GRAPHICS FIND S/R CALL: PARAM=SHAPE~ROT~SCALE
$D2F9 (-11527) [BPOSN] \SE\	HI-RES GRAPHICS POSN S/R CALL PARAM= X0~Y0~COLR
$D30E (-11506) [BPLOT] \SE\	HI-RES GRAPHICS PLOT S/R CALL PARAM= X0~Y0~COLR
$D314 (-11500) [BLIN1] \SE\	HI-RES GRAPHICS LINE S/R CALL PARAM= X0~Y0~COLR
$D331 (-11471) [BGND] \SE\	HI-RES GRAPHICS BKGND S/R CALL PARAM= COLR
$D337 (-11465) [BDRAW1] \SE\	HI-RES GRAPHICS LINE S/R CALL: PARAM=X0~Y0~COLR

```
$D33A (-11462) [BDRAW] \SE\          HI-RES GRAPHICS  DRAW1 S/R CALL: PARAM= XO~YO~COLR~SHAPE~ROT~SCALE
$D393 (-11373) [BLTU] \SE\           APPLESOFT BLOCK TRANSFER UTILITY. MAKES ROOM BY MOVING EVERYTHING FORWARD.
                                     Y-REG(MSB)&A-REG(LSB) AND HIGHDS=DEST OF HIGH ADR;LOWTR=LOWEST ADDR TO BE
                                     MOVED;HIGHTR=HIGHEST ADDR TO BE MOVED+1
$D3E3 (-11293) [REASON] \SE\         CHECKS FOR ENOUGH ROOM IN MEMORY; CHECKS THAT ADDR Y-REG(MSB)&A-REG(LSB) LESS
                                     THAN FRETOP. MAY CAUSE GARBAGE COLLECTION. CAUSE OMERR IF NO ROOM
$D3B9 (-11335) [SHLOAD] \SE\         HI-RES GRAPHICS SHLOAD S/R CALL
$D410 (-11248) [(OUT OF MEM PRT)]    APPLESOFT - PRINT "OUT OF MEMORY" THEN HALT AT APPLESOFT (]) LEVEL
$D412 (-11246) [ERROR] \SE\          APPLESOFT ERROR PROCESSING - CHECKS ERRFLG AND JUMPS TO HNDLERR IF ONERR IS
                                     ACTIVE OTHERWISE PRINTS ERROR MSG BASED ON CODE IN X-REG
$D43C (-11204) \SE\                  APPLESOFT LOCATION TO WHICH DOS 3.2 JUMPS TO MAKE A SOFT ENTRY TO ROM (OR
                                     LANGUAGE PACK) APPLESOFT
$D4BC (-11076)                       INTEGER BASIC PA#1 APPEND PROGRAM ENTRY
$D4F2 (-11022) \SE\                  APPLESOFT - SET (OR RESET) POINTERS & LINKAGES FOR FIRMWARE APPLESOFT (ROM) (OR
                                     LANGUAGE PACK LOCATED IN TOP 16K OF 64K MEMORY)
$D4F2 (-11022) \SE\                  APPLESOFT - TO CONVERT  FROM RAM APPLESOFT STORED AT $0800-$3003 TO FIRMWARE
                                     APPLESOFT IN ROM OR TOP 16K RAM- ~CALL -11022~LIST~SAVE
$D52C (-10964) [INLIN] \SE\          APPLESOFT - INPUT LINE OF TEXT FROM CURRENT INPUT DEVICE INTO INPUT BUFFER (BUF)
                                     & FALL INTO GDBUFS. NO PROMPT!
$D52E (-10962) [INLIN+2] \SE\        APPLESOFT - INPUT LINE OF TEXT FROM CURRENT INPUT DEVICE INTO INPUT BUFFER (BUF)
                                     & FALL INTO GDBUFS. CHAR IN X-REG USED AS PROMPT
$D535 (-10955)                       INTEGER BASIC PA#1 TAPE VERIFY PROG ENTRY
$D539 (-10951) [GDBUFS] \SE\         APPLESOFT - PUT ZERO AT END OF INPUT BUFFER (BUF) AND MASK OFF MOST SIGNIFICANT
                                     BIT ON ALL BYTES. ON ENTRY X-REG=END OF INPUT LINE (A- X- Y-REGS ALTERED)
$D553 (-10925) [INCHR] \SE\          APPLESOFT - GET ONE CHAR FROM CURRENT INPUT DEVICE IN A-REG & MASK OF MSB. USES
                                     MAIN APPLE INPUT ROUTINES & SUPPORTS HANDSHAKING
$D566 (-10906) [RUN] \SE\            APPLESOFT - RUN THE PROGRAM IN MEMORY. THIS ROUTINE DOES NOT RETURN
$D61A (-10726) [FNDLIN] \SE\         APPLESOFT - SEARCHES PROGRAM FOR LINE WHOSE NUMBER IS IN LINNUM. ON EXIT IF CARRY
                                     SET LOWTR POINTS TO LINK FIELD OF DESIRED LINE; IF NOT LOWTR TO NEXT HIGHER LINE
$D64B (-10677) [SCRTCH] \SE\         APPLESOFT INITIALIZATION - THE 'NEW' COMMAND. CLEARS PROGRAM VARIABLES & STACK
$D66C (-10644) [CLEARC] \SE\         APPLESOFT INITIALIZATION - THE 'CLEAR' COMMAND. CLEARS VARIABLES & STACK
$D683 (-10621) [STKINI] \SE\         APPLESOFT STACK INITIALIZATION - CLEARS THE STACK
$D697 (-10601) [STXTPT] \SE\         APPLESOFT INITIALIZATION - SET TXTPTR TO BEGINNING OF PROGRAM
$D6DD (-10531) \SE\                  INTEGER BASIC PA#1 RENUMBER PROG ENTRY (WHOLE PROG)
$D6E7 (-10521) \SE\                  INTEGER BASIC PA#1 RENUMBER PROG ENTRY (PART PROG)
$D717 (-10473) \SE\                  INTEGER BASIC PA#1 MUSIC PROG ENTRY
$D7D2 (-10286) [NEWSTT] \SE\         APPLESOFT - EXECUTE A NEW STATEMENT. ON ENTRY TXTPTR POINTS TO THE':' PRECEDING
                                     THE STMT OR ZERO AT END OF PREVIOUS LIN. USE NEWSTT TO RESTART THE PROGRAM WITH
                                     CONT. THIS ROUTINE DOES NOT RETURN
$D800~$DFFF (-10240~-8193) \HB\      ROM SOCKET D8
$D849 (-10167) [RESTOR] \SE\         APPLESOFT RESTORE FUNCTION - SET DATA POINTER (DATPTR) TO BEGINNING OF THE PROGRAM
$D858 (-10152) [ISCNTC] \SE\         APPLESOFT - CHECK KEYBOARD FOR CONTROL-C ($83). EXECUTES BREAK ROUTINE IF THESE IS
$D865 (-10139) \SE\                  APPLESOFT - POINT TO WHICH DOS 3.2 JUMPS INTO ROM APPLESOFT WHEN PROCESSING ERRORS
$D898 (-10088) [CONT] \SE\           APPLESOFT - MOVES OLDTXT & OLDLIN INTO TXTPTR & CURLIN
$D8B0 (-10064) [SAVE] \SE\           APPLESOFT CASSETTE - SAVE THE PROGRAM IN MEMORY TO CASSETTE TAPE
$D8C9 (-10039) [LOAD] \SE\           APPLESOFT CASSETTE - LOAD A PROGRAM FROM CASSETTE TAPE
$D8F0 (-10000) [VARTIO] \SE\         APPLESOFT CASSETTE - SET UP A1 & A2 TO SAVE 3 BYTES ($0050~$0052) FOR LENGTH
$D901 (-9983) [PROGIO] \SE\          APPLESOFT CASSETTE - SET UP A1 & A2 TO SAVE PROGRAM TEXT ON CASSETTE
$D93E (-9922) [GOTO] \SE\            APPLESOFT - USES LINGET & FNDLIN TO UPDATE TXTPTR. GOTO ASSUMES 6502 REGS HAVE
                                     BEEN SET UP BY CHRGET THAT FETCHED 1ST DIGIT
$D979 (-9863) [(RET W/O GOSUB)]      APPLESOFT - PRINT "RETURN WITHOUT GOSUB" THEN HALT AT APPLESOFT (]) LEVEL
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------

$D97C (-9860) [(UNDEF'D STMT PRT)]   APPLESOFT - PRINT "UNDEF'D STATEMENT" THEN HALT AT APPLESOFT (]) LEVEL
$D995 (-9835) [DATA] \SE\            APPLESOFT - MOVE TXTPTR TO END OF STATEMENT; LOOKS FOR ':' OR EOL(0).
$D9A3 (-9821) [DATAN] \SE\           APPLESOFT - CALCULATE OFFSET IN Y-REG FROM TXTPTR TO NEXT ':' OR EOL(0)
$D9A6 (-9818) [REMN] \SE\            APPLESOFT - CALCULATE OFFSET IN Y-REG FROM TXTPTR TO NEXT COL(0)
$D998 (-9832) [ADDON] \SE\           APPLESOFT - ADD Y-REG TO TXTPTR
$DA0C (-9716) [LINGET] \SE\          READ 16BIT INTEGER LINE # FROM TXTPTR INTO LINNUM. SEE APPLE ORCHARD V1#1P13 FOR
                                     DETAILS
$DA46 (-9658) [LET] \SE\             APPLESOFT LET - USES CHRGET TO GET ADDRESS OF '=';EVALUATES FORMULA & STORES IT.
                                     ON ENTRY TXTPTR POINTS TO FIRST CHAR OF VARIABLE NAME
$DA65 (-9627) \SE\                   APPLESOFT - PACK EXTENSION BYTE IN FAC AND CONVERT FAC (WHERE |FAC|<2^15) TO
                                     2-BYTE INTEGER. STORE INTEGER IN FORPNT ($0085^$0086) (Y-REG=>1)
$DAFB (-9477) [CRDO] \SE\            APPLESOFT - PRINT A CARRIAGE RETURN
$DAB7 (-9545) [COPY] \SE\            APPLESOFT - FREE STRING POINTED TO BY Y-REG (MSB) & A-REG (LSB) & MOVE IT TO MEM
                                     LOC POINTED TO BY FORPNT
$DB3A (-9414) [STROUT] \SE\          APPLESOFT - PRINT STRING POINTED TO BY Y-REG (MSB) & A-REG (LSB). STRING MUST END
                                     WITH A ZERO OR QUOTE
$DB3D (-9411) [STRPRT] \SE\          APPLESOFT - PRINT A STRING WHOSE DESCRIPTOR IS POINTED TO BY FACMO^FACLO
$DB5C (-9380) [OUTDO] \SE\           APPLESOFT - PRINT THE CHARACTER IN A-REG. INVERSE^FLASH^NORMAL OPTIONS IN EFFECT
$DB57 (-9385) [OUTSPC] \SE\          APPLESOFT - PRINT A SPACE
$DB5A (-9382) [OUTQST] \SE\          APPLESOFT - PRINT A QUESTION MARK
$DD0B (-8949) [(NEXT W/O FOR PRT)] \SE\APPLESOFT - PRINT ERROR MESSAGE "NEXT WITHOUT FOR" THEN HALT AT APPLESOFT (])
                                     LEVEL
$DD67 (-8857) [FRMNUM] \SE\          APPLESOFT - EVALUATE EXPRESSION POINTED TOBY TXTPTR ($00B8^$00B9) (POINTS TO 1ST
                                     CHAR OF FORMULA). PUT RESULT INTO FAC & MAKE SURE IT IS A NUMBER
$DD6A (-8854) [CHKNUM] \SE\          APPLESOFT - MAKE SURE FAC IS NUMERIC (SEE CHKVAL)
$DD6C (-8852) [CHKSTR] \SE\          APPLESOFT - MAKE SURE FAC IS STRING (SEE CHKVAL)
$DD6D (-8851) [CHKVAL] \SE\          APPLESOFT - IF C SET CHECK FOR STRINGS;C CLEAR CHECK FOR NUMRIC VBL. TYPE MISMATCH
                                     ERROR OCCURS IF C AND FAC DON'T AGREE
$DD76 (-8842) \SE\                   APPLESOFT - PRINT "TYPE MISMATCH" THEN HALT AT APPLESOFT (]) LEVEL
$DD7B (-8837) [FRMEVL] \SE\          APPLESOFT - EVAL FORMULA AT TXTPTR USING CHRGET & LEAVE RESULT IN FAC. ON ENTRY
                                     TXTPTR POINTS TO 1ST CHAR OF FORMULA
$DD7B (-8837) [FRMEVL] \SE\          APPLESOFT - EVAL FORMULA AT TXTPTR USING CHRGET. IF FORMULA IS STRING LITERAL
                                     FRMEVL GOBBLES OPENING QUOTE AND EXECUTES STRLIT & ST2TXT
$DE10 (-8688) \SE\                   APPLESOFT - PACK EXTENSION BYTE OF FAC INTO FAC & PUSH FAC ONTO STACK (6 BYTES).
                                     MODIFIES INDEX
$DE47 (-8633) \SE\                   APPLESOFT - PULL ARG AND PUT EXCLUSIVE OR OF SIGNS OF FAC & ARG INTO (XORFPSGN)
                                     $00AB. MUST BE EXECUTED BY JMP INSTRUCTION
$DE81 (-8575) [STRTXT] \SE\          APPLESOFT - SET Y-REG (MSB) & X-REG(LSB) TO TXTPTR + CARRY BIT AND FALL INTO STRLIT
$DE98 (-8552) [(NOTFAC)] \SE\        APPLESOFT - LET FAC = NOT(FAC); I.E. RETURNS FAC=1 IF FAC=0 OR FAC=0 IF FAC<>0
$DEB2 (-8526) [PARCHK] \SE\          APPLESOFT PARENTHESIS CHECK - CHECK FOR '(';EVALUATE FORMULA;CHECK FOR ')'. USES
                                     CHKOPN & FRMEVL THEN FALLS INTO CHKCLS
$DEB8 (-8520) [CHKCLS] \SE\          APPLESOFT CLOSE PARENTHESIS CHECK - CHECKS TXTPTR FOR ')'. USES SYNCHR.
$DEBB (-8517) [CHKOPN] \SE\          APPLESOFT OPEN PARENTHESIS CHECK - CHECKS TXTPTR FOR '('. USES SYNCHR.
$DEBE (-8514) [CHKCOM] \SE\          APPLESOFT COMMA CHECK - CHECKS TXTPTR FOR COMMA. USES SYNCHR.
$DEC0 (-8512) [SYNCHR] \SE\          APPLESOFT SYNTAX CHARACTER CHECK - CHECKS TO VERIFY TXTPTR POINTS TO SAME
                                     CHARACTER AS THAT IN A-REG. NORMAL EXIT THRU CHGET TO GET NEX CHAR FROM INPUT
                                     BUFFER OTHEWISE SYNTAX ERROR. TXTPTR NOT MODIFIED. (Y-REG RESET TO ZERO)
$DEC9 (-8503) \SE\                   SNERR S/R. PRINTS "SYNTAX ERROR" AND HALTS PROG
$DEE9 (-8471) [(INT=>FP)] \SE\       APPLESOFT - PULL INTEGER (%) VARIABLE POINTED TO BY FACMO^FACLO ($00A0^$00A1) INTO
                                     A-REG & Y-REG AND CONVERT TO FP IN FAC. RESETS VALTYP (RESETS Y-REG TO 0)
$DF4F (-8369) [(FAC/ARG OR)] \SE\    APPLESOFT - LET FAC = FAC 'OR' ARG; I.E. FAC=1 IF EITHER FAC OR ARG OR BOTH <>0;
                                     FAC=0 ONLY IF BOTH FAC & ARG = 0

-------------------------------------------------------------------------------------------------------------------
$D97C - $DF4F                        Prof. Luebbert's "What's Where in the Apple"                    NUMERIC ATLAS
```

$DF55 (-8363) [(FAC/ARG AND)] \SE\APPLESOFT - LET FAC = FAC 'AND' ARG; I.E. FAC=1 ONLY IF BOTH FAC & ARG <>0; IF EITHER FAC OR ARG OR BOTH =0 THEN FAC=0
$DF6A (-8342) [(FAC/ARG COMPARE)] \SE\APPLESOFT - COMPARERS FAC WITH ARG. TYPE OF COMPARISON CONTROLLED BY $0016. IF CONDITION MET FAC SET TO ONE; ELSE FAC RESET TO ZERO
$DFE3 (-8221) [PTRGET] \SE\APPLESOFT - READ VAR NAME FROM CHRGET AND FIND IT IN MEMORY (OR CREATE APPROPRIATE SIMPLE VARIABLE OR ARRAY). DOES MUCH HOUSEKEEPING
$E000~$E7FF (-8192~-6145) \HB\ ROM SOCKET E0 (APPLE II (NOT II+) = INTEGER BASIC)
$E000 (-8192) [BASIC] INTEGER BASIC - 'HARD' OR 'COLD' OR 'CONTROL-B' ENTRY POINT (COMPLETE REINITIALIZATION. START WITH A TOTALLY FRESH SLATE)
$E000 (-8192) [BASIC] APPLESOFT - 'HARD' OR 'COLD' OR 'CONTROL-B' ENTRY POINT (COMPLETE REINITIALIZATION. START WITH A TOTALLY FRESH SLATE.)
$E003 (-8189) [BASIC2] \SE\ INTEGER BASIC - 'SOFT' OR 'WARM' OR 'CONTROL-C' OR 'ENTRY2' ENTRY POINT (REENTRY WITHOUT REINITIALIZATION OF SYMBOL-TABLE~ VARIABLES OR DATA)
$E006 (-8186) [~SETPRMPT~] \SE\ INTEGER BASIC ENTRY POINT TO SET UP '>' PROMPT
$E02A (-8150) [~NXTBYTE~] \SE\ INTEGER BASIC ENTRY POINT TO GET NEXT BYTE 16-BIT POINTER
$E04B (-8117) \SE\ INTEGER BASIC 'LIST' ROUTINE (LIST ALL THE PROGRAM)
$E05D (-8099) \SE\ INTEGER BASIC ENTRY POINT TO LIST X~Y (LIST A RANGE OF THE PROGRAM
$E06D (-8083) [~UNPACK~] \SE\ INTEGER BASIC ENTRY POINT TO UNPACK TOKENED CODE TO MNEMONICS
$E07D (-8067) [ISLETC (CHARCHEK)] \SE\APPLESOFT - CHECKS A-REG FOR ASCII LETTER OTHERWISE CLEAR IT TO ZERO ('A' TO 'Z'). SET C (CARRY FLAG) TO 1 IF A IS A LETTER OTHERWISE CLEAR IT TO ZERO (A- X- Y-REGS NOT ALTERED)
$E0FE~$E104 (-7938~-7932) [(-32K)] \P5\APPLESOFT FIVE-BYTE FLOATING POINT CONSTANT -32768 (-2~16)
$E105 (-7931) [(EVAL EXPR =>INT)] \SE\APPLESOFT - EVALUATE EXPRESSION POINTED TO BY TXTPTR ($00B8~$00B9) AND CONVERT RESULT (WHICH MUST BE NON-NEGATIVE) TOA TWO-BYTE INTEGER IN FACMO~FACLO ($00A0~$00A1)
$E108 (-7928) [(AYPOSINT +FP=>INT)] \SE\APPLESOFT - SAME AS AYINT ($E10C) EXCEPT FAC MUST BE POSITIVE
$E10C (-7924) [AYINT (FP=>INT)] \SE\APPLESOFT - IF FAC SUITABLE FOR CONVERSION TO INTEGER (FAC<32767 & FAC>-32768) THEN PERFORM QINT (RESET Y-REG=0)
$E130 (-7888) [~DIMSTR~] \SE\ INTEGER BASIC ENTRY POINT TO DIMENSION A STRING FOR MEMORY
$E171 (-7823) [~INPUTSTR~] \SE\ INTEGER BASIC ENTRY POINT TO 'INPUT A STRING' ROUTINE
$E196 (-7786) [(BAD SUBSCRPRT)] \SE\APPLESOFT - PRINT "BAD SUBSCRIPT" AND HALT AT APPLESOFT LEVEL ())
$E199 (-7783) [(ILLEGAL QTY PRT)] \SE\APPLESOFT - PRINT "ILLEGAL QUANTITY" AND HALT AT APPLESOFT LEVEL ())
$E222 (-7646) [~MULT~] \SE\ INTEGER BASIC ENTRY POINT TO MULTIPLY ROUTINE
$E27A (-7558) [~MOD~] \SE\ INTEGER BASIC ENTRY POINT TO MODULO FUNCTION
$E28A (-7542) [~SCRN~] \SE\ INTEGER BASIC ENTRY POINT TO SCREEN X~ Y~ COLOR VALUE FUNCTION
$E2B3 (-7501) [~MAINLINE~] \SE\ INTEGER BASIC ENTRY POINT TO MAIN LINE OF COMPILE/EXECUTE CODE
$E2F2 (-7438) [GIVAYF (INT=>FP)] \SE\APPLESOFT - FLOAT THE SIGNED INTEGER W/ LSB IN A-REG MSB IN Y-REG INTO FAC. RESETS VALTYP. (RESETS Y-REG=0)
$E301 (-7423) [SNGFLT] \SE\ APPLESOFT - FLOAT THE UNSIGNED INTEGER IN Y-REG INTO FAC. RESETS VALTYP. (RESET Y-REG=0)
$E306 (-7418) [ERRDIR] \SE\ APPLESOFT - CAUSES ILLEGAL DIRECT ERROR IF PROGRAM NOT RUNNING (X-REG ALTERED)
$E30B (-7413) [(ILLDIRPRT)] \SE\ PRINT "ILLEGAL DIRECT" THEN HALT AT APPLESOFT ()) LEVEL
$E30E (-7410) \SE\ APPLESOFT - PRINT "UNDEFINED FUNCTION" THEN HALT AT APPLESOFT ()) LEVEL
$E36B (-7317) [MEMFUL] \SE\ INTEGER BASIC MEMORY FULL ERROR
$E36F (-7313) [~DELETE~] \SE\ INTEGER BASIC ENTRY POINT TO DELETE LINES OF TEXT X~Y
$E3C0 (-7232) [~ERRORMESS*~] \SE\ INTEGER BASIC ENTRY POINT - INPUT ERROR MESSAGE
$E3CE (-7218) [~GETCMD~] \SE\ INTEGER BASIC ENTRY POINT TO GET A COMMAND FROM THE KEYBOARD
$E3D5 (-7211) [STRINI] \SE\ APPLESOFT - GET SPACE FOR CREATION OF A STRING & CREATE DISCRIPTOR FOR IT IN DSCTMP. ON ENTRY A-REG = LEN OF STRING.
$E3DD (-7203) [STRSPA] \SE\ APPLESOFT - JSR TO GETSPA. STORE THE POINTER & LENGTH IN DSCTMP.
$E3E0 (-7200) [~ERRORMESS~] \SE\ INTEGER BASIC ENTRY POINT TO PRINT ERROR MESSAGE AND GOTO MAINLINE

$E3E3 (-7197) \SE\ INTEGER BASIC ENTRY POINT TO WHICH DOS 3.2 CHAINS WHEN PROCESSING ERRORS
$E3E7 (-7193) [STRLIT] \SE\ APPLESOFT - STORE A QUOTE IN ENDCHR AND CHARAC SO THAT STRLT2 WILL STOP ON IT
$E3ED (-7187) [STRLT2] \SE\ APPLESOFT - BUILD DESCRIPTOR FOR STRING LITERAL WHOSE 1ST CHAR POINTED TO BY Y-REG (MSB) & X-REG (LSB). PUT INTO TEMPORARY & POINTER TO IT IN FACMO~FACLO.
$E42A (-7126) [PUTNEW] \SE\ APPLESOFT - STRING FUNCTION RETURNING WITH RESULT INDSCTMP. MOVE DSCTMP TO TEMP DESCRIPTOR & PUT POINTER TO DESCRIPTOR IN FACMO~FACLO & FLAG RESULT AS STRING
$E430 (-7120) [(TOOCOMPLEX)] \SE\ APPLESOFT - PRINT "FORMULA TOO COMPLEX" THEN HALT AT APPLESOFT (]) LEVEL
$E452 (-7086) [GETSPA] \SE\ APPLESOFT - GET SPACE FOR CHARACTER STRING. MOVES FRESPC & FRETOP DOWN. A-REG = # OF CHARS. POINTER TO SPC IN Y-REG(MSB) & X-REG(LSB)
$E484 (-7036) [GARBAG] \SE\ APPLESOFT GARBAGE COLLECTOR - MOVES ALL CURRENTLY USED STRINGS UP IN MEMORY AS FAR AS POSSIBLE
$E51B (-6885) [~HEX/DEC~] \SE\ INTEGER BASIC - DECIMAL LPRINT (LINE NUMBER PRINT) S/R; CONVERTS 2-BYTE (16-BIT) BINARY/HEX TO UNSIGNED DECIMAL (0-65535)
$E597 (-6761) [CAT] \SE\ APPLESOFT - CONCATENATE TWO STRINGS. FACMO (MSB) & FACLO (LSB) POINT TO FIRST STRING'S DESCRIPTOR & TXTPTR POINTS TO '+'
$E5AD (-6739) [~NEW~] \SE\ INTEGER BASIC ENTRY POINT TO CLEAR OUT OLD PROGRAM AND RESET POINTERS FOR A NEW PROGRAM
$E5B7 (-6729) [~CLR~] \SE\ INTEGER BASIC ENTRY POINT TO CLEAR OUT VARIABLE WORK SPACE
$E5D4 (-6700) [MOVINS] \SE\ APPLESOFT - MOVE STRING WHOSE DESCRIPTOR IS POINTED TO BY STRNG1 TO MEM LOC POINTED TO BY FORPNT
$E5E2 (-6686) [MOVSTR] \SE\ APPLESOFT - MOVE STRING POINTED TO BY Y-REG (MSB) & X-REG (LSB) WITH LENGH IN A-REG TO MEMORY POINTED TO BY FRESPA
$E5FD (-6659) [FRESTR] \SE\ APPLESOFT - MAKE SURE THAT LAST FAC RESULT WAS A STRING & FALL INTO FREFAC
$E604 (-6652) [FRETMP] \SE\ APPLESOFT - FREE A TEMPORARY STRING. ON ENTRY POINTER TO DESCRIPTOR IS IN Y-REG (MSB) & X-REG (LSB)
$E635 (-6603) [FRETMS] \SE\ APPLESOFT - FREE TEMPORARY DESCRIPTOR W/O FREEING UP THE STRING. Y-REG (MSB) & X-REG(LSB) POINT TO DESCRIPTOR TO BE FREED. ON EXIT Z SET IF ANYTHING FREED
$E6EC (-6420) [~BRANCH~] \SE\ INTEGER BASIC ENTRY POINT TO BRANCH (GET LO/HI THEN JSR)
$E6F5 (-6411) [GTBYTC] \SE\ APPLESOFT - JSR TO CHRGET TO GOBBLE A CHARACTER AND FALL INTO GETBYT
$E6F8 (-6408) [GETBYT] \SE\ GETBYT S/R. EVALS EXPRESSION (FORMULA) POINTED TO BY TXTPTR ($00B8~$00B9) & CONVTS TO 1-BYT VAL IN X-REG & FACLO($00A1). A-REG GETS EXPRESSION TERMINAL SIGN (RESETS Y-REG=0)
$E6F8 (-6408) [GETBYT] \SE\ APPLESOFT - EVAL FORMULA AT TXTPTR. LEAVE RESULT IN FAC AND FALL INTO CONINT. AT ENTRY TXTPTR POINTS TO FIRST CHAR IN FORMULA FOR FIRST NUMBER PLOTFNS PUTS FIRST NUMBER IN FIRST AND SECOND NUMBER IN H2 AND V2
$E6FB (-6405) [CONINT] \SE\ APPLESOFT FP - CONVERT FAC INTO SINGLE BYTE IN X-REG & FACLO.NORMAL EXIT THRU CHRGET. IF FAC<0 OR FAC>255 ILLEGAL QUANT ERROR
$E6FF (-6401) [~GETVERB~] \SE\ INTEGER BASIC ENTRY TO GET NEXT VERB TO USE
$E715 (-6379) [~GET16BIT~] \SE\ INTEGER BASIC ENTRY TO GET A 16-BIT VALUE
$E736 (-6346) [~NOT~] \SE\ INTEGER BASIC ENTRY TO 'NOT' (NOT A VALUE FUNCTION)
$E746 (-6330) [GETNUM] \SE\ APPLESOFT FP - READ 2-BYTE NUM INTO LINNUM FROM TXTPTR. CHECK FOR COMMA. GET SINGLE BYTE NUMB IN X-REG.
$E74A (-6326) [~ABS~] \SE\ INTEGER BASIC ENTRY TO GET ABSOLUTE VALUE OF A NUMBER
$E74C (-6324) [COMBYTE] \SE\ APPLESOFT - CHECK FOR COMMA & GET A BYTE IN X-REG. USES CHKCOM& BETBYT. ON ENTRY TXTPTR POINTS TO COMMA
$E752 (-6318) [GETADR] \SE\ APPLESOFT FP - CONVERT FAC (-65535 TO 65535) INTO 2-BYTE INTEGER (0-65535) IN LINNUM. 'WRAPAROUND' OCCURS IF VALUE IN FAC TOO BIG (A- Y-REGS ALTERED)
$E75C (-6308) [~SGN~] \SE\ INTEGER BASIC ENTRY POINT TO GET SIGN OF A NUMBER
$E782 (-6270) [~SUBTRACTION~] \SE\ INTEGER BASIC ENTRY POINT TO SUBTRACTION FUNCTION
$E785 (-6267) [~ADDITION~] \SE\ INTEGER BASIC ENTRY POINT TO ADDITION FUNCTION
$E7A0 (-6240) [FADDH] \SE\ APPLESOFT FP - ADD 1/2 TO FAC (1/2 IN $EE64)

```
$E7A4 (-6236) [~TAB~] \SE\INTEGER BASIC ENTRY POINT TO HORIZONTAL TAB FUNCTION
$E7A7 (-6233) [FSUB (FPSUB)] \SE\APPLESOFT - MOVE FP NUMBER IN MEMORY POINTED TO BY Y-REG &A-REG INTO ARG AND FALL INTO
                    FSUB (FPSUB)T
$E7AA (-6230) [FSUBT] \SE\APPLESOFT - FP SUBTRACT FAC FROM ARG. ON ENTRY A-REG & 6502 ZERO FLAG REFLECT FACEXP. RESULT
                    TO FAC
$E7BE (-6210) [FADD (FPADD)] \SE\APPLESOFT FP - MOVE THE FP NUMBER IN MEMORY POINTED TO BY Y-REG & A-REG INTO ARG AND
                    FALL INTO FADDT (FPADD). MODIFIES INDEX & XORFPSGN
$E7C1 (-6207) [~COMMA~] \SE\INTEGER BASIC ENTRY POINT TO COMMA FUNCTION
$E7C1 (-6207) [FADDT] \SE\APPLESOFT FP - ADD FAC AND ARG. ON ENTRY A-REG AND ZERO FLAG REFLECT FACEXP. RESULT TO FAC
$E800~$EFFF (-6144~-4097) \HB\      ROM SOCKET E8 (INTEGER BASIC)
$E7E2 (-6174) [~AUTO~] \SE\         INTEGER BASIC ENTRY TO AUTO LINE NUMBERING FUNCTION
$E828 (-6104) [~IF/THEN~] \SE\      INTEGER BASIC ENTRY TO IF/THEN ROUTINE
$E836 (-6090) \SE\                  INTEGER BASIC 'RUN' - LOCATION INTO WHICH DOS CHAINS TO RUN AN INTEGER BASIC PROGRAM
$E83C (-6084) [~GOSUB~] \SE\        INTEGER BASIC ENTRY TO GOSUB HANDLER
$E84E (-6066) [(RESET)] \SE\        RESET FACEXP($009D) AND $00A2 (FACSIGN) & A-REG TO ZERO (A-REG=>0;X- Y-REG NOT
                                    ALTERED)
$E85B (-6053) [~GOTO~] \SE\         INTEGER BASIC ENTRY TO 'GOTO' HANDLER
$E875 (-6027) [~GETNEXT~] \SE\      INTEGER BASIC ENTRY TO 'GETNEXT' (FETCH NEXT STATEMENT FROM TEXT SOURCE)
$E8A5 (-5979) [~RETURN~] \SE\       INTEGER BASIC ENTRY TO ROUTINE FOR RETURN FROM GOSUB
$E8C3 (-5949) [~STOPPED AT~] \SE\   INTEGER BASIC ENTRY TO ROUTINE TO PRINT 'STOPED AT LINE #'
$E8D5 (-5931) [(OVERFLOWPRT)] \SE\  PRINT "OVERFLOW" THEN HALT AT THE APPLESOFT (]) LEVEL
$E8D6 (-5930) [~NEXT~] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO HANDLE 'NEXT' LOOP END
$E913~$E917 (-5869~-5865) [(ONE)] \P5\APPLESOFT FP CONSTANT ONE =1.
$E92D~$E931 (-5843~-5839) [(SQR(.5))] \P5\APPLESOFT FP CONSTANT SQR(.5) = .707..
$E932~$E936 (-5838~-5834) [(SQR(2))] \P5\APPLESOFT FP CONSTANT SQR(2) = 1.414...
$E937~$E93B (-5833~-5813) [(MINUS.ONE.HALF)] \P5\APPLESOFT FP CONSTANT MINUS ONE HALF (-1/2)
$E93A (-5830) [~FOR~] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO HANDLE 'FOR' LOOP INITIALIZATION
$E93C~$E940 (-5828~-5824) [(LN(2))] \P5\APPLESOFT FP CONSTANT (LN(2) = .30103...
$E950 (-5808) [~TO/FOR~] \SE\       INTEGER BASIC ENTRY POINT TO ROUTINE TO HANDLE LOOP COUNTER # TO # STEP #
$E97F (-5761) [FMULT (FPMULT)] \SE\ APPLESOFT FP - MOVE THE FP NUMBER IN MEMORY POINTED TO BY Y-REG & A -REG INTO ARG
                                    AND FALL INTO FMULTT (FPMULT). ALTERS INDEX XORFPSGN
$E982 (-5758) [FMULTT] \SE\         APPLESOFT FP - MULTIPLY FAC AND ARG. ON ENTRY A-REG & ZERO FLAG REFLECT FACEXP.
                                    RESULT TO FAC. XORFPSGN MUST BE COMPUTED BEFORE CALL
$E9E3 (-5661) [CONUPK] \SE\         APPLESOFT FP - LOAD ARG FROM MEMORY POINTED TO BY Y-REG & A-REG. ON EXIT A & Z
                                    REFLECT FACEXP. MODIFIES INDEX & XORFPSGN. (RESET Y-REG=0)
$E9E7 (-5657) \SE\                  APPLESOFT FP - SAME AS $E9E3 EXCEPT USE MEMORY LOCATION POINTED TO BY INDEX
                                    ($005E~$005F)
$EA10~$EA87 (-5616~-5497) [~VERBADL~] \PB\INTEGER BASIC VERB DISPATCH TABLE LOW BYTE
$EA39 (-5575) [MUL10] \SE\          APPLESOFT FP - MULTIPLY FAC BY 10. WORKS FOR BOTH POSITIVE & NEGATIVE NUMBERS
$EA55 (-5547) [DIV10] \SE\          APPLESOFT FP - DIVIDE FAC BY 10. RETURNS POSITIVE NUMBERS ONLY
$EA66 (-5530) [FDIV (FPDIV)] \SE\   APPLESOFT FP - MOVE THE FP NUMBER IN MEMORY POINTED TO BY R-REG & A-REG INTO ARG
                                    AND FALL INTO FDIVT. ALTERS INDEX & XORFPSGN
$EA69 (-5527) [FDIVT (FPDIV2)] \SE\ APPLESOFT FP - DIVIDE ARG BY FAC. ON ENTRY A-REG AND Z REFLECT FACEXP. RESULT IN
                                    FAC. XORFPSGN SHOULD BE COMPUTED BEFORE CALL
$EA88 (-5496) [~VERBADRH~] \PB\     INTEGER BASIC VERB DISPATCH TABLE HI BYTE
$EAE1 (-5407) [(DIVZEROPRT)] \SE\   APPLESOFT - PRINT "DIVISION BY ZERO" THEN HALT AT APPLESOFT (]) LEVEL
$EAF9 (-5383) [MOVFM (FPLOAD)] \SE\ APPLESOFT FP MOVE MEMORY POINTED TO BY Y-REG & A-REG INTO FAC. ON EXIT A-REG & ZERO
                                    FLAG REFLECT FACEXP. RESET EXTENSION BYTE=0 (RESET Y-REG=0)
$EAFD (-5379) \SE\                  APPLESOFT FP - PULL MEMORY POINTED TO BY INDEX ($005E~$005F) INTO FAC & RESET
                                    EXTENSION BYTE = 0 (RESET Y-REG=0)
$EB00~$EB99 (-5376~-5223) \PB\      INTEGER BASIC ERROR TABLE OF CANNED ERROR MESSAGES
```

```
$EB1E (-5346) [MOV2F] \SE\          APPLESOFT FP - PACK FAC AND MOVE IT INTO TEMP2 ($0098~$009C). USES MOVMF. ON EXIT
                                    A-REG & Z FLAG REFLECT FACEXP (RESET Y-REG=0)
$EB21 (-5343) [MOV1F] \SE\          APPLESOFT FP - PACK FAC AND MOVE IT INTO TEMP1 ($0093~$0097). USES MOVMF. ON EXIT
                                    A-REG & Z FLAG REFLECT FACEXP. MODIFIES INDEX ($005E~$005F) (RESET Y-REG=0)
$EB23 (-5341) [MOVML] \SE\          APPLESOFT FP - PACK FAC AND MOVE IT INTO ZERO PAGE AREA POINTED TO BY X-REG. USES
                                    MOVMF. ON EXIT A-REG & Z FLAG REFLECT FACEXP
$EB27 (-5337) \SE\                  APPLESOFT FP - PAC FAC AND STORE IT INTO MEMORY POINTED TO BY FORPNT ($0085~$0086).
                                     MODIFIES INDEX ($005E~$005F) (RESET Y-REG=0)
$EB2B (-5333) [MOVMF (FPSTR)] \SE\  APPLESOFT FP - PACK FAC AND MOVE IT INTO MEMORY POINTED TO BY Y-REG (MSB) & X-REG
                                    (LSB). ON EXIT A-REG & ZERO FLAG REFLECT FACEXP. MODIFIES INDEX ($005E~$005F)
$EB36 (-5322) \SE\                  INTEGER BASIC CONTINUE RUN ROUTINE (W/O DELETING VARIABLES)
$EB53 (-5293) [MOVFA (TR2=>1)] \SE\ APPLESOFT FP - MOVE ARG INTO FAC. ON EXIT A-REG = FACEXP AND ZERO FLAG IS SET
$EB63 (-5277) [MOVAF (TR1=>2)] \SE\ APPLESOFT FP - PACK EXTENSION BYTE INTO FAC & MOVE FAC INTO ARG. ON EXIT A-REG =
                                    FACEXP AND ZERO FLAG IS SET. RESET EXTENSION BYTE = 0 (RESET X-REG=0)
$EB66 (-5274) \SE\                  APPLESOFT FP - SAME AS $EB63 BUT EXTENSION BYTE NOT ALTERED
$EB82 (-5246) [SIGN] \SE\           APPLESOFT FP - SETS A-REG ACCORDING TO VALUE OF FAC. ON EXIT A-REG=1 IF FAC
                                    +;A-REG=0 IF FAC=0;A-REG=$FF IF FAC - (X- Y-REGS NOT ALTERED)
$EB90 (-5232) [SGN (FPSGN)] \SE\    APPLESOFT FP - CALLS SIGN AND FLOATS THE RESULT IN THE FAC. FAC=+1 IF FAC WAS +;=0
                                    IF FAC WAS 0;=-1 IF FAC WAS -
$EB93 (-5229) [FLOAT] \SE\          APLESOFT FP - FLOAT THE SIGNED INTEGER IN A-REG INTO FAC
$EBAA (-5206) [~INPUT~] \SE\        INTEGER BASIC ENTRY TO INPUT ROUTINE
$EBAF (-5201) [ABS (FPABS)] \SE\    APPLESOFT FP - TAKES ABS OLUTE VALUE OF NUMBER IN FAC & LEAVES RESULT IN FAC
$EBB2 (-5198) [FCOMP] \SE\          APPLESOFT FP - COMPARE FAC AND PACKED NUMBER IN MEMORY POINTED TO BY Y-REG & A-REG.
                                    ON EXIT A=1 IF MEM<FAC;A=0 IF MEM=FAC;A=$FF IF MEM>FAC
$EBF2 (-5134) [QINT] \SE\           APPLESOFT QUICK GREATEST INTEGER FUNCTION. LEAVE INT(FAC)IN FAC MANTISSA (HO~MO~LO
                                    SIGNED). ASUMES FAC<2~23 (RESET Y-REG=0)
$EC00~$EDFF (-5120~-4609) [~SYNTABL~] \PB\INTEGER BASIC SYNTAX TABLE
$EC23 (-5085) [INT (FPINT)] \SE\    APPLESOFT FP - COMPUTES GREATES INT (FPINT)EGER VALUE OF FAC. MODIFIES CHARAC
                                    ($000D). USES QINT (FPINT). RESULT TO FAC. MODIFIES CHARAC ($000D)
$EC40 (-5056) [(INITFACMANT)] \SE\  APPLESOFT FP - INITIALIZED MANTISSA OF FAC (EXCEPT EXTENSION BYTE) TO VALUE IN
                                    A-REGISTER
$EC4A (-5046) [FIN] \SE\            APPLESOFT - INPUT FP NUMB INTO FAC FROM CHRGET. ASSUMES 6502 REGS HAVE BEEN SET UP
                                    BY CHRGET THAT FETCHED 1ST DIGIT
$ED14~$$ED18[(ONE.BILLION)] \P5\    APPLESOFT 5-BYTE FLOATING POINT CONSTANT 1000000000 (1E9)
$ED19 (-4839) [INPRT] \SE\          APPLESOFT - PRINT 'IN' & CURRENT LINE # FROM CURLIN. USES LPRINT
$ED24 (-4828) [LINPRT] \SE\         APPLESOFT - PRINTS 2-BYTE UNSIGNED NUMBER IN X-REG (MSB) & A-REG (LSB)
$ED2E (-4818) [PRNTFAC] \SE\        APPLESOFT - PRINTS & DESTROYS CURRENT VALUE OF FAC. USES FOUT & STROUT
$ED34 (-4812) [FOUT] \SE\           CREATES A STRING IN FBUFFR EQUIVALENT IN VALUE TO FAC. ON EXITY-REG &A-REG POINT TO
                                    THE STRING. FAC SCRAMBLED
$EE03 (-4605) [~PRNTSTR~] \SE\      INTEGER BASIC ENTRY TO FUNCTION WHICH PRINTS A STRING
$EE22 (-4574) [~LEN~] \SE\          INTEGER BASIC ENTRY TO FUNCTION TO OBTAIN LENGTH OF A STRING
$EE34 (-4556) [~GETVAL~] \SE\       INTEGER BASIC ENTRY TO ROUTINE TO GET A VALUE WHICH WILL FIT INTO A SINGLE BYTE
                                    (VAL<=255)
$EE3F (-4545) [~PLOT~] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO DO A LO~RES PLOT (I.E. PLOT A COLORED SQUARE ON
                                    LO-RES SCREEN)
$EE4E (-4530) [~COLOR~] \SE\        INTEGER BASIC ENTRY TO ROUTINE TO SET COLOR VALUE FOR LO-RES
$EE54 (-4524) [~MAN~] \SE\          INTEGER BASIC ENTRY TO MANUAL LINE NUMBER FUNCTION
$EE57 (-4521) [~VTAB~] \SE\         INTEGER BASIC ENTRY TO VERTICAL TAB FUNCTION
$EE64~$EE68 (-4508~-4504) [(ONE.HALF)] \P5\APPLESOFT 5-BYTE FP CONSTANT ONE HALF (1/2)
$EE68 (-4504) [RNGERR] \P1\         INTEGER BASIC RANGE ERROR
$EE8D (-4457) [SQR (FPSQR)] \SE\    APPLESOFT FP - TAKE SQUARE ROOT OF FAC. RESULT TO FAC. MODIFIES CHARAC INDEX AND
                                    MANY OTHER FP LOCNS
```

```
$EE97 (-4457) [FPWRT (FPEXP)] \SE\  APPLESOFT FP EXPONENTATION (ARG TO FAC POWER) ON ENTRY A-REG & ZERO FLAG SHOULD
                                    REFLECT VALUE OF FACEXP. RESULT TO FAC. MODIFIES MANY FP LOCNS
$EEA0 (-4448) ["CALL"] \SE\         INTEGER BASIC ENTRY POINT TO CALL A SUB/ROT FUNCTION
$EEB0 (-4432) ["HLIN"] \SE\         INTEGER BASIC ENTRY POINT TO DRAW A LO-RES HORIZONTAL LINE
$EEC6 (-4410) ["VLIN"] \SE\         INTEGER BASIC ENTRY POINT TO DRAW A LO-RES VERTICAL LINE
$EED0 (-4400) [NEGOP] \SE\          APPLESOFT FP - LET FAC = -FAC (X- Y-REGS NOT ALTERED)
$EED3 (-4397) ["PRINT"] \SE\        INTEGER BASIC ENTRY POINT TO PRINT ERROR MESSAGE/BELL
$EED8"$EEDF (-4389"-4385) [(LOG(E)2)] \P5\APPLESOFT FP CONSTANT LOG(E)2
$EEF6 (-4362) ["PEEK"] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO 'PEEK' AT THE CONTENTS OF A MEMORY LOCATION
$EF00 (-4352) ["GETVAL255"] \SE\    INTEGER BASIC ENTRY TO ROUTINE TO GET A ONE-BYTE VALUE
$EF09 (-4343) [EXP] \SE\            APPLESOFT FP - RAISE E TO THE FAC POWER. RESULT TO FAC. MODIFIES INDEX CHARAC
                                    COMPRTYP XORFPSGN AND MANY OTHER FP LOCNS
$EF10 (-4336) ["DIVIDE"] \SE\       INTEGER BASIC ENTRY TO DIVIDE FUNCTION
$EF1E (-4322) ["DIMVARB"] \SE\      INTEGER BASIC ENTRY TO ROUTINE TO DIMENSION A VARIABLE
$EF4E (-4274) ["RND"] \SE\          INTEGER BASIC ENTRY TO RANDOM NUMBER GENERATOR
$EFAE (-4178) [RND] \SE\            APPLESOFT FP - FORM A 'RANDOM' NUMBER IN FAC USING ORIGINAL VALUE IN FAC AS
                                    PARAMETER 'KEY' OR 'SEED'. MODIFIES MANY FP LOCNS
$EFEA (-4118) [COS] \SE\            APPLESOFT FP - COMPUTE THE COSINE OF THE NUMBER IN FAC. RESULT TO FAC. MODIFIES
                                    INDEX CHARAC COMPRTYP XORFPSGN AND MANY OTHER FP LOCNS
$EFEC (-4116) ["RUN"] \SE\          APPLE INTEGER BASIC RUN ROUTINE (RUN FROM BEGINNING)
$EFF2 (-4110) ["RUN #N"] \SE\       INTEGER BASIC ENTRY TO ROUTINE TO RUN FROM LINE #N
$EFF1 (-4111) [SIN] \SE\            APPLESOFT FP - COMPUTE THE SINE OF THE NUMBER IN FAC. RESULT TO FAC. MODIFIES INDEX
                                    CHARAC COMPTRTYP XORFPSGN & MANY OTHER FP LOCNS
$F000"$F7FF (-4096"-2049) \HB\      ROM SOCKET F0 (1K INTEGER BASIC" 1 K MONITOR IN APPLE II (NOT II+))
$F000 (-4096) ["SCRATCH"] \SE\      INTEGER BASIC ENTRY TO SCRATCH EVERYTHING ROUTINE
$F03A (-4038) [TAN] \SE\            APPLESOFT FP - COMPUTE THE TANGENT OF THE NUMBER IN FAC. RESULT TO FAC. MODIFIES
                                    CHARAC INDEX XORFPSGN AND MANY OTHER FP LOCNS
$F04D (-4019) ["HIMEM"] \SE\        INTEGER BASIC ENTRY TO THE HIMEM FUNCTION
$F063"$F067 (-3997"-3993) [(PI/2)] \P5\APPLESOFT 5-BYTE FLOATING POINT CONSTANT PI/2 = 1.508..
$F06B"$F06F (-3989"-3985) [(TWO PI)] \P5\APPLESOFT 5-BYTE FLOATING POINT CONSTANT 2*PI = 6.2832...
$F070"$F075 (-3984"-3979) [(ONE-QUARTER)] \P5\APPLESOFT5-BYTE FLOATING POINT CONSTANT 1/4 (0.25)
$F078 (-3976)                       INTEGER BASIC 'LOAD' (CASSETTE TAPE)
$F09E (-3938) [ATN] \SE\            APPLESOFT FP COMPUTE THE ARCTANGENT OF NUMBER IN FAC. RESULT TO FAC. MODIFIES INDEX
                                    XORFPSGN AND MANY OTHER FP LOCNS
$F0C9 (-3895) ["LOMEM"] \SE\        INTEGER BASIC ENTRY TO LOMEM ROUTINE
$F0DF (-3873) ["LOAD"]              INTEGER BASIC ENTRY TO LOAD SUBROUTINE (LOAD A PROGRAM FROM CASSETTE TAPE)
$F11E (-3810) ["SETHDR"] \SE\       INTEGER BASIC ENTRY TO SET UP HEADER FOR SAVE/LOAD PARAMETERS
$F11E (-3810) [ACADR]               HI-RES GRAPHICS 2-BYTE TAPE READ SETUP
$F12C (-3796) ["SETBUF"] \SE\       INTEGER BASIC ENTRY TO ROUTINE TO SET UP PROGRAM SAVE/LOAD PARAMETERS
$F140 (-3776) ["SAVE"] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO SAVE A PROGRAM TO CASSETTE TAPE
$F161 (-3743) ["PRTERR"] \SE\       INTEGER BASIC ENTRY TO ROUTINE TO PRINT AN ERROR MESSAGE
$F167 (-3737) ["POP"] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO POP THE RETURN STACK FOR GOSUB
$F171 (-3727) ["TRACE"] \SE\        INTEGER BASIC ENTRY TO ROUTINE TO SET TRACE MODE FOR EXECUTION
$F176 (-3722) ["NOTRACE"] \SE\      INTEGER BASIC ENTRY TO ROUTINE TO TURN OFF TRACE MODE
$F17D (-3715) ["TRACEIT"] \SE\      INTEGER BASIC ENTRY TO ROUTINE TO EXECUTE THE TRACE FUNCTION
$F1EC (-3604) [PLOTFNS] \SE\        APPLESOFT - GET 2 LO-RES PLOTTING COORDS SEPARATED BY COMMA FM TXTPTR. PUT FIRST #
                                    IN FIRST AND SECOND # IN H2 & V2
$F279 (-3463) ["STEP"] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO HANDLE STEP FUNCTION FOR FOR/NEXT LOOP
$F2E0 (-3360) ["NODSP"] \SE\        INTEGER BASIC ENTRY TO ROUTINE TO TURN OFF DISPLAY FUNCTION
$F2E9 (-3351) [HANDLERR] \SE\       APPLESOFT ERROR PROC - SAVE CURLIN IN ERRLIN;TXTPTR IN ERRPOS;X-REG IN ERRNUM;
                                    REMSTK IN ERRSTK
```

```
$F304 (-3324) [~DSP~] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO DISPLAY A VARIABLE SET
$F30A (-3318) [~CON~] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO CONTINUE EXECUTION
$F317 (-3305) [RESUME] \SE\         APPLESOFT ERROR PROC - RESTORE CURLIN FROM ERRLIN & TXTPTR FROM ERRPOS. TRANSFER
                                    ERRSTK INTO 6502 STACK POINTER
$F31D (-3299) [~ASC~] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO PERFORM THE ASC (ASCII) FUNCTION
$F33B (-3269) [~PDL~] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO READ A PADDLE
$F351 (-3247) [~RDKEY~] \SE\        INTEGER BASIC ENTRY TO ROUTINE TO READ AN INPUT FOR BASIC FROM KEYBOARD
$F371 (-3215) [~EXP~] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO EXPONENTIATE (RAISE TO A POWER)
$F3C9 (-3127) [~PR#S~] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO SET OUTPUT PORT
$F3D4 (-3116) [HGR2] \SE\           APPLESOFT HI-RES - INITIALIZE & CLEAR PAGE 2 HI-RES REGARDLESS  OF SCREEN BEING
                                    DISPLAYED
$F3DE (-3106) [HGR] \SE\            APPLESOFT HI-RES - INITIALIZE & CLEAR PAGE 1 HI-RES REGARDLESS OF SCREEN BEING
                                    DISPLAYED
$F3EE (-3090) [HCLR] \SE\           APPLESOFT HI-RES - CLEAR HI-RES SCREEN TO BLACK
$F3F2 (-3086) [BKGND] \SE\          APPLESOFT HI-RES - CLEAR HI-RES SCREEN TO LAST PLOTTED COLOR
$F40D (-3059) [HPOSN] \SE\          APPLESOFT HI-RES - POSN HI-RES CURSOR W/O PLOTTING. HPAG DETERMINES WHICH PAGE;
                                    HORIZ = Y-REG(MSB)&X-REG(LSB);VERT= A-REG
$F41A (-3046) [~IN#S~] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO SET INPUT PORT
$F425~$F65D (-3035~-2467)           APPLE II FLOATING POINT PACKAGE (NOT USED IN APPLESOFT)
$F453 (-2989) [HPLOT] \SE\          APPLESOFT HI-RES - CALL HPOSN THEN PLOT DOT THERE. NO DOT MAY BE PLOTTED IS
                                    PLOTTING NON-WHITE AT COMPLEMENTARY COLOR X COORD
$F425 (-3035) [ADD] \SE\            ADD 3-BYTE M1 TO 3-BYTE M2 AND LEAVE RESULT IN M1 (NOT FP ADD BUT USED IN FP PKG)
                                    (A- X-REGS ALTERED)
$F437 (-3017) [ABSWAP] \SE\         TAKE ABSOLUTE VALUE OF FP1; THEN SWAP FP1 WITH FP2 (FP1=$00F8;$FP2=$00F4) (A-
                                    X-REGS ALTERED)
$F451 (-2991) [FLOAT] \SE\          CONVERT INTEGER (HIGH BYTE IN M1;LOW BYTE IN M1+1;M1+2 CLEARED) TO NORMALIZED FL
                                    POINT EQUIV IN FP1 (A-REG ALTERED)
$F463 (-2973) [NORM] \SE\           NORMALIZE FLOATING POINT NUMBER IN FP1 (A-REG ALTERED)
$F4A4 (-2908) [FCOMPL] \SE\         VALUE OF FLOATING POINT NUMBER IN FP1 IS NEGATED THEN NORMALIZED (A- X-REGS ALTERED)
$F468 (-2968) [FSUB] \SE\           FLOATING POINT SUBTRACTION MINUEND IN FP1;SUBTRAHEND IN FP2;NORMALIZED DIFFERENCE
                                    TO FP1 (A- X-REGS ALTERED)
$F46E (-2962) [FADD] \SE\           FLOATING POINT NUMBER IN FP1 ADDED TO THAT IN FP2. NORMALIZED RESULT LEFT IN FP1
                                    (A- X-REGS ALTERED)
$F47D (-2947) [RTAR] \SE\           DENORMALIZE FP1 BY SHIFTING M1(&E) RIGHT 1 BIT POSN & INCREMENTING X1 (A- X-REGS
                                    ALTERED)
$F48C (-2932) [FMUL] \SE\           FLOATING POINT MULTIPLY S/R: MUTIPLICAND IN FP1; MULTIPLIER IN FP2; SIGNED
                                    NORMALIZED PRODUCT IN FP1 (A- X- Y-REGS ALTERED)
$F4B2 (-2894) [FMUL] \SE\           FL PT DIVIDE S/R: NORM DIVIDEND IN FP2;NORM DIVIDER IN FP1;SIGNED NORM FP QUOTIENT
                                    TO FP1 (A- X- Y-REGS ALTERED)
$F500~$F666 (-2816~-2458)           APPLE II MINIASSEMBLER SOFTWARE PACKAGE
$F500 (-2816) [REL]                 MINIASSEMBLER MEMORY LOCATION 'REL'
$F50C (-2804) [REL2]                MINIASSEMBLER MEMORY LOCATION 'REL2'
$F516 (-2794) [REL3]                MINIASSEMBLER MEMORY LOCATION 'REL3'
$F519 (-2791) [ERR3]                MINIASSEMBLER MEMORY LOCATION 'ERR3'
$F51B (-2789) [FINDOP]              MINIASSEMBLER MEMORY LOCATION 'FINDOP'
$F51D (-2787) [FNDOP2]              MINIASSEMBLER MEMORY LOCATION 'FNDOP2'
$F530 (-2768) [HLIN] \SE\           APPLESOFT HI-RES HORIZ LINE DRAWING FROM LAST POINT PLOTTED TOX-COORD =
                                    X-REG(MSB)&A-REG(LSB);Y-COORD=Y-REG
$F538 (-2760) [FAKEMON3]            MINIASSEMBER MEMORY LOCATION 'FAKEMON3'
$F53D (-2755) [FAKEMON]             MINIASSEMBLER MEMORY LOCATION 'FAKEMON'
$F544 (-2748) [FAKEMON2]            MINIASSEMBLER MEMORY LOCATION 'FAKEMON2'
```

```
$F55C (-2724) [TRYNEXT]              MINIASSEMBLER MEMORY LOCATION 'TRYNEXT'
$F578 (-2696) [NREL]                 MINIASSEMBLER MEMORY LOCATION 'NREL'
$F57C (-2692) [NEXTOP]               MINIASSEMBLER MEMORY LOCATION 'NEXTOP'
$F586 (-2682) [ERR]                  MINIASSEMBLER MEMORY LOCATION 'ERR'
$F588 (-2680) [ERR2]                 MINIASSEMBLER MEMORY LOCATION 'ERR2'
$F592 (-2670) [RESETZ]               MINIASSEMBLER MEMORY LOCATION 'RESETZ'
$F595 (-2667) [NXTLINE]              MINIASSEMBER MEMORY LOCATION 'NXTLINE'
$F5B1 (-2639) [ERR4]                 MINIASSEMBLER MEMORY LOCATION 'ERR4'
$F5B9 (-2631) [SPACE]                MINIASSEMBLER MEMORY LOCATION 'SPACE'
$F5BD (-2627) [NXTMN]                MINIASSEMBLER MEMORY LOCATION 'NXTMN'
$F5C0 (-2624) [NXTM]                 MINIASSEMBLER MEMORY LOCATION 'NXTM'
$F5CB (-2613) [HFIND] \SE\           APPLESOFT HI-RES HFIND. CONVERT HI-RES CURSOR POSN TO X-Y COORDS. ON EXIT
                                     $00E0=HORIZ LSB;$00E1=HORIZ MSB;$00E2=VERT
$F5CB (-2613) [NXTM2]                MINIASSEMBLER MEMORY LOCATION 'NXTM2'
$F5D9 (-2599) [FORM1]                MINIASSEMBLER MEMORY LOCATION 'FORM1'
$F5DB (-2597) [FORM2]                MINIASSEMBLER MEMORY LOCATION 'FORM2'
$F5F8 (-2568) [FORM3]                MINIASSEMBLER MEMORY LOCATION 'FORM3'
$F5F9 (-2567) [FORM4]                MINIASSEMBLER MEMORY LOCATION 'FORM4'
$F5FA (-2566) [FORM5]                MINIASSEMBLER MEMORY LOCATION 'FORM5'
$F601 (-2559) [DRAW] \SE\            APPLESOFT HI-RES - DRAW SHAPE POINTED TO BY Y-REG(MSB)&X-REG(LSB) BY INVERTING
                                     EXISTING COLOR OF DOTS THE SHAPE DRAWS OVER. A-REG=ROTATION FACTOR
$F608 (-2552) [FORM6]                MINIASSEMBLER MEMORY LOCATION 'FORM6'
$F60D (-2547) [FORM7]                MINIASSEMBLER MEMORY LOCATION 'FORM7'
$F622 (-2526) [FORM8]                MINIASSEMBLER MEMORY LOCATION 'FORM8'
$F631 (-2511) [FORM9]                MINIASSEMBLER MEMORY LOCATION 'FORM9'
$F634 (-2508) [GETNSP]               MINIASSEMBLER MEMORY LOCATION 'GETNSP'
$F640 (-2496) [FIX] \SE\             FROMFLOATING POINT NUMBER IN FP1 EXTRACT INTEGER. PUT HIGH-ORDER BYTE IN
                                     M1;LOW-ORDER IN M1+1 (A- X-REGS ALTERED)
$F65D (-2467) [XDRAW] \SE\           APPLESOFT HI-RES - DRAW SHAPE POINTED TO BY Y-REG(MSB)&X-REG(LSB) BY INVERTING
                                     EXISTING COLOR OF DOTS SHAPE DRAWS OVER. A-REG = ROT FACTOR
$F666 (-2458) [MINASM]               TURN ON MINIASSEMBLER (KEYBOARD INPUT WILL BE INTERPRETED AS A SEMBLY-LANGUAGE
                                     INSTRUCTION)
$F689~$F7FA (-2423~-2054) \SB\       'SWEET-16' 16-BIT PSEUDO-MACHINE INTERPRETER
$F689 (-2423) \SE\                   SWEET-16 INTERPRETER ENTRY
$F6B9 (-2375) [HFNS] \SE\            APPLESOFT - GET HI-RES PLOTTING COORDINATE FROM TXTPTR SETS UP 6502 REGISTERS FOR
                                     HPOSN: A-REG=VERT COORD;X-REG LSB OF HORIZ;Y-REG MSB OF HORIZ (A- X- Y-REGS ALTERED)
$F6EC (-2324) [SETHCOL] \SE\         APPLESOFT HI-RES - SET COLOR TO CONTENTS OF X-REG (MUST BE LESS THAN 8)
$F775 (-2187) [SHLOAD] \SE\          APPLESOFT HI-RES. LOADS SHAPE TABLE INTO MEMORY FROM TAPE ABOVE MEMSIZ (HIMEM) AND
                                     SETS POINTER AT $00E8
$F7D9 (-2087) [GETARYPT] \SE\        APPLESOFT - READ VAR NAME FROM CHRGET & FIND IT IN MEMORY.ON EXIT VAL OF VAR IN
                                     VARPNT AND Y-REG(MSB)&A-REG(LSB)
$F800~$FFFF (-2048~-1) \HB\          ROM SOCKET F8 (MONITOR) NOTE: WHEN LANGUAGE CARD RAM DESELECTED MONITOR ON CARD
                                     ACTIVE
$F800~$FFFF (-2048~-1) \SB\          APPLE II SYSTEM MONITOR (MAIN BODY)
$F800~$FFFF (-2048~-1) \HB\          APPLE LANGUAGE CARD ADDITIONAL ROM/RAM
$F800 (-2048) [PLOT] \SE\            LO-RES PLOT POINT AT X-COORD=(Y-REG) Y-COORD=(A-REG) LEAVING GBASL~H AND MASK SET
                                     (SEE CALL-APPLE DEC 78) (A-REG ALTERED)
$F80C (-2036) [RTMASK]               MONITOR MEMORY LOCATION 'RTMASK'
$F80E (-2034) [PLOT1] \SE\           LO-RES PLOT A POINT X-COORD=(Y-REG) Y-COORD PER GBASL~H & MASK (A-REG ALTERED)
$F819 (-2023) \SE\                   HLINE S/R (SEE CALL-APPLE NOV/DEC 78 PG4)
$F819 (-2023) [HLINE] \SE\           LO-RES S/R TO DRAW HORIZONTAL LINE AT Y-COORD = (A-REG) WITH X-COORDS FROM (A-REG)
                                     THRU (H2)($002C) (A- Y-REGS ALTERED)
```

```
$F81C (-2020) [HLINE1] \SE\          LO-RES S/R. DRAW HORZ LINE AT Y-COORD ESTAB BY GBASL~H & MASK. X-CORDS FROM (Y-REG)
                                     THRU ($002C) {A- Y-REGS ALTERED}
$F826 (-2010) [VLINE2] \SE\          LO-RES PLOT VERTICAL LINE AT X-COORD = (Y-REG) AND Y-COORD FROM (A-REG)+1+CARRY
                                     THRU ($002D) {A-REG ALTERED}
$F828 (-2008) [VLINE] \SE\           LO-RES PLOT VERT LINE AT X-COORD = (Y-REG) AND Y-COORD FROM (A-REG) THRU ($002D)
                                     {A-REG ALTERED}
$F831 (-1999) [RTS1]                 MONITOR MEMORY LOCATION 'RTS1'
$F832 (-1998) [CLRSCR] \SE\          MONITOR S/R TO CLEAR SCREEN - GRAPHICS MODE FULL SCREEN) {A- Y-REGS ALTERED}
$F832 (-1998) [CLRSCR] \SE\          CLEAR LO-RES GRAPHICS SCREEN1 TO BLACK (INVERSE @ IN TEXT MODE) MIXED GRAPHICS AREA
                                     ONLY {A- Y-REGS ALTERED}
$F836 (-1994) [CLRTOP] \SE\          CLEAR TOP 20 LINES PAGE1 TO INVERSE @ IN TEXT; BLACK IN LO-RES GRAPHICS (40 LO-RES
                                     GRAPHIC 'LINES') {A- Y-REGS ALTERED}
$F838 (-1992) [CLRSC2] \SE\          CLEAR LINES 0 THRU (Y-REG) 40 COLUMNS WIDE TO BLACK IN LO-RES GRAPHICS OR INVERSE @
                                     IN TEXT PAGE 1 {A- Y-REGS ALTERED}
$F83C (-1988) [CLRSC3] \SE\          CLEAR LO-RES GRAPHICS PARTIAL TOP LEFT: X-COORD 0 THRU (Y-REG); Y-COORD 0 THRU
                                     ($002D) {A- Y-REGS ALTERED}
$F847 (-1977) [GBASCALC] \SE\        COMPUTE GRAPHICS BASE MEMORY ADDRESS FOR LINE IN A-REG (NOTE: 2 LO-RES GRAPHICS
                                     LINES PER TEXT LINE SO (A)= LINE/2); SET GBASL~H {A-REG ALTERED}
$F856 (-1962) [GBCALC]               MONITOR MEMORY LOCATION 'GBCALC'
$F85F (-1953) [NXTCOL] \SE\          MONITOR LO-RES S/R. CHANGE COLOR TO (COLOR)+3 {A-REG ALTERED}
$F864 (-1948) [SETCOL] \SE\          SET LO-RES COLOR TO COLOR CODE SPECIFIED BY A-REG FOR FUTURE PLOTTING {A-REG
                                     ALTERED}
$F871 (-1935) [SCRN] \SE\            GET (LOAD TO A-REG) LO-RES GRAPHICS COLOR OF POINT Y-COORD = (A-REG); X-COORD =
                                     (X-REG)  {A-REG ALTERED}
$F879 (-1927) [SCRN2]                MONITOR MEMORY LOCATION 'SCRN2'
$F87F (-1921) [RTMSKZ]               MONITOR MEMORY LOCATION 'RTMSKZ'
$F882 (-1918) [INSDS1]               MONITOR MEMORY LOCATION 'INSDS1'
$F88E (-1906) [INSDS2]               MONITOR S/R - DISASSEMBLER ENTRY
$F89B (-1893) [IEVEN]                MONITOR MEMORY LOCATION 'IEVEN'
$F8A5 (-1883) [ERR]                  MONITOR MEMORY LOCATION 'ERR'
$F8A9 (-1879) [GETFMT]               MONITOR MEMORY LOCATION GETFMT
$F8BE (-1858) [MNNDX1]               MONITOR MEMORY LOCATION 'MNNDX1'
$F8C2 (-1854) [MNNDX2]               MONITOR MEMORY LOCATION 'MNNDX2'
$F8C9 (-1847) [MNNDX3]               MONITOR MEMORY LOCATION 'MNNDX3'
$F8D0 (-1840) [INSTDSP]              MONITOR & MINIASSEMBLER MEMORY LOCATION 'INSTDSP' (INSTRUCTION DISPLAY)
$F8D4 (-1836) [PRNTOP]               MONITOR MEMORY LOCATION 'PRNTOP' (PRINT OPERATION CODE)
$F8DB (-1829) [PRNTBL]               MONITOR MEMORY LOCATION 'PRNTBL'
$F8F5 (-1803) [PRMN1]                MONITOR MEMORY LOCATION 'PRMN1' (PRINT MNEMONIC)
$F8F5 (-1803) [NXTCOL]               AUTOSTART MONITOR MEMORY LOCATION 'NXTCOL'
$F8F9 (-1799) [PRMN2]                MONITOR MEMORY LOCATION 'PRMN2'
$F910 (-1776) [PRADR1]               MONITOR MEMORY LOCATION 'PRADR1' (PRINT ADDRESS)
$F914 (-1772) [PRADR2]               MONITOR MEMORY LOCATION 'PRADR2'
$F926 (-1754) [PRADR3]               MONITOR MEMORY LOCATION 'PRADR3'
$F92A (-1750) [PRADR4]               MONITOR MEMORY LOCATION 'PRADR4'
$F930 (-1744) [PRADR5]               MONITOR MEMORY LOCATION 'PRADR5'
$F938 (-1736) [RELADR]               MONITOR MEMORY LOCATION 'RELADR' (RELATIVE ADDRESS)
$F940 (-1728) [PRNTYX] \SE\          MONITOR S/R- PRINT CONTENTS OF Y AND X AS 4 HEX DIGITS {A- X-REGS ALTERED}
$F941 (-1727) [PRNTAX] \SE\          MONITOR S/R-PRINT CONTENTS OF A-REG & X-REG AS HEX DIGITS {A- X-REGS ALTERED}
$F944 (-1724) [PRNTX] \SE\           PRINT CONTENTS OF X-REG AS HEX DIGITS {A- X-REGS ALTERED}
$F948 (-1720) [PRBLNK] \SE\          PRINT THREE BLANKS THROUGH COUT {A- X-REGS ALTERED}
$F94C (-1716) [PRBL2] \SE\           MONITOR S/R- PRINT BLANKS: X REG CONTAINS NUMBER TO PRINT. CLOBBERS AC~X {A- X-REGS
                                     ALTERED}
```

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

$F94C (-1716) [PRBL3] \SE\PRINT A-REG FOLLOWED BY (X-REG)-1 BLANKS (A- X-REGS ALTERED)
$F953 (-1709) [PCADJ]   MINIASSEMBLER MEMORY LOCATION 'PCADJ' (PROGRAM COUNTER ADJUST: 0=1 BYTE; 1=2 BYTES; 2=3 BYTES)
$F954 (-1708) [PCADJ2]  MONITOR & MINIASSEMBLER MEMORY LOCATION 'PCADJ2'
$F956 (-1706) [PCADJ3]  MONITOR MEMORY LOCATION 'PCADJ3'
$F95C (-1700) [PCADJ4]  MONITOR MEMORY LOCATION 'PCADJ4'
$F961 (-1695) [RTS2]    MONITOR MEMORY LOCATION 'RTS2'
$F962 (-1694) [FMT1]    MONITOR MEMORY LOCATION 'FMT1'
$F9A6 (-1626) [FMT2]    MONITOR MEMORY LOCATION 'FMT2'
$F9B4 (-1612) [CHAR1]   MONITOR & MINIASSEMBER MEMORY LOCATION 'CHAR1'
$F9BA (-1606) [CHAR2]   MONITOR & MINIASSEMBLER MEMORY LOCATION 'CHAR2'
$F9C0 (-1600) [MNEML]   MONITOR & MINIASSEMBLER MEMORY LOCATION 'MNEML'
$FA00 (-1536) [MNEMR]   MONITOR & MINIASSEMBER MEMORY LOCATION 'MNEMR'
$FA40~$FA85 (-1472~-1403)          SINGLE-STEP SIMULATOR SUBROUTINE (NOT IN AUTOSTART ROM)
$FA40 (-1472) [IRQ] \SE\           AUTOSTART ROM MONITOR S/R - IRQ HANDLER
$FA43 (-1469) [STEP]               MONITOR S/R- PERFORM A SINGLE STEP (NOT AVAILABLE WITH AUTOSTART ROM). EXECUTES ONE
                                   INSTRUCTION AT (PCL~H) WITH REGISTER RESTORE BEFORE; REGISTER SAVE AFTER; UPDATE OF
                                   PCL~H; DISPLAY OF INSTRUCTION & DISPLAY OF RESULT REGISTERS
$FA4E (-1458) [XQINIT]             MONITOR MEMORY LOCATION 'XQINIT'
$FA59 (-1447) [OLDBRK]             AUTOSTART MONITOR MEMORY LOCATION 'OLDBRK'
$FA62 (-1438) [RESET]              AUTOSTART MONITOR MEMORY LOCATION 'RESET'
$FA6F (-1425) [INITAN]             AUTOSTART MONITOR MEMORY LOCATION 'INITAN'
$FA78 (-1416) [XQ1]                MONITOR MEMORY LOCATION 'XQ1'
$FA7A (-1414) [XQ2]                MONITOR MEMORY LOCATION 'XQ2'
$FA81 (-1407) [NEWMON]             AUTOSTART MONITOR MEMORY LOCATION 'NEWMON'
$FA86 (-1402) [IRQ] \SE\           MONITOR S/R- IRQ HANDLER. NOTE: MOVED TO $FA40 IN AUTOSTART ROM
$FA92 (-1390) [BREAK] \SE\         MONITOR S/R - BREAK HANDLER
$FA9B (-1381)                      AUTOSTART MONITOR MEMORY LOCATION 'FIXSEV'
$FA9C (-1380) [XBRK]               MONITOR MEMORY LOCATION 'XBRK'
$FAA5~$FAD6 (-1371~-1322)          BLOCK OF CODE ASSOCIATED WITH SINGLE-STEP SIMULATOR IN NORMAL MONITOR REMOVED FROM
                                   AUTOSTART ROM
$FAA5 (-1371) [XRTI]               MONITOR MEMORY LOCATION 'XRTI'
$FAA6 (-1370) [PWRUP]              AUTOSTART MONITOR MEMORY LOCATION 'PWRUP'
$FAA9 (-1367) [SETPG3]             AUTOSTART MONITOR MEMORY LOCATION 'SETPG3'
$FAA9 (-1367) [XRTS]               MONITOR MEMORY LOCATION 'XRTS'
$FAAD (-1363) [PCINC2]             MONITOR MEMORY LOCATION 'PCINC2'
$FAAF (-1361) [PCINC3]             MONITOR MEMORY LOCATION 'PCINC3'
$FAB9 (-1351) [XJSR]               MONITOR MEMORY LOCATION 'XJSR'
$FAB9 (-1351) [SLOOP]              AUTOSTART MONITOR MEMORY LOCATION 'SLOOP'
$FAC4 (-1340) [XJMP]               MONITOR MEMORY LOCATION 'XJMP'
$FAC5 (-1339) [XJMPAT]             MONITOR MEMORY LOCATION 'XJMPAT'
$FAC7 (-1337) [NXTBYT]             AUTOSTART MONITOR MEMORY LOCATION 'NXTBYT'
$FACD (-1331) [NEWPCL]             MONITOR MEMORY LOCATION 'NEWPCL'
$FAD1 (-1327) [RTNJMP]             MONITOR MEMORY LOCATION 'RTNJMP'
$FAD7 (-1321) [REGDSP] \SE\        DISPLAY SAVED REGISTER CONTENTS FROM MEMORY LOCNS $0045~$0049 WITH PRECEDING
                                   CARRIAGE RETURN (SEE 'SAVE' ROUTINE AT $FF4A) (A- X-REGS ALTERED)
$FADA (-1318) [RGDSP1] \SE\        DISPLAY SAVED REGISTER CONTENTS FROM MEMORY LOCNS $0045~$0049 WITHOUT PRECEDING
                                   CARRIAGE RETURN (SEE 'SAVE' ROUTINE AT $FF4A) (A- X-REGS ALTERED)
$FAE4 (-1308) [RDSP1]              MONITOR MEMORY LOCATION 'RDSP1'
$FAFD~$FB18 (-1283~-1256)          BLOCK OF CODE ASSOCIATED WITH SINGLE-STEP SIMULATOR IN NORMAL MONITOR REMOVED FROM
                                   AUTOSTART ROM
$FAFD (-1283) [BRANCH]             MONITOR MEMORY LOCATION 'BRANCH'
-------------------------------------------------------------------------------------------------------------------------------
```

$FAFD (-1283) [PWRCON] AUTOSTART MONITOR MEMORY LOCATION 'PWRCON'
$FB05 (-1275) [DISKID] AUTOSTART MONITOR MEMORY LOCATION 'DISKID'
$FB09 (-1271) [TITLE] AUTOSTART MONITOR MEMORY LOCATION 'TITLE'
$FB0B (-1269) [NBRNCH] MONITOR MEMORY LOCATION 'NBRNCH'
$FB11 (-1263) [INITBL] MONITOR MEMORY LOCATION 'INITBL'
$FB11 (-1263) AUTOSTART MONITOR MEMORY LOCATION 'XLTBL'
$FB19 (-1255) [RTBL] MONITOR MEMORY LOCATION 'RTBL'
$FB1E (-1250) [PREAD] \SE\ MONITOR S/R TO READ PADDLE. X-REG CONTAINS PADDLE NUMBER (0-3) OF PADDLE TO BE READ. PADDLE VALUE TO Y-REG (A- Y-REGS ALTERED)
$FB25 (-1243) [PREAD2] MONITOR MEMORY LOCATION 'PREAD2'
$FB2E (-1234) [RTS2D] MONITOR MEMORY LOCATION 'RTS2D'
$FB2F (-1233) [INIT] \SE\ MONITOR S/R- SCREEN INITIALIZATION (RESET TEXT MODE)
$FB39 (-1223) [SETTXT] \SE\ MONITOR S/R- SET SCREEN TO TEXT MODE. CLOBBERS ACCUMULATOR (A-REG ALTERED)
$FB40 (-1216) [SETGR] \SE\ MONITOR S/R- SET GRAPHIC MODE (GR). THIS INCLUDES SETTING TO MIXED MODE;CLEARING GRAPHICS PART OF SCREEN; AND RESETTING WNDTOP~WNDLFT~WNDWDTH~WNDBTM & TABV (A-REG ALTERED)
$FB43 (-1213) \SE\ MONITOR S/R - ALL OF SETGR EXCEPT SETTING COLOR GRAPHICS DISPLAY MODE
$FB46 (-1210) \SE\ MONITOR S/R - ALL OF SETGR EXCEPT SETTING COLOR GRAPHICS DISPLAY MODE & CLEARING GRAPHICS PART OF SCREEN; I.E. WINDOW & TAB SETTING ONLY
$FB4B (-1205) [SETWND] \SE\ MONITOR S/R- SET NORMAL LOW-RESOLUTION GRAPHICS WINDOW
$FB5B (-1189) [TABV] \SE\ PLACE CURSOR AT LINE (A-REG) COLUMN (CH) SETTING CV AND BASL~H FROM A-REG (A-REG ALTERED)
$FB60~$FB80 (-1184~-1152) [MULPM] \SB\ MONITOR 16-BIT MULTIPLY S/R (NOT IN AUTOSTART ROM). MULTIPLIER IN AUXL~AUXH ($0054~$0055); MULTIPLICAND IN ACL~ACH ($0050~$0051);XTNDL~XTNDH ($0052~$0053) CLEARED TO ZEROS; RESULT GOES TO EXTENDED AC ($0050~$0053). ALSO SEE 'SIGN' AT $002F. (A- X-REGSY-REG ALTERED)
$FB60 (-1184) [MULPM] \SE\ MONITOR - SIGNED 16-BIT MULTIPLY LEAVING SIGN IN LSB OF 'SIGN' (A- X- Y-REGS ALTERED)
$FB60 (-1184) [APPLEII] \SE\ CLEAR SCREEN AND POKE 'APPLE II' INTO FIRST LINE OF TEXT BUFFER (AUTOSTART ROM ONLY) (A- Y-REGS ALTERED)
$FB63 (-1181) [MUL] \SE\ MONITOR - UNSIGNED 16-BIT MULTIPLY S/R (NOT AVAILABLE WITH AUTOSTART ROM). SAME AS MULPM ($FB60) EXCEPT UNSIGNED. SEE 'SIGN' AT $002F (A- X- Y-REGS ALTERED)
$FB65 (-1179) [MUL2] MONITOR MEMORY LOCATION 'MUL2'
$FB65 (-1179) [STITLE] AUTOSTART MONITOR MEMORY LOCATION 'STITLE'
$FB6D (-1171) [MUL3] MONITOR MEMORY LOCATION 'MUL3'
$FB6F (-1169) [SETPWRC] \SE\ SET POWER CONDITION (AUTOSTART ROM ONLY)
$FB76 (-1162) [MUL4] MONITOR MEMORY LOCATION 'MUL4'
$FB78 (-1160) [MUL5] MONITOR MEMORY LOCATION 'MUL5'
$FB78 (-1160) [VIDWAIT] AUTOSTART MONITOR MEMORY LOCATION 'VIDWAIT'
$FB81~$FBC0 (-1151~-1088) MONITOR 16-BIT DIVIDE ROUTINE (NOT IN AUTOSTART ROM)
$FB81 (-1151) [DIVPM] MONITOR SIGNED DIVISION - DIVIDES NUMBER IN EXTENDED AC ($0050~$0053) BY NUMBER IN AUXL~AUXH ($0054~$0055) LEAVING QUOTIENT IN ACL~ACH ($0050~$0051) AND REMAINDER IN $0053. BE CAREFUL OF SIGNS SCALING & OVERFLOW. IF (XTNDL~XTNDH ($0052~$0053))>(AUXL~AUXH ($0054~$0055)) OVERFLOW WILL RESULT
$FB84 (-1148) [DIV] \SE\ MONITOR S/R- UNSIGNED DIVIDE ROUTINE - SAME AS $FB81 (DIVPM) EXCEPT NO SIGNS USED.
$FB86 (-1146) [DIV2] MONITOR MEMORY LOCATION 'DIV2'
$FB88 (-1144) AUTOSTART MONITOR MEMORY LOCATION 'KBDWAIT'
$FBA0 (-1120) [DIV3] MONITOR MEMORY LOCATION 'DIV3'
$FBA4 (-1116) [MD1] MONITOR 16-BIT MULITIPLY/DIVIDE SIGN-PROCESSOR. SETS ABSOLUTE VALUES OF ACL~H MEMORY LOCATION 'MD1' AUXL~H LEAVING RESULTING SIGN IN LSB OF SIGN ($002F)
$FBAF (-1105) [MD2] MONITOR MEMORY LOCATION 'MD2'

```
$FBB4 (-1100) [MD3]                 MONITOR MEMORY LOCATION 'MD3'
$FBC0 (-1088) [MDRTS]               MONITOR MEMORY LOCATION 'MDRTS'
$FBC1 (-1087) [BASCALC] \SE\        MONITOR S/R- CALCULATE TEXT BASE ADDRESS. SET BASL~H TO LEFT END OF SCREEN LINE
                                    (NOT WINDOW LINE) IN A-REG (A-REG ALTERED)
$FBD0 (-1072) [BSCLC2]              MONITOR MEMORY LOCATION 'BSCLC2'
$FBD9 (-1063) [BELL1]               MONITOR MEMORY LOCATION 'BELL1'
$FBDD (-1059)                       SOUNDS BELL IN APPLE REGARDLESS OF OUTPUT DEVICE IN USE (A- Y-REGS ALTERED)
$FBE4 (-1052) [BELL2] \SE\          MONITOR S/R- SOUND BELL (BEEPER)
$FBEF (-1041) [RTS2B]               MONITOR MEMORY LOCATION 'RTS2B'
$FBF0 (-1040) [STOADV] \SE\         MONITOR -- LOAD Y FROM CH; STORE A-REG TO SCREEN AT (BASL)~Y; AND GOTO ADVANCE
                                    ($FBF4) (A- Y-REG ALTERED)
$FBF4 (-1036) [ADVANCE] \SE\        MONITOR S/R- MOVE CURSOR RIGHT; I.E. INCREMENT (CH); COMPARE (CH) WITH (WNDWDTH) GO
                                    TO CR IF CH NOT LESS ELSE RETURN (RTS) (A-REG ALTERED)
$FBF6 (-1034) \SE\                  COMPARE (CH) WITH (WNDWDTH) GO TO CR IF CH NOT LESS ELSE RETURN (RTS) (A-REG
                                    ALTERED)
$FBFC (-1028) [RTS3]                MONITOR MEMORY LOCATION 'RTS3'
$FBFD (-1027) [VIDOUT] \SE\         MONITOR S/R- OUTPUT A-REGISTER AS ASCII ON TEXT SCREEN OR PROCESS CONTROL
                                    CHARACTER. IF (A)<$80 GOTO STOADV;  =$87 SOUND BELL; =$88 GOTO BS; =$8A GOTO LF;
                                    =$8D GOTO CR; >$9F GOTO STOADV; OTHERWISE IGNORE ENTRY SCREEN RTS 1
$FC10 (-1008) [BS] \SE\             MONITOR S/R TO MOVE CURSOR LEFT (BACKSPACE); IF AT START OF LINE MOVE UP TO RIGHT
                                    END OF LINE ABOVE IF POSSIBLE (A-REG ALTERED)
$FC1A (-998) [UP ~ CURSUP] \SE\     MONITOR S/R TO MOVE CURSOR UPWARD (IF POSSIBLE) (A-REG ALTERED)
$FC22 (-990) [VTAB] \SE\            PERFORM A VERTICAL TAB TO ROW SPECIFIED IN A-REG ($0~$17). SET BASL~H FROM CV (AND
                                    WNDLFT) (A-REG ALTERED)
$FC24 (-988) [VTABZ] \SE\           SET BASL~H FROM (A-REG) AND WNDLFT WITHOUT REGARD TO CV (A-REG ALTERED)
$FC2B (-981) [RTS4]                 MONITOR MEMORY LOCATION 'RTS4'
$FC2C (-980) [ESC1] \SE\            ROUTINE (IF A=@ GO TO HOME; =A GO TO ADVANCE; =B GO TO BS (BACKSPACE); =C GO TO LF
                                    (LINEFEED); =D GO TO UP (INVERSE LINEFEED); =E GOTO CLREOL; =F GOTO CLREOP;
                                    =ANYTHING ELSE RTS & IGNORE ENTRY) CALLED BY 'RDCHAR' IF ESCAPE KEY IS INPUTTED.
                                    CALLS APPROPRIATE SCROLL WINDOW SERVICE ROUTINE (IF A=@ GO TO HOME; =A GO TO
                                    ADVANCE; =B GO TO BS (BACKSPACE); =C GO TO LF (LINEFEED); =D GO TO UP (INVERSE
                                    LINEFEED); =E GOTO CLREOL; =F GOTO CLREOP; =ANYTHING ELSE RTS & IGNORE ENTRY) (USES
                                    A-REG)
$FC42 (-958) [CLREOP] \SE\          MONITOR S/R TO CLEAR FROM CURSOR TO END OF PAGE. (A- Y-REGS ALTERED)
$FC46 (-954) [CLEOP1]               MONITOR MEMORY LOCATION 'CLEOP1'
$FC58 (-936) [HOME] \SE\            CLEAR SCROLL WINDOW TO BLANKS. SET CURSOR TO TOP LEFT CORNER (A- Y-REGS ALTERED)
$FC5A (-934) \SE\                   SET CV (CURSOR VERTICAL POSN) FROM A-REG. CLEAR WINDOW TO END OF WINDOW; SET CH=0
                                    (A- Y-REGS ALTERED)
$FC62 (-926) [CR] \SE\              MONITOR S/R TO PERFORM A CARRIAGE RETURN; I.E. LOAD ZERO TO A-REG & CH (A-REG
                                    ALTERED)
$FC66 (-922) [LF] \SE\              MONITOR S/R TO TO PERFORM A LINE FEED; I.E. INCREMENT CV; COMPARE CV TO WNDBTM IF
                                    CV<WNDBTM GOTO VTABZ TO SET BASL~H AND RETURN ELSE DECREMENT CV AND DO SCROLL
                                    (A-REG ALTERED)
$FC70 (-912) [SCROLL] \SE\          MONITOR S/R TO SCROLL UP 1 LINE.  (A- Y-REGS ALTERED)
$FC76 (-906) [SCRL1]                MONITOR MEMORY LOCATION 'SCRL1'
$FC8C (-884) [SCRL2]                MONITOR MEMORY LOCATION 'SCRL2'
$FC95 (-875) [SCRL3]                MONITOR - CLEAR LINE (BASL~H) (WHOLE LINE) THEN SET NEW BASL~H FROM CV & WNDLFT
$FC9C (-868) [CLREOL] \SE\          MONITOR S/R TO CLEAR TO END OF LINE (A- Y-REGS ALTERED)
$FC9E (-866) [CLEOLZ]               MONITOR MEMORY LOCATION 'CLEOLZ'
$FCA0 (-864) [CLEOL2]               MONITOR MEMORY LOCATION 'CLEOL2'
$FCA8 (-856) [WAIT] \SE\            CALL FOR WAIT LOOP. WAIT ESTIMATED AT 2.5A~2+13.5A+13 WAIT CYCLES OF 1.02
                                    MICROSECONDS WHERE A IS CONTENTS OF A-REG WHEN S/R CALLED
```

$FCA9 (-855) [WAIT2] MONITOR MEMORY LOCATION 'WAIT2'
$FCAA (-854) [WAIT3] MONITOR MEMORY LOCATION 'WAIT3'
$FCB4 (-844) [NXTA4] \SE\MONITOR S/R TO INCREMENT A4 (16 BITS) THEN DO NXTA1 (A-REG ALTERED)
$FCBA (-838) [NXTA1] \SE\MONITOR S/R TO INCREMENT A1 (16 BITS). SET CARRY IF RESULT >=A2. (A-REG ALTERED)
$FCC8 (-824) [RTS4B] MONITOR MEMORY LOCATION 'RTS4B'
$FCC9 (-823) [HEADR] MONITOR - WRITES SYNCHRONIZATION MONOTONE WHICH IS FIRST PART OF EVERY CASSETTE TAPE RECORD
$FCD6 (-810) [WRBIT] MONITOR - WRITES A BIT TO CASSETTE TAPE (CALLED BY WRBYTE AND HEADR)
$FCDB (-805) [ZERDLY] MONITOR MEMORY LOCATION 'ZERDLY'
$FCE2 (-798) [ONEDLY] MONITOR MEMORY LOCATION 'ONEDLY'
$FCE5 (-795) [WRTAPE] MONITOR MEMORY LOCATION 'WRTAPE'
$FCEC (-788) [RDBYTE] MONITOR - READS BITS FROM CASSETTE TAPE UNTIL BYTE ACCUMULATED (CALLED BY MONITOR READ MEMORY LOCATION 'RDBYTE' SHAPE TABLE LOAD)
$FCEE (-786) [RDBYT2] MONITOR MEMORY LOCATION 'RDBYT2'
$FCFA (-774) [RD2BIT] MONITOR TWO-EDGE TAPE SENSE; I.E. LOOPS DECREMENTING Y-REG UNTIL HARDWARE HAS INDICATED TWO TRANSITIONS OF TAPE INPUT REGISTER. CONTENTS OF Y-REG ON RETURN COMPARED WITH CONTENTS ON ENTRY MEASURE TIME REQUIRED FOR TRANSITIONS. CALLS RDBIT
$FCFD (-771) [RDBIT] MONITOR - LOOPS DECREMENTING Y-REG UNTIL CASSETTE TAPE INPUT REGISTER CHANGES (EITHER 0=>1 OR 1=>0). BIT VALUE RETURNED IS DETERMINED FROM RESIDUAL COUNT OF Y-REG. CALLED BY RD2BIT AND READ
$FD0C (-756) [RDKEY] \SE\SAME AS RDCHAR EXCEPT BYPASSES ESCAPE KEY MONITOR SUPPORT; PICKS UP AND SAVE THE CHARACTER IN THE SCREEN AREA AT BASL~H CH (LEAVING Y-REG CONTAINING CONTENTS OF CH) IT THEN CHANGES THAT CHARACTER TO BLINKING TO INDICATE CURRENT CURSOR POSN; ASKS FOR NEXT INPUT CHAR TO BE PLACED IN A-REG BY DOING AN INDIRECT JUMP VIA KSWL~H WHICH IS NORMALLY POINTING AT KEYIN. RETURN IS THEREFORE TO THE CALLER OF RDKEY - NOT TO RDKEY ROUTINE ITSELF. SET-UP: A- X- Y-REGS NOT SIGNIFICANT; CV AND BASL~H SHOULD BE COMPATABLE POINTING IN THE SCROLL WINDOW; CH INDICATES HORIZONTAL POSITION WHERE CURSOR WILL BLINK. RESULTS: A-REG CONTAINS THE INPUT CHARACTER (WHICH MAY BE ANY CHARACTER INCLUDING ANY CONTROL KEY OR ESCAPE KEY);X-REG IS UNCHANGED; Y-REG CONTAINS CONTENTS OF CH; CV CH BASL~H REMAIN UNCHANGED (A- X- Y-REGS ALTERED)
$FD1B (-741) [KEYIN] \SE\GETS NEXT KEY INPUT FROM KEYBOARD HARDWARE. REQUIRES LOOP TO TEST THAT KEY HAS INDEED BEEN READ; BY PRESENCE OF $80 BIT. ALSO REQUIRES KEYBOARD STROBE TO BE HIT BEFORE NEXT KEYBOARD INPUT. AUXILLIARY ACTIONS TAKEN BY KEYIN INCLUDE RESTORING TO THE SCREEN AREA THE CHARACTER MODIFIED BY RDKEY TO REMOVE BLINK INSERTED BY RDKEY AND COUNTING UP THE RANDOM NUMBER FIELD~ IGNORING OVERFLOW. SET-UP: X-REG NOT SIGNIFICANT & NOT AFFECTED; A-REG INPUT TO THIS ROUTINE STORED AT (BASL)~Y WHEN A KEY IS PRESSED BEFORE THE A-REG IS FILLED FROM THE KEYBOARD REGISTER; Y-REG USED FOR STORING A-REG IN SCREEN AREA TO (BASL)~Y; CH AND CV NOT REFEENCED;BASL~H ARE USED AS INDICATED IN RDKEY. RESULT: A-REG CONTAINS INPUT FROM KEYBOARD REGISTER; IT IS ONLY ITEM CHANGED (A-REG ALTERED)
$FD21 (-735) [KEYIN2] MONITOR MEMORY LOCATION KEYIN2
$FD2F (-721) [ESC] MONITOR MEMORY LOCATION 'ESC'
$FD35 (-715) [RDCHAR] \SE\CALLS RDKEY TO GET NEXT CHAR PLACED INTO A-REG. IF ESCAPE KEY PRESSED CALLS 'ESC1' FOR ESCAPE KEY PROCESSING; AFTER ESCAPE KEY AND KEY FOLLOWING HAVE BEEN READ & PROCESSED CONTROL RETURNS TO RDCHAR ROUTINE AS IF IT WERE JUST BEING ENTERED (A- X- Y-REGS ALTERED)
$FD3D (-707) [NOTCR] MONITOR MEMORY LOCATION 'NOTCR'
$FD5F (-673) [NOTCR1] MONITOR MEMORY LOCATION 'NOTCR1'
$FD62 (-670) [CANCEL] \SE\MONITOR S/R TO PERFORM A LINE CANCEL (\)
$FD67 (-665) [GETLNZ] \SE\OUTPUT A C/R (THROUGH COUT). GO TO GETLN TO WRITE PROMPT & GET A LINE OF DATA (USUALLY FROM KEYBOARD); ON SET-UP A- X- Y-REGS CH AND BASL~H NOT SIGNIFICANT. CV SHOULD POINT TO A LINE IN SCROLL WINDOW; ON OUTPUT KEYED IN INFO IS IN $200 THRU $200~X WHERE $200~X CONTAINS A CARRIAGE RETURN;A-REG CONTAINS CARRIAGE RETURN;X-REG CONTAINS NUMBER OF CHARACTERS READ EXCLUDING TERMINATING CARRIAGE RETURN;Y-REG CONTAINS CONTENTS OF WNDWDTH; CH CONTAINS ZERO;CV CONTAINS LINE POINTER (CURRENT VALUE);BASL~H CONTAINS MEMORY ADDRESS CORRESPONDING TO CV AND WNDLFT; SCREEN LINE IS BLANKS TO THE RIGHT OF THE END OF ECHOED INPUT (A- X- Y-REGS ALTERED)

$FD6A (-662) [GETLN] \SE\PROMPT & GET LINE OF TEXT. ON CALLING A- X- Y-REGS NOT SIGNIFICANT. CV AND BASL~H SHOULD BE COMPATIBLE POINTING IN THE SCROLL WINDOW. CH INDICATES WHERE ON LINE THE PROMPT CHARACTER IS TO BE PLACED TO BE FOLLOWED BY ECHOED KEYBOARD INPUT; OUTPUT AS FOR GETLNZ {X-REG GETS #CHARS READ. DATA TO $200~$200~X (MAX $2FF) $200~X & Y-REG GET C/R (USES NXTCHAR)} {A- X- Y-REGS ALTERED}
$FD6F (-657) \SE\ MONITOR S/R TO GET LINE OF TEXT FM KEYBD (SAME AS GETLN EXCEPT NO PROMPT!) {A- X- Y-REGS ALTERED}
$FD71 (-655) [BCKSPC] MONITOR MEMORY LOCATION 'BCKSPC'
$FD75 (-651) [NXTCHAR] \SE\TOP POINT IN CHAR INPUT LOOP. SAME EFFECT AS GETLN EXCEPT BYPASS PRINT OF PROMPT CHARACTER; ON SET-UP X-REG SHOULD BE SET TO ZERO TO BEGIN STORING OF INPUT AT $200; A- Y-REGS NOT SIGNIFICANT;CV AND BASL~H SHOULD BE COMPATIBLE POINTING IN THE SCROLL WINDOW; CH INDICATES WHERE ECHOING OF KEYBOARD INPUT IS TO START & SHOULD BE LESS THAN WNDWDTH; RESULTS SAME AS FOR GETLNZ {A- X- Y-REGS ALTERED}
$FD7E (-642) [CAPTST] MONITOR MEMORY LOCATION 'CAPTST'
$FD80 (-640) [INSTDSP] MONITOR S/R TO DISASSEMBLE INSTRUCTION AT PCH/PCL {A- X- Y-REGS ALTERED}
$FD84 (-636) [ADDINP] MONITOR MEMORY LOCATION 'ADDINP'
$FD8E (-626) [CROUT] \SE\MONITOR S/R TO PRINT A CARRIAGE RETURN THROUGH COUT {A- Y-REGS ALTERED}
$FD92 (-622) [PRA1] \SE\PRINT CARRIAGE RET; THEN HEX OF A1H~A1L; THEN MINUS SIGN {A- X- Y-REGS ALTERED}
$FD96 (-618) [PRYX2] \SE\MONITOR S/R TO PRINT CAR RET THEN HEX OF Y-REG & X-REG THEN A DASH {A-REG ALTERED}
$FD99 (-615) \SE\ PRINT HEX OF Y-REG & X-REG THEN MINUS SIGN {A-REG ALTERED}
$FDA3 (-605) [XAM8] \SE\MONITOR S/R TO EXAMINE 8 MEM LOCNS. PRINTS HEX OF MEMORY FROM XXXX TO XXX7 WHERE XXXX IS CONTENTS OF A1L~A1H; Y-REG MUST =0 ON ENTRY {A-REG ALTERED }
$FDAD (-595) [MOD8CHK] MONITOR MEMORY LOCATION 'MOD8CHK'
$FDB3 (-589) [XAM] \SE\MONITOR S/R TO EXAMINE CONTENTS OF MEMORY FROM (A1L~A1H) TO(A2L~A2H). Y-REG=0 BEFORE CALL {A-REG ALTERED}
$FDB6 (-586) [DATAOUT] MONITOR MEMORY LOCATION 'DATAOUT'
$FDC5 (-571) [RTS4C] MONITOR MEMORY LOCATION 'RTS4C'
$FDC6 (-570) [XAMPM] MONITOR MEMORY LOCATION 'XAMPM'
$FDD1 (-559) [ADD] MONITOR MEMORY LOCATION 'ADD'
$FDDA (-550) [PRBYTE] \SE\MONITOR S/R TO PRINT CONTENTS OF A-REG AS 2 HEX DIGITS {A-REG ALTERED}
$FDE3 (-541) [PRHEX] \SE\MONITOR S/R TO PRINT RIGHT NIBBLE OF A-REG AS A SINGLE HEX DIGIT {A-REG ALTERED}
$FDE5 (-539) [PRHEXZ] MONITOR MEMORY LOCATION 'PRHEXZ'
$FDED (-531) [COUT] \SE\PRINT BYTE IN A-REG TO OUTPUT DEVICE SPECIFIED BY 'CSWL' (NORMALLY 'COUT1') {A-REG ALTERED}
$FDF0 (-528) [COUT1] \SE\WRITE BYTE IN A-REG TO SCREEN AT CURSOR POSN (CV)~(CH) USING 'INVFLG' & SUPPORTING CURSOR MOVE
$FDF6 (-522) [COUTZ] \SE\WRITE BYTE FROM A-REG TO SCREEN AT (CV)~(CH) WITH CURSOR MOVE BUT NOT 'INVFLG' {NONE ALTERED}
$FE00 (-512) [BL1] MONITOR & MINIASSEMBLER MEMORY LOCATION 'BL1'
$FE04 (-508) [BLANK] MONITOR MEMORY LOCATION 'BLANK'
$FE0B (-501) [STOR] MONITOR MEMORY LOCATION 'STOR'
$FE17 (-489) [RTS5] MONITOR MEMORY LOCATION 'RTS5'
$FE18 (-488) [SETMODE] MONITOR MEMORY LOCATION 'SETMODE'
$FE1D (-483) [SETMDZ] MONITOR MEMORY LOCATION 'SETMDZ'
$FE20 (-480) [LT] MONITOR MEMORY LOCATION 'LT'
$FE22 (-478) [LT2] MONITOR MEMORY LOCATION 'LT2'
$FE2C (-468) [MOVE] \SE\MONITOR S/R TO PERFORM A MEMORY MOVE (A1-A2 TO A4)(Y-REG MUST =0 AT CALL) {A-REG ALTERED}
$FE36 (-458) [VFY] \SE\MONITOR S/R TO PERFORM A MEMORY VERIFY (A1-A2 TO A4)
$FE58 (-424) [VFYOK] MONITOR MEMORY LOCATION 'VFYOK'
$FE5E (-418) [LIST] \SE\CALL TO DISASSEMBLE 20 INSTRUCTIONS
$FE63 (-413) [LIST2] MONITOR MEMORY LOCATION 'LIST2'
$FE78 (-392) [A1PCLP] MONITOR & MINIASSEMBLER MEMORY LOCATION 'A1PCLP'
$FE7F (-385) [A1PCRTS] MONITOR MEMORY LOCATION 'A1PCRTS'
$FE80 (-384) [SETINV] \SE\MONITOR S/R TO SET VIDEO OUTPUT TO INVERSE
$FE84 (-380) [SETNORM] \SE\MONITOR S/R TO SET VIDEO OUTPUT TO NORMAL (NOT INVERSE)

$FE86 (-378) [SETIFLG] MONITOR MEMORY LOCATION 'SETIFLG'
$FE89 (-375) [SETKBD] MONITOR MEMORY LOCATION 'SETKBD'
$FE8B (-373) [INPORT] MONITOR MEMORY LOCATION 'INPORT'
$FE8D (-371) [INPRT]   MONITOR MEMORY LOCATION 'INPRT'
$FE93 (-365) [SETVID] MONITOR MEMORY LOCATION'SETVID'
$FE95 (-363) [OUTPORT] MONITOR MEMORY LOCATION 'OUTPORT'
$FE97 (-361) [OUTPRT] MONITOR MEMORY LOCATION 'OUTPRT'
$FE9B (-357) [IOPRT]   MONITOR MEMORY LOCATION 'IOPRT'
$FEA7 (-345) [IOPRT1] MONITOR MEMORY LOCATION 'IOPRT1'
$FEA9 (-343) [IOPRT2] MONITOR MEMORY LOCATION 'IOPRT2'
$FEB0 (-336) [XBASIC] \SE\MONITOR S/R TO JUMP TO BASIC
$FEB3 (-333) [BASCONT] \SE\MONITOR S/R TO CONTINUE BASIC
$FEB6 (-330) [GO] \SE\ MONITOR MEMORY LOCATION 'GO'
$FEB9 (-327) \SE\   RESTORE REGISTERS (CALL RESTORE) THEN JMP (PCL) TO CONTINUE EXECUTION (A- X- Y- P-REGS ALTERED)
$FEBF (-321) [REGZ] \SE\MONITOR S/R TO DISPLAY REGISTERS
$FEC2 (-318) [TRACE] \SE\CALL TO PERFORM MONITOR TRACE
$FEC4 (-316) [STEPZ]   MONITOR MEMORY LOCATION 'STEPZ'
$FECA (-310) [USR]   MONITOR MEMORY LOCATION 'USR'
$FECD (-307) [WRITE] \SE\MONITOR S/R TO WRITE DATA FROM MEMORY TO CASSETTE TAPE - FIRST MEMORY LOCATION POINTED TO BY
                      A1L~H ($003C~$003D); LAST BY A2L~H ($003E~$003F). CASSETTE TAPE GETS 10 SECONDS OF TONE
                      HEADER THEN THE DESIGNATED DATA BITS AND ONE CHECKSUM BYTE
$FED4 (-300) [WR1]   MONITOR MEMORY LOCATION 'WR1'
$FEED (-275) [WRBYTE] MONITOR - USES WRBIT TO WRITE 10 BITS TO CASSETTE TAPE
$FEEF (-273) [WRBYT2] MONITOR MEMORY LOCATION 'WRBYT2'
$FEF6 (-266) [CRMON]   MONITOR MEMORY LOCATION 'CRMON'
$FEFD (-259) [READ] \SE\READS DATA FROM CASSETTE TAPE PUTTING FIRST DATA READ INTO LOCATION POINTED TO BY A1L~H
                      ($003C~$003D) AND CONTINUING TO READ UNTIL DATA GOES TO LOCATION POINTED TO BY A2L~H
                      ($003E~$003F). ALSO COMPUTES A RUNNING EXCLUSIVE OR CHECKSUM IN 'CHECKSUM' ($002E)
$FF02 (-254) [READX1] HI-RES GRAPHICS - READ WITHOUT HEADER
$FF0A (-246) [RD2]   MONITOR MEMORY LOCATION 'RD2'
$FF16 (-234) [RD3]   MONITOR MEMORY LOCATION 'RD3'
$FF2D (-211) [PRERR] \SE\MONITOR S/R TO PRINT "ERR" AND SOUND BELL. (A- Y-REGS(?) ALTERED)
$FF3A (-198) [BELL] \SE\MONITOR S/R TO SOUND BELL IN CURRENT OUTPUT DEVICE (WHETHER IT IS APPLE OR EXTERNAL PRINTER)
                      (A--REG ALTERED)
$FF3F (-193) [RESTORE] \SE\RESTORE 6502 REGISTERS: ($0045)=>A-Reg; ($0046)=>X-Reg; ($0047)=>Y-Reg; ($0048)=>P-Reg;
                      (A- X- Y- P-REGS ALTERED)
$FF44 (-188) [RESTR1] MONITOR MEMORY LOCATION 'RESTR1'
$FF4A (-182) [SAVE] \SE\MONITOR S/R TO SAVE 6502 REGISTERS: (A-REG)=>$0045; (X-REG)=>$0046; (Y-REG)=>$0047;
                      (P-REG)=>$0048; (S-REG)=>$0049 (NONE)
$FF4C (-180) [SAV1]   MONITOR MEMORY LOCATION 'SAV1'
$FF58 (-168) [IORTS]   JSR HERE TO FIND OUT WHERE ONE IS. SETS OVERFLOW FLAG
$FF59 (-167) [RESET] \SE\CALL HERE HAS SAME EFFECT AS PUSHING RESET BUTTON
$FF65 (-155) [MON] \SE\MONITOR S/R- NORMAL ENTRY TO 'TOP' OF MONITOR WHEN RUNNING (BEEPS!)
$FF69 (-151) [MONZ] \SE\MONITOR S/R TO RESET AND ENTER MONITOR (NO BEEP)
$FF70 (-144) \SE\   MONITOR S/R TO SCAN INPUT BUFFER
$FF73 (-141) [NXTITM] MONITOR MEMORY LOCATION 'NXTITM'
$FF7A (-134) [CHRSRCH] MONITOR MEMORY LOCATION 'CHRSRCH'
$FF7C (-132) [ZMODE]   MONITOR & MINIASSEMBLER MEMORY LOCATION 'ZMODE'
$FF8A (-118) [DIG]   MONITOR MEMORY LOCATION 'DIG'
$FF90 (-112) [NXTBIT] MONITOR MEMORY LOCATION 'NXTBIT'
$FF98 (-104) [NXTBAS] MONITOR MEMORY LOCATION 'NXTBAS'

```
HEX LOCN (DEC LOCN) [NAME]  \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

$FFA2 (-94) [NXTBS2]    MONITOR MEMORY LOCATION 'NXTBS2'
$FFA7 (-89) [GETNUM]    MONITOR & MINIASSEMBLER MEMORY LOCATION 'GETNUM'
$FFAD (-83) [NXTCHR]    MONITOR - TOP POINT IN GETLN CHARACTER INPUT LOOP;RDCHAR CALLED TO GET CHAR INTO A-REG; ON
                        RETURN A-REG TESTED FOR PRESENCE OF CTRL-U (RIGHT ARROW); IF SO A-REG LOADED FROM SC/REEN MEMORY
                        ASSUMING Y-REG CONTAINS SAME VALUE AS CH; IF A-REG VAL >$DF LOWER-CASE LETTER CONVERTED TO UPPER
                        CASE; IF CHAR IS A C/R IT IS PRINTED THROUGH COUT AND RTS EXIT OF COUT WILL GIVE CONTROL BACK TO
                        CALLING PROGRAM W/ X-REG INDICATING INPUT CHAR COUNT +1; THAT IS INPUT IS IN LOCNS $200 THRU
                        $200~X WHERE $200~X CONTAINS A C/R; ON SET-UP A- X- Y-REGS NOT SIGNIFICANT; CV & BASL~H SHOULD
                        BE COMPATIBLE (POINTING IN THE SCROLL WINDOW);CH INDICATES HORIZ POSN IN SCROLL WINDOW WHERE
                        CURSOR WILL BE INDICATED BY BLINKING. ON RETURN CALLER A-REG WILL CONTAIN KEY VALUE;Y-REG WILL
                        CONTAIN CONTENTS OF CH;X-REG WILL CONTAIN SAME VALUE AS INPUT; CV CH & BASL~H WILL HAVE CHANGE
                        ONLY IF AN ESCAPE KEY SEQUENCE HAS BEEN PERFORMED
$FFBE (-66) [TOSUB]     MONITOR & MINIASSEMBER MEMORY LOCATION 'TOSUB'
$FFC7 (-57) [ZMODE]     MONITOR MEMORY LOCATION 'ZMODE'
$FFCC (-52) [CHRTBL]    MONITOR & MINIASSEMBLER MEMORY LOCATION 'CHRTBL' (TABLE USED TO DECODE MONITOR KEYBOARD INPUT)
$FFE3~$FFE9 (-29~-23) [SUBTBL] \PB\TABLE OF SUBROUTINE ADDRESSES -1 (INDEX PC WITH TBL ITEM FOR S/R ENTRY): (ADDRESS
                         MSB = $FE; LSB = TABLE ENTRY +1)
$FFE3 (-29) [SUBTBL]      'SUBTBL' L.S.B. ADDRESS-1 OF BASCONT SUBROUTINE
$FFE4 (-28) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF USR SUBROUTINE (M.S.B.=$FE)
$FFE5 (-27) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF REGZ SUBROUTINE (M.S.B.=$FE)
$FFE6 (-26) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF TRACE SUBROUTINE (M.S.B.=$FE)
$FFE7 (-25) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF VFY SUBROUTINE (M.S.B.=$FE)
$FFE8 (-24) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF INPRT SUBROUTINE (M.S.B.=$FE)
$FFE9 (-23) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF STEPZ SUBROUTINE (M.S.B.=$FE)
$FFEA (-22) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF OUTPRT SUBROUTINE (M.S.B.=$FE)
$FFEB (-21) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF XBASIC SUBROUTINE (M.S.B.=$FE)
$FFEC (-20) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF SETMODE SUBROUTINE (M.S.B.=$FE)
$FFED (-19) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF SETMODE SUBROUTINE (M.S.B.=$FE)
$FFEE (-18) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF MOVE SUBROUTINE (M.S.B.=$FE)
$FFEF (-17) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF LT SUBROUTINE (M.S.B.=$FE)
$FFF0 (-16) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF SETNORM SUBROUTINE (M.S.B.=$FE)
$FFF1 (-15) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF SETINV SUBROUTINE (M.S.B.=$FE)
$FFF2 (-14) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF LIST SUBROUTINE (M.S.B.=$FE)
$FFF3 (-13) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF WRITE SUBROUTINE (M.S.B.=$FE)
$FFF5 (-11) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF READ SUBROUTINE (M.S.B.=$FE)
$FFF6 (-10) \P1\          'SUBTBL' L.S.B. ADDRESS-1 OF SETMODE SUBROUTINE (M.S.B.=$FE)
$FFF7 (-9) \P1\           'SUBTBL' L.S.B. ADDRESS-1 OF SETMODE SUBROUTINE (M.S.B.=$FE)
$FFF8 (-8) \P1\           'SUBTBL' L.S.B. ADDRESS-1 OF CRMON SUBROUTINE (M.S.B.=$FE)
$FFFA~$FFFB (-6~-5) \P2\  FULL (16-BIT) ADDRESS OF NMI (NON-MASKABLE INTERRUPT) VECTOR
$FFFC~$FFFD (-4~-3) \P2\  FULL (16-BIT) ADDRESS OF RESET VECTOR
$FFFE~$FFFF (-2~-1) \P2\  FULL (16-BIT) ADDRESS OF IRQ (INTERUPT REQUEST) VECTOR
-------------------------------------------------------------------------------------------------------------------------------
```

GAZETTEER

**** For Apple //e specific information, see Appendix A ****

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

(-32K)   (-7938~-7932) [$E0FE~$E104] \P5\APPLESOFT FIVE-BYTE FLOATING POINT CONSTANT -32768 (-2^16)
1002   (1002) [$03EA] \SE\          DOS 3.2 ENTRY POINT FOR ROUTINE THAT UPDATES I/O HOOK TABLES IN $0036~$0039. (JMP
                                    $A851 - SAVES ADDRESSES OF CHARACTER INPUT & OUTPUT ROUTINES CURRENTLY IN USE AND
                                    RECONNECTS DOS I/O)
(3DOG)   (976) [$03D0] \SE\         DOS 3.2 SOFT-ENTRY POINT; I.E. RE-ENTRY POINT (3DOG) FOR RE-INITIALIZATION SAVING ALL
                                    VARIABLES & DATA OF CURRENT BASIC PROGRAM (JMP $9DBF)
995   (995) [$03E3] \SE\            DOS 3.1~3.2 ENTRY POINT TO LOAD Y~A WITH ADDRESS OF IOBLK
A   (74) [$004A] \P1\               DOS DISK SYSTEM FORMATTER DUMMY LOCATION FOR TIMING PURPOSES AND SCRATCH. DOS WILL
                                    REPAIR IN INIT COMMAND; USER MUST REPAIR IF RWTS FORMATTER CALLED DIRECTLY
(A/S POINTERS)   (80~97) [$0050~$0061] \PB\GENERAL PURPOSE POINTERS FOR APPLESOFT (PB)
(A/S RESVD)   (10~22) [$000A~$0016] \PB\APPLESOFT RESERVED BLOCK IN PAGE ZERO
A1L-A1H   (60~61) [$003C~$003D] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A1. MANY USES INCLUDE SOURCE POINTER
                                    DURING MONITOR MOVE
A1PCLP   (-392) [$FE78]             MONITOR & MINIASSEMBLER MEMORY LOCATION 'A1PCLP'
A1PCRTS   (-385) [$FE7F]            MONITOR MEMORY LOCATION 'A1PCRTS'
A2L-A2H   (62~63) [$003E~$003F] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A2. USED IN CALLING LIST OF MANY MONITOR
                                    SUBROUTINES SUCH AS MOVE & CASSETTE ROUTINES
A3L-A3H   (64~65) [$0040~$0041] \P1\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A3. USED IN CALLING LIST OF MOST MONITOR
                                    SUBROUTINES
A4L-A4H   (66~67) [$0042~$0043] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A4.  USED IN CALLING LIST OF SOME
                                    MONITOR SUBROUTINES
A5L-A5H   (68~69) [$0044~$0045] \P2\MONITOR GENERAL USAGE SUBROUTINE PARAMETER A5. USED MOSTLY BY SINGLE-CYLCLE & TRACE
ABS (FPABS)   (-5201) [$EBAF] \SE\ APPLESOFT FP - TAKES ABS OLUTE VALUE OF NUMBER IN FAC & LEAVES RESULT IN FAC
ABSWAP   (-3017) [$F437] \SE\       TAKE ABSOLUTE VALUE OF FP1; THEN SWAP FP1 WITH FP2 (FP1=$00F8;$FP2=$00F4) (A- X-REGS
                                    ALTERED)
~ABS~   (-6326) [$E74A] \SE\        INTEGER BASIC ENTRY TO GET ABSOLUTE VALUE OF A NUMBER
AC   (80~83) [$0050~$0053] \P4\     32-BIT EXTENDED ACCUMULATOR USED IN MONITOR 16-BIT MULT & DIVIDE
ACADR   (-3810) [$F11E]             HI-RES GRAPHICS 2-BYTE TAPE READ SETUP
ACC   (69) [$0045] \P1\             USER A-REG SAVED HERE ON BRK TO MONITOR & DURING TRACE
ACL~ACH   (80~81) [$0050~$0051] \P2\OLD MONITOR (NOT AUTOSTART). USED BY 16 BIT MULT & DIVIDE ROUTINES AS
                                    PSEUDO-ACCUMULATOR
ACL~ACH   (206~207) [$00CE~$00CF] \P2\INTEGER BASIC MAIN ACCUMULATOR
ADD   (-3035) [$F425] \SE\          ADD 3-BYTE M1 TO 3-BYTE M2 AND LEAVE RESULT IN M1 (NOT FP ADD BUT USED IN FP PKG)
                                    (A- X-REGS ALTERED)
ADD   (-559) [$FDD1]                MONITOR MEMORY LOCATION 'ADD'
ADDINP   (-636) [$FD84]             MONITOR MEMORY LOCATION 'ADDINP'
~ADDITION~   (-6267) [$E785] \SE\   INTEGER BASIC ENTRY POINT TO ADDITION FUNCTION
ADDON   (-9832) [$D998] \SE\        APPLESOFT - ADD Y-REG TO TXTPTR
ADVANCE   (-1036) [$FBF4] \SE\      MONITOR S/R- MOVE CURSOR RIGHT; I.E. INCREMENT (CH); COMPARE (CH) WITH (WNDWDTH) GO
                                    TO CR IF CH NOT LESS ELSE RETURN (RTS) (A-REG ALTERED)
ALLDONE   (15911) [$3E27] \SL\      DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'ALLDONE'
ALLDONE   (-16826~-16825) [$BE46~$BE47] DOS 3.3 - SKIP OVER SET CARRY INSTRUCTION IN 'HNDLERR'
AMPERV   (1013~1015) [$03F5~$03F7]  APPLESOFT - HOLDS JMP (JUMP) INSTRUCTION TO S/R WHICH HANDLES & COMMANDS.  DEFAULT
                                    $4C $58 $FF (JUMP TO $FF58)
APPLEII   (-1184) [$FB60] \SE\      CLEAR SCREEN AND POKE 'APPLE II' INTO FIRST LINE OF TEXT BUFFER (AUTOSTART ROM
                                    ONLY) (A- Y-REGS ALTERED)
ARG   (165~170) [$00A5~$00AA] \PB\  APPLESOFT SECONDARY FLOATING POINT ACCUMULATOR (USES 6-BYTE UNPACKED MATH PACKAGE
                                    FORMAT DESCRIBED BELOW)
ARGEXP   (165) [$00A5] \P1\         EXPONENT PART OF ARG. SINGLE BYTE SIGNED NUMBER IN EXCESS $80 FORM (SIGNED VALUE
                                    HAS $80 ADDED TO IT)
ARYTAB   (107~108) [$006B~$006C] \P2\APPLESOFT ARRAY TABLE POINTER (POINTS TO BEGINNING OF ARRAY SPACE)
-------------------------------------------------------------------------------------------------------------------------------
(-32K) - ARYTAB                  Prof. Luebbert's "What's Where in the Apple"                  ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

"ASC"   (-3299) [$F31D] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO PERFORM THE ASC (ASCII) FUNCTION
ATN   (-3938) [$F09E] \SE\            APPLESOFT FP COMPUTE THE ARCTANGENT OF NUMBER IN FAC. RESULT TO FAC. MODIFIES INDEX
                                      XORFPSGN AND MANY OTHER FP LOCNS
AUTOINCL"AUTOINCH   (244"245) [$00F4"$00F5] \P2\INTEGER BASIC MEMORY LOCATIONS 'AUTOINCL"AUTOINCH' (CURRENT AUTO LINE
                                      NUMBER VALUE)
AUTOLNL"AUTOLNH   (246"247) [$00F6"$00F7] \P2\INTEGER BASIC MEMORY LOCATIONS 'AUTOLNL"AUTOLNH'
AUTOMODE   (248) [$00F8] \P1\         INTEGER BASIC MEMORY LOCATION 'AUTOMODE' (THE AUTOMODE FLAG)
(AUTOSTART RESVD)   (32"79) [$0020"$004F] \PB\AUTOSTART MONITOR RESERVED LOCATIONS
"AUTO"   (-6174) [$E7E2] \SE\         INTEGER BASIC ENTRY TO AUTO LINE NUMBERING FUNCTION
AUXL"AUXH   (84"85) [$0054"$0055] \P2\  OLD MONITOR (NOT AUTOSTART) - USED FOR 16-BIT MULT & DIVIDE AS AUXILLIARY
                                      REGISTER
AUXL"AUXH   (218"219) [$00DA"$00DB] \P2\INTEGER BASIC MEMORY LOCATIONS 'AUXL"AUXH' (AULILIARY COUNTER)
AYINT (FP=>INT)   (-7924) [$E10C] \SE\  APPLESOFT - IF FAC SUITABLE FOR CONVERSION TO INTEGER (FAC<32767 & FAC>-32768)
                                      THEN PERFORM QINT (RESET Y-REG=0)
(AYPOSINT +FP=>INT)   (-7928) [$E108] \SE\APPLESOFT - SAME AS AYINT ($E10C) EXCEPT FAC MUST BE POSITIVE
(BAD SUBSCRPRT)   (-7786) [$E196] \SE\  APPLESOFT - PRINT "BAD SUBSCRIPT" AND HALT AT APPLESOFT LEVEL (]
BAS2L"BAS2H   (42"43) [$002A"$002B] \P2\USED DURING SCROLLING AS DESTINATION LINE POINTER AS EACH LINE IS MOVED TO
                                      POSITION ABOVE CURRENT
BASCALC   (-1087) [$FBC1] \SE\        MONITOR S/R- CALCULATE TEXT BASE ADDRESS. SET BASL"H TO LEFT END OF SCREEN LINE
                                      (NOT WINDOW LINE) IN A-REG (A-REG ALTERED)
BASCONT   (-333) [$FEB3] \SE\         MONITOR S/R TO CONTINUE BASIC
BASIC   (-8192) [$E000]               APPLESOFT - 'HARD' OR 'COLD' OR 'CONTROL-B' ENTRY POINT (COMPLETE
                                      REINITIALIZATION. START WITH A TOTALLY FRESH SLATE.)
BASIC   (-8192) [$E000]               INTEGER BASIC - 'HARD' OR 'COLD' OR 'CONTROL-B' ENTRY POINT  (COMPLETE
                                      REINITIALIZATION. START WITH A TOTALLY FRESH SLATE)
BASIC2   (-8189) [$E003] \SE\         INTEGER BASIC - 'SOFT' OR 'WARM' OR 'CONTROL-C' OR 'ENTRY2' ENTRY POINT
                                      (REENTRY WITHOUT REINITIALIZATION OF SYMBOL-TABLE" VARIABLES OR DATA)
BASL"BASH   (40"41) [$0028"$0029] \P2\  MEMORY ADDRESS FOR LEFT END CHARACTER POS'N OF CURRENT TEXT LINE
BCKSPC   (-655) [$FD71]               MONITOR MEMORY LOCATION 'BCKSPC'
BDRAW   (-11462) [$D33A] \SE\         HI-RES GRAPHICS  DRAW1 S/R CALL: PARAM= XO"YO"COLR"SHAPE"ROT"SCALE
BDRAW1   (-11465) [$D337] \SE\        HI-RES GRAPHICS LINE S/R CALL: PARAM=XO"YO"COLR
BELL   (-198) [$FF3A] \SE\            MONITOR S/R TO SOUND BELL IN CURRENT OUTPUT DEVICE (WHETHER IT IS APPLE OR
                                      EXTERNAL PRINTER) (A--REG ALTERED)
BELL1   (-1063) [$FBD9]               MONITOR MEMORY LOCATION 'BELL1'
BELL2   (-1052) [$FBE4] \SE\          MONITOR S/R- SOUND BELL (BEEPER)
BGND   (-11471) [$D331] \SE\          HI-RES GRAPHICS BKGND S/R CALL  PARAM= COLR
BKGND   (-12270) [$D012] \P1\         HI-RES GRAPHICS MEMORY LOCATION 'BKGND' (ROM)
BKGND   (-3086) [$F3F2] \SE\          APPLESOFT HI-RES - CLEAR HI-RES SCREEN TO LAST PLOTTED COLOR
BKGND0   (-12272) [$D010]             HI-RES GRAPHICS 'BKGND0 (HCOLOR1 SET FOR BLACK BKGND)
BL1   (-512) [$FE00]                  MONITOR & MINIASSEMBLER MEMORY LOCATION 'BL1'
BLANK   (-508) [$FE04]                MONITOR MEMORY LOCATION 'BLANK'
BLIN1   (-11500) [$D314] \SE\         HI-RES GRAPHICS LINE S/R CALL PARAM= XO"YO"COLR
BLTU   (-11373) [$D393] \SE\          APPLESOFT BLOCK TRANSFER UTILITY. MAKES ROOM BY MOVING EVERYTHING FORWARD.
                                      Y-REG(MSB)&A-REG(LSB) AND HIGHDS=DEST OF HIGH ADR;LOWTR=LOWEST ADDR TO BE
                                      MOVED;HIGHTR=HIGHEST ADDR TO BE MOVED+1
(BOOT DISK #)   (1528) [$05F8] \P1\   CONTAINS SLOT # OF DISK CONTROLLER CARD FROM WHICH ANY ACTIVE DOS 3.2 WAS BOOTED
BPLOT   (-11506) [$D30E] \SE\         HI-RES GRAPHICS PLOT S/R CALL PARAM= XO"YO"COLR
BPOSN   (-11527) [$D2F9] \SE\         HI-RES GRAPHICS POSN S/R CALL PARAM= XO"YO"COLR
BRANCH   (-1283) [$FAFD]              MONITOR MEMORY LOCATION 'BRANCH'
"BRANCH"   (-6420) [$E6EC] \SE\       INTEGER BASIC ENTRY POINT TO BRANCH (GET LO/HI THEN JSR)
BRATE   (1144+S) [$0478+S] \P1\       EXAMPLE: SERIAL INTERFACE BAUD QUANTUM RATE. $1= 19200 BAUD;$40=300 BAUD
-------------------------------------------------------------------------------------------------------------------------------
ASC - BRATE                      Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

BREAK   (-1390) [$FA92] \SE\              MONITOR S/R - BREAK HANDLER
BRKV    (1008~1009) [$03F0~$03F1] \P2\    AUTOSTART ROM BREAK VECTOR - DEFAULT VALUE $FA59
BS   (-1008) [$FC10] \SE\                 MONITOR S/R TO MOVE CURSOR LEFT (BACKSPACE); IF AT START OF LINE MOVE UP TO
                                          RIGHT END OF LINE ABOVE IF POSSIBLE (A-REG ALTERED)
BSCLC2    (-1072) [$FBD0]                 MONITOR MEMORY LOCATION 'BSCLC2'
BUF INBUFF    (512~767) [$0200~$02FF] \HB\KEYIN (CHARACTER INPUT) BUFFER (MONITOR~INTEGER BASIC~APPLESOFT BASIC)
BUFPTR    (62~63) [$003E~$003F] \P2\      DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER 'BUFPTR' (POINTS TO DATA BUFFER IN
                                          RWTS)
BXSAV   (803) [$0323]                     HI-RES GRAPHICS 'BXSAV'
BYTE   (1656+S) [$0678+S]                 EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S INPUT OUTPUT BUFFER
~CALL~    (-4448) [$EEA0] \SE\            INTEGER BASIC ENTRY POINT TO CALL A SUB/ROT FUNCTION
CANCEL    (-670) [$FD62] \SE\             MONITOR S/R TO PERFORM A LINE CANCEL (\)
CAPTST    (-642) [$FD7E]                  MONITOR MEMORY LOCATION 'CAPTST'
CAT    (-6761) [$E597] \SE\               APPLESOFT - CONCATENATE TWO STRINGS. FACMO (MSB) & FACLO (LSB) POINT TO FIRST
                                          STRING'S DESCRIPTOR & TXTPTR POINTS TO '+'
CH   (36) [$0024] \P1\                    CURSOR HORIZONTAL DISPLACEMENT FROM WNDLFT: RANGE 0 TO (WNDWDTH)-1
CHAR    (249) [$00F9] \P1\                INTEGER BASIC MEMORY LOCATION 'CHAR' (CURRENT CHARACTER)
CHAR1    (-1612) [$F9B4]                  MONITOR & MINIASSEMBER MEMORY LOCATION 'CHAR1'
CHAR2    (-1606) [$F9BA]                  MONITOR & MINIASSEMBLER MEMORY LOCATION 'CHAR2'
CHARAC    (13) [$000D]                    APPLESOFT - USED BY STRLT2 STRING UTILITY
CHGIT    (16326) [$3FC6] \SL\             DOS 3.2 DISK FORMATTER INTERIOR LABEL 'CHGIT'
CHKCLS    (-8520) [$DEB8] \SE\            APPLESOFT CLOSE PARENTHESIS CHECK - CHECKS TXTPTR FOR ')'. USES SYNCHR.
CHKCOM    (-8514) [$DEBE] \SE\            APPLESOFT COMMA CHECK - CHECKS TXTPTR FOR COMMA. USES SYNCHR.
CHKNUM    (-8854) [$DD6A] \SE\            APPLESOFT - MAKE SURE FAC IS NUMERIC (SEE CHKVAL)
CHKOPN    (-8517) [$DEBB] \SE\            APPLESOFT OPEN PARENTHESIS CHECK - CHECKS TXTPTR FOR '('. USES SYNCHR.
CHKSTR    (-8852) [$DD6C] \SE\            APPLESOFT - MAKE SURE FAC IS STRING (SEE CHKVAL)
CHKSUM    (46) [$002E] \P1\               LOCN WHERE CHECKSUM IS ACCUMULATED DURING CASSETTE TAPE READ
CHKVAL    (-8851) [$DD6D] \SE\            APPLESOFT - IF C SET CHECK FOR STRINGS;C CLEAR CHECK FOR NUMRIC VBL. TYPE
                                          MISMATCH ERROR OCCURS IF C AND FAC DON'T AGREE
CHRGET    (177) [$00B1] \SE\              APPLESOFT CHRGET S/R CALL - GETS NEXT SEQUENTIAL CHR OR TOKEN - LOADS A-REG
                                          FROM LOCN SPECIFIED BY TXTPTR($00B8~$00B9 & INCREMENTS TXTPTR. CARRY IS RESET
                                          TO ZERO IF CHARACTER IS A DIGIT OTHERWISE IT IS SET; ZERO FLAT SET IF CHAR =0
                                          (END OF LINE SIGN) OR $3A (END OF STATEMENT SIGN ':') OTHERWISE RESET (X-
                                          Y-REGS NOT ALTERED)
CHRGET    (177~200) [$00B1~$00C8] \SB\    APPLESOFT CHRGET ROUTINE. CALLED WHEN WANTS ANOTHER CHARACTER (X- Y-REGS NOT
                                          ALTERED)
CHRGOT    (183) [$00B7] \SE\              APPLESOFT CHRGOT S/R CALL. CHRGET INCREMENTS TXTPTR. CHRGOT DOES NOT
CHRSRCH    (-134) [$FF7A]                 MONITOR MEMORY LOCATION 'CHRSRCH'
CHRTBL    (-52) [$FFCC]                   MONITOR & MINIASSEMBLER MEMORY LOCATION 'CHRTBL' (TABLE USED TO DECODE MONITOR
                                          KEYBOARD INPUT)
CIN   (-22120~-22119) [$A998~$A999] \P2\DOS 3.1  INTERNAL HOOK ENTRY ADDRESS TO INPUT A CHARACTER
CLEARC    (-10644) [$D66C] \SE\           APPLESOFT INITIALIZATION - THE 'CLEAR' COMMAND. CLEARS VARIABLES & STACK
CLEOL2    (-864) [$FCA0]                  MONITOR MEMORY LOCATION 'CLEOL2'
CLEOLZ    (-866) [$FC9E]                  MONITOR MEMORY LOCATION 'CLEOLZ'
CLEOP1    (-954) [$FC46]                  MONITOR MEMORY LOCATION 'CLEOP1'
CLRAN0    (-16295) [$C059] \FF\           VALUE <>0 WHEN GAME AN0 IS RESET (CLEARED). POKE 0 TO SET GAME I/O OUTPUT AN0
                                          (0.3V AT PIN 15)
CLRAN1    (-16293) [$C05B] \FF\           POKE 0 TO SET GAME I/O OUTPUT AN1 (0.3V AT PIN 14)
CLRAN2    (-16291) [$C05D] \FF\           POKE 0 TO SET GAME I/O OUTPUT AN2 (0.3V AT PIN 13)
CLRAN3    (-16289) [$C05F] \FF\           POKE 0 TO SET GAME I/O OUTPUT AN3 0.3V AT PIN 12)
CLREOL    (-868) [$FC9C] \SE\             MONITOR S/R TO CLEAR TO END OF LINE (A- Y-REGS ALTERED)
-------------------------------------------------------------------------------------------------------------------------------
BREAK - CLREOL                  Prof. Luebbert's "What's Where in the Apple"                  ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

CLREOP    (-958) [$FC42] \SE\      MONITOR S/R TO CLEAR FROM CURSOR TO END OF PAGE. (A- Y-REGS ALTERED)
CLRROM    (-12289) [$CFFF] \H1\    SPECIAL LOCATION RECOGNIZED BY PERIPHERAL CARDS AS SIGNAL TO TURN OFF FLIP FLOPS WHICH
                                   DISABLE EXPANSION ROM
CLRSC2    (-1992) [$F838] \SE\     CLEAR LINES 0 THRU (Y-REG) 40 COLUMNS WIDE TO BLACK IN LO-RES GRAPHICS OR INVERSE @ IN
                                   TEXT PAGE 1 (A- Y-REGS ALTERED)
CLRSC3    (-1988) [$F83C] \SE\     CLEAR LO-RES GRAPHICS PARTIAL TOP LEFT: X-COORD 0 THRU (Y-REG); Y-COORD 0 THRU ($002D)
                                   (A- Y-REGS ALTERED)
CLRSCR    (-1998) [$F832] \SE\     CLEAR LO-RES GRAPHICS SCREEN1 TO BLACK (INVERSE @ IN TEXT MODE) MIXED GRAPHICS AREA
                                   ONLY (A- Y-REGS ALTERED)
CLRSCR    (-1998) [$F832] \SE\     MONITOR S/R TO CLEAR SCREEN - GRAPHICS MODE FULL SCREEN) (A- Y-REGS ALTERED)
CLRTOP    (-1994) [$F836] \SE\     CLEAR TOP 20 LINES PAGE1 TO INVERSE @ IN TEXT; BLACK IN LO-RES GRAPHICS (40 LO-RES
                                   GRAPHIC 'LINES') (A- Y-REGS ALTERED)
"CLR"    (-6729) [$E5B7] \SE\     INTEGER BASIC ENTRY POINT TO CLEAR OUT VARIABLE WORK SPACE
CNUM    (68"69) [$0044"$0045]     DOS - POINTS TO AVAILABLE BUFFER IN OPEN. ALSO USED AS ARITHMETIC REGISTER BY DOS
                                   FIRST & SECOND LEVEL ROUTINES
COLLSN    (810) [$032A] \P1\       COLLISION COUNT FROM DRAW"DRAW1
COLOR    (48) [$0030] \P1\         LOW-RES COLOR GRAPHICS COLOR CODE (FOR PLOT/HLIN/VLIN FUNCTIONS) - CONTAINS SELECTED
                                   COLOR VALUES FOR TWO LOW-RES GRAPHICS 'LINES' ONE IN EACH NIBBLE OF BYTE
"COLOR"    (-4530) [$EE4E] \SE\    INTEGER BASIC ENTRY TO ROUTINE TO SET COLOR VALUE FOR LO-RES
COMBYTE    (-6324) [$E74C] \SE\    APPLESOFT - CHECK FOR COMMA & GET A BYTE IN X-REG. USES CH<COM& BETBYT. ON ENTRY
                                   TXTPTR POINTS TO COMMA
(COMMAND TBL)    (26756) [$6884] \PB\DOS 3.2 COMMAND TABLE (32K APPLE ONLY!)
"COMMA"    (-6207) [$E7C1] \SE\     INTEGER BASIC ENTRY POINT TO COMMA FUNCTION
(COMPRTYP)    (22) [$0016] \P1\     APPLESOFT - PARAMETER TO CONTROL TYPE OF COMPARISON MADE BY FLOATING POINT COMPARISON
                                   ROUTINE AT $DF6A (1:> ;2:= ;3:>= ;4:< ;5:<> ;6:<=)
CONINT    (-6405) [$E6FB] \SE\     APPLESOFT FP - CONVERT FAC INTO SINGLE BYTE IN X-REG & FACLO.NORMAL EXIT THRU CHRGET.
                                   IF FAC<0 OR FAC>255 ILLEGAL QUANT ERROR
CONL"CONH    (242"243) [$00F2"$00F3] \P2\INTEGER BASIC MEMORY LOCATIONS 'CONL"CONH' (CONTINUE POINTER)
CONSYNC    (16074) [$3ECA] \SL\     DOS 3.2 DISK FORMATTER - LABEL AT POINT WHERE CONSTRUCTION OF SYNC BEGINS
CONT    (-10088) [$D898] \SE\       APPLESOFT - MOVES OLDTXT & OLDLIN INTO TXTPTR & CURLIN
CONUPK    (-5661) [$E9E3] \SE\      APPLESOFT FP - LOAD ARG FROM MEMORY POINTED TO BY Y-REG & A-REG. ON EXIT A & Z
                                   REFLECT FACEXP. MODIFIES INDEX & XORFPSGN. (RESET Y-REG=0)
CONWAIT    (15743) [$3D7F] \SL\     DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL - STARTS CONSTANT WAIT DELAY
                                   LOOP RETURN POINT
"CON"    (-3318) [$F30A] \SE\       INTEGER BASIC ENTRY TO ROUTINE TO CONTINUE EXECUTION
COPY    (-9545) [$DAB7] \SE\        APPLESOFT - FREE STRING POINTED TO BY Y-REG (MSB) & A-REG (LSB) & MOVE IT TO MEM LOC
                                   POINTED TO BY FORPNT
CORRECTSECT    (15895) [$3E17] \SL\DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF CODE WHICH ASSUME
                                   SECTOR CORRECTLY CHOSEN AND JUMPS TO APPROPRIATE SUBROUTINE TO READ OR WRITE
CORRECTVOL    (15878) [$3E06] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL WHICH ASSUMES CORRECT VOLUME
                                   HAS BEEN DETECTED AND CHECKS FOR SECTOR SELECTION
COS    (-4118) [$EFEA] \SE\         APPLESOFT FP - COMPUTE THE COSINE OF THE NUMBER IN FAC. RESULT TO FAC. MODIFIES INDEX
                                   CHARAC COMPRTYP XORFPSGN AND MANY OTHER FP LOCNS
COUNT - CSUM    (44) [$002C] \P1\   DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER (RETURNS CHECKSUM)
COUNT    (249) [$00F9] \P1\         INTEGER BASIC MEMORY LOCATION 'COUNT'
COUNTH    (29) [$001D] \P1\         HI-RES GRAPHICS HIGH-ORDER BYTE OF STEP COUNT FOR LINE
COUT    (-22122"-22121) [$A996"$A997] \P2\DOS 3.1 INTERNAL HOOK ENTRY ADDRESS TO OUTPUT A CHARACTER
COUT    (-531) [$FDED] \SE\         PRINT BYTE IN A-REG TO OUTPUT DEVICE SPECIFIED BY 'CSWL' (NORMALLY 'COUT1') (A-REG
                                   ALTERED)
COUT1    (-528) [$FDF0] \SE\        WRITE BYTE IN A-REG TO SCREEN AT CURSOR POSN (CV)"(CH) USING 'INVFLG' & SUPPORTING
                                   CURSOR MOVE
-------------------------------------------------------------------------------------------------------------------------------
CLREOP - COUT1                  Prof. Luebbert's "What's Where in the Apple"                  ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

COUTZ   (-522) [$FDF6] \SE\        WRITE BYTE FROM A-REG TO SCREEN AT (CV)~(CH) WITH CURSOR MOVE BUT NOT 'INVFLG' (NONE
                                   ALTERED)
CR   (-926) [$FC62] \SE\           MONITOR S/R TO PERFORM A CARRIAGE RETURN; I.E. LOAD ZERO TO A-REG & CH (A-REG ALTERED)
CRCTVOL   (-16858~-16827) [$BE26~$BE45] DOS 3.3 - CHECK TO SEE IF SECTOR CORRECT. USE 'ILEAV' TABLE ($BFB8) FOR
                                     SOFTWARE SECTOR INTERLEAVING. IF WRONG SECTOR TRY AGAIN AT 'TRYADR ($BDC1). IF
                                     WRITE BRANCH TO 'WRIT' ($BE51). OTHERWISE GOTO 'READ16' ($B8DC). IF GOOD READ
                                     CALL 'POSTNB16' ($B8C2) AND RETURN TO CALLER WITH NO ERROR
CRDO   (-9477) [$DAFB] \SE\        APPLESOFT - PRINT A CARRIAGE RETURN
CRFLAG   (213) [$0CD5] \P1\        INTEGER BASIC MEMORY LOCATION 'CRFLAG' (CARRIAGE RETURN FLAG)
CRMON   (-266) [$FEF6]             MONITOR MEMORY LOCATION 'CRMON'
CROUT   (-626) [$FD8E] \SE\        MONITOR S/R TO PRINT A CARRIAGE RETURN THROUGH COUT (A- Y-REGS ALTERED)
CSWL~CSWH   (54~55) [$0C36~$0037] \P2\MONITOR OUTPUT REG & OUTPUT HOOK TO DOS; I.E. ADDRESS OF ROUTINE WHICH IS TO
                                     RECEIVE AND DISPOSE OF OUTPUT CHARACTERS. RESET 0 CTRL-P & PR#0 SET THIS LOCN TO
                                     $FDF0 (MONITOR OUTPUT TO SCREEN); S CTRL-P & PR#S SET THIS LOCN TO $CS00 (SLOT S
                                     ROM)
CURLIN   (117~118) [$0075~$0076] \P2\APPLESOFT - LINE # OF LINE CURRENTLY BEING EXECUTED NOTE: HI BYTE OF CURLIN TESTED
                                     BY DOS FOR DIRECT-DEFERRED MODE USAGE - BYTE SET TO $FF IN DIRECT. IF CONTENTS OF
                                     $AAB6<>0 AND IF PROMPT=']' OR IF THIS LOCN CONTAINS $FF DOS ASSUMES DIRECT MODE
                                     AND WILL NOT DO OPEN OR OTHER DIRECT MODE COMMANDS
CURTRK   (1144) [$0478] \P1\       DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) PARAMETER CURRENT TRACK (LAST TRACK
                                     'SEEK'-ED)
CV   (37) [$0025] \P1\             CURSOR VERTICAL POSITION RELATIVE TO TOP OF SCREEN: RANGE 0-23 ($0~$17)
DATA   (-9835) [$D995] \SE\        APPLESOFT - MOVE TXTPTR TO END OF STATEMENT; LOOKS FOR ':' OR EOL(0).
DATAN   (-9821) [$D9A3] \SE\       APPLESOFT - CALCULATE OFFSET IN Y-REG FROM TXTPTR TO NEXT ':' OR EOL(0)
DATAOUT   (-586) [$FDB6]           MONITOR MEMORY LOCATION 'DATAOUT'
DATLIN   (123~124) [$007B~$007C] \P2\APPLESOFT CURRENT LINE # FROM WHICH DATA IS BEING READ
DATPTR   (125~126) [$007D~$007E] \P2\POINTS TO ABS LOC IN MEM FROM WHICH DATA IS BEING READ BY APPLESOFT
~DELETE~   (-7313) [$E36F] \SE\    INTEGER BASIC ENTRY POINT TO DELETE LINES OF TEXT X~Y
DELL~DELH   (226~227) [$00E2~$00E3] \P2\INTEGER BASIC MEMORY LOCATIONS 'DELL~DELH' (DELETE LINE POINTER)
(DEV SELECT 0)   (-16256~-16241) [$C080~$C08F] 16 MEMORY LOCNS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #0. WHEN
                                     ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED. SINCE SLOT #0 IS COMMON
                                     AREA USED IN COMMON FOR PARAMETERS OF INTEREST TO ALL SLOTS
(DEV SELECT 1)   (-16240~-16225) [$C090~$C09F] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #1.
                                     WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED
(DEV SELECT 2)   (-16224~-16209) [$C0A0~$C0AF] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #2.
                                     WHEN ADDRESS PIN 41 TELLS DEVICE IT IS SELECTED
(DEV SELECT 3)   (-16208~-16193) [$C0B0~$C0BF] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #3.
                                     WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED
(DEV SELECT 4)   (-16192~-16177) [$C0C0~$C0CF] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #4.
                                     WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED
(DEV SELECT 5)   (-16176~-16161) [$C0D0~$C0DF] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #5.
                                     WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED
(DEV SELECT 6)   (-16160~-16145) [$C0E0~$C0EF] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #6.
                                     WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED
(DEV SELECT 7)   (-16144~-16129) [$C0F0~$C0FF] 16 MEMORY LOCATIONS ALLOCATED TO USE OF PERIPHERAL DEVICE IN SLOT #7.
                                     WHEN ADDRESSED PIN 41 TELLS DEVICE IT IS SELECTED
DEVCTBL   (60~61) [$003C~$003D] \P2\   DOS RWTS  DEVICE IN READ-WRITE TRACK-SECTOR PARAMETER POINTING TO DEVICE
                                     TABLE. PRESET TO 'PTRSDEST' = POINTER TO DESTINATION  DEVICE IN DEVICE
                                     TABLE. NOT A SYNONYM FOR BUFPTR
DEVCTBL   (60~61) [$003C~$003D]    DOS RWTS (READ-WRITE TRACK-SECTOR) DEVICE TABLE ~ SYNONYM FOR BUFPTR
DIG   (-118) [$FF8A]               MONITOR MEMORY LOCATION 'DIG'
-------------------------------------------------------------------------------------------------------------------------------
COUTZ - DIG                        Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
"DIMSTR"   (-7888) [$E130] \SE\          INTEGER BASIC ENTRY POINT TO DIMENSION A STRING FOR MEMORY
"DIMVARB"   (-4322) [$EF1E] \SE\         INTEGER BASIC ENTRY TO ROUTINE TO DIMENSION A VARIABLE
DISKID   (-1275) [$FB05]                 AUTOSTART MONITOR MEMORY LOCATION 'DISKID'
DIV   (-1148) [$FB84] \SE\               MONITOR S/R- UNSIGNED DIVIDE ROUTINE - SAME AS $FB81 (DIVPM) EXCEPT NO SIGNS
                                         USED.
DIV10   (-5547) [$EA55] \SE\             APPLESOFT FP - DIVIDE FAC BY 10. RETURNS POSITIVE NUMBERS ONLY
DIV2   (-1146) [$FB86]                   MONITOR MEMORY LOCATION 'DIV2'
DIV3   (-1120) [$FBA0]                   MONITOR MEMORY LOCATION 'DIV3'
"DIVIDE"   (-4336) [$EF10] \SE\          INTEGER BASIC ENTRY TO DIVIDE FUNCTION
DIVPM   (-1151) [$FB81]                  MONITOR SIGNED DIVISION - DIVIDES NUMBER IN EXTENDED AC ($0050"$0053) BY
                                         NUMBER IN AUXL"AUXH ($0054"$0055) LEAVING QUOTIENT IN ACL"ACH ($0050"$0051)
                                         AND REMAINDER IN $0053. BE CAREFUL OF SIGNS SCALING & OVERFLOW. IF
                                         (XTNDL"XTNDH ($0052"$0053))>(AUXL"AUXH ($0054"$0055)) OVERFLOW WILL RESULT
(DIVZEROPRT)   (-5407) [$EAE1] \SE\      APPLESOFT - PRINT "DIVISION BY ZERO" THEN HALT AT APPLESOFT (]) LEVEL
DONEDSK   (16312) [$3FB8] \SL\           DOS 3.2 DISK FORMATTER INTERIOR LABEL AT POINT WHERE DISK IS COMPLETED AND
                                         NO ERRORS HAVE BEEN DETECTED
(DOS 3.1 COMMAND TBL)   (-22560"-22429) [$A7E0"$A863] \PB\DOS 3.1 COMMAND TABLE (DOS 3.1 - 48K APPLE ONLY!)
(DOS 3.1 ERROR MSGS)   (-22323"-22144) [$A8CD"$A980] \PB\DOS 3.1 ERROR MSG TABLE (DOS 3.1 - 48K APPLE ONLY!)
(DOS 3.2 ERR MSGS)   (26996) [$6974] \PB\        DOS 3.2 ERROR MESSAGES (32K APPLE ONLY!)
(DOS 3.2/3.3 COMMAND TBL)  [L)$A884"$A908] \PB\  DOS 3.2 (48K) COMMAND NAME TABLE OF DOS COMMAND DECODER (TABLE-DRIVEN
                                                 COMMAND PARSER). CONTAINS NAMES OF DOS COMMANDS WITH LAST BYTE OF
                                                 EACH NAME HAVING HIGH (7TH) BIT SET; OTHER BYTES HAVE IT CLEAR. THIS
                                                 PERMITS CLOSE PACKING FOR SEQUENTIAL SEARCH. EOT IS $00 BYTE
(DOS 3.2/3.3 ERROR MSGS)  [)$A971"$AA3E] \PB\    DOS 3.2/3.3 ERROR MESSAGES (DOS 3.2/3.3 - 48K APPLE ONLY!)
DRAW   (-2559) [$F601] \SE\                      APPLESOFT HI-RES - DRAW SHAPE POINTED TO BY Y-REG(MSB)&X-REG(LSB) BY
                                                 INVERTING EXISTING COLOR OF DOTS THE SHAPE DRAWS OVER. A-REG=ROTATION
                                                 FACTOR
DRIVENO   (53) [$0035] \P1\                      DOS DISK DRIVE NO
DRIVERR   (16307) [$3FB3] \SL\                   DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF CLEANUP IF
                                                 DRIVE ERROR IS DETECTED
DRVOEN   (-16246) [$C08A] \P1\                   DOS 3.2 READ\WRITE TRACK"SECTOR (RWTS) PACKAGE PARAMETER 'DRVOEN'
                                                 (DRIVE 0 ENABLE)
DRV1EN   (-16245) [$C08B] \P1\                   DOS 3.2 READ\WRITE TRACK"SECTOR (RWTS) PACKAGE PARAMETER 'DRV1EN'
                                                 (DRIVE 1 ENABLE)
DRV1TRK   (1272+S) [$04F8+S] \P1\                EXAMPLE: 'DRV1TRK' = DISK DRIVE 1 CURRENT TRACK (VALUE = 2*TRACK#);
                                                 DOS 3.2 PARAMETER FOR DISK IN SLOT #S
DRVERR   (15838) [$3DDE] \SL\                    DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE
                                                 FOR CLEANUP WHEN DRIVE ERROR DETECTED
DRVERR   (-16892"-16886) [$BE04"$BE0A] \SE\      DOS 3.3 - CLEAN UP STACK & STATUS REG; LOAD A-REG WITH $40 (DRIVE
                                                 ERROR) AND GOTO 'HNDLERR' ($BE48)
DRVSEL   (15719) [$3D67] \SL\                    DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL
DSCTMP   (157"159) [$009D"$009F] \P3\            APPLESOFT TEMPORARY STRING DESCRIPTOR (SEE VALTYP & TEMPPT)
DSKF2   (16043) [$3EAB] \SL\                     DOS 3.2 DISK FORMATTER LABEL AT POINT WHERE MOTOR IS RUNNING AND ON
                                                 TRACK 0. BEGINS CODE WHICH FORMATS THIS TRACK
DSKFORM   (16028) [$3E9C] \SE\                   DOS 3.2 DISK FORMATTER ENTRY POINT - TURN MOTOR ON & FILL TRACK WITH
                                                 SYNC
DSKFORM   (16028"16089) [$3E9C"$3ED9] \SB\       DOS 3.2 DISK FORMATTER MODULE TO FILL TRACK WITH SYNC
DSKFORM   (16028"16340) [$3E9C"$3FD4] \SB\       DOS 3.2 DISK FORMATTER PACKAGE
DSKFORM   (-16721"-16628) [$BEAF"$BF0C] \SB\     DOS 3.3 - INIT COMMAND HANDLER
"DSP"   (-3324) [$F304] \SE\             INTEGER BASIC ENTRY TO ROUTINE TO DISPLAY A VARIABLE SET
DVOTRK   (1144+S) [$0478+S] \P1\                 EXAMPLE:'DRVOTRAK'= DISK DRIVE 0 CURRENT TRACK (VALUE = 2*TRACK#);
                                                 DOS 3.2 PARAMETER FOR DISK IN SLOT #S
```

```
DXL~DXH    (80~81) [$0050~$0051] \P2\          HI-RES GRAPHICS DELTA-X FOR HLIN SHAPE
DY    (82) [$0052] \P1\                        HI-RES GRAPHICS DELTA-Y FOR HLIN SHAPE
E    (252~254) [$00FC~$00FE] \P3\              MONITOR & FLOATING POINT ROUTINES MEMORY LOC 'E' (3 BYTE MANTISSA
                                               EXTENTION OF FP ACCUMULATOR 1)
EL~EH    (84~85) [$0054~$0055] \P2\           HI-RES GRAPHICS ERROR FOR HLIN
ENDCHR    (14) [$000E]                         APPLESOFT - USED BY STRLT2 STRING UTILITY
ERR    (-2682) [$F586]                         MINIASSEMBLER MEMORY LOCATION 'ERR'
ERR    (-1883) [$F8A5]                         MONITOR MEMORY LOCATION 'ERR'
ERR2    (-2680) [$F588]                        MINIASSEMBLER MEMORY LOCATION 'ERR2'
ERR3    (-2791) [$F519]                        MINIASSEMBLER MEMORY LOCATION 'ERR3'
ERR4    (-2639) [$F5B1]                        MINIASSEMBLER MEMORY LOCATION 'ERR4'
ERRDIR    (-7418) [$E306] \SE\                 APPLESOFT - CAUSES ILLEGAL DIRECT ERROR IF PROGRAM NOT RUNNING (X-REG
                                               ALTERED)
ERRFLG    (208) [$00D0] \P1\                   ERROR FLAG. ON IF BIT 7 SET (PEEK(216)>127). POKE 0 TO CLEAR.
ERRFLG    (216) [$00D8] \P1\                   APPLESOFT ERROR FLAG: $80 IF ONERR ACTIVE. SET TO 0 TO DISABLE 'ONERR
                                               GOTO'
ERRLIN    (218~219) [$00DA~$00DB] \P2\         APPLESOFT LINE # WHERE ERROR OCCURED
ERRNUM    (222) [$00DE] \P1\                   APPLESOFT - WHEN ERROR OCCURS~ TYPE-OF-ERROR CODE APPEARS HERE - SEE
                                               MANUAL FOR CODE NUMBER MEANINGS
ERROR    (-11246) [$D412] \SE\                 APPLESOFT ERROR PROCESSING - CHECKS ERRFLG AND JUMPS TO HNDLERR IF
                                               ONERR IS ACTIVE OTHERWISE PRINTS ERROR MSG BASED ON CODE IN X-REG
~ERRORMESS*~    (-7232) [$E3C0] \SE\           INTEGER BASIC ENTRY POINT - INPUT ERROR MESSAGE
~ERRORMESS~    (-7200) [$E3E0] \SE\            INTEGER BASIC ENTRY POINT TO PRINT ERROR MESSAGE AND GOTO MAINLINE
ERRPOS    (220~221) [$00DC~$00DD] \P2\         APPLESOFT TEXPTR SAVE FOR HNDLERR SUBROUTINE
ERRSTK    (223) [$00DF] \P1\                   APPLESOFT STACK POINTER VALUE BEFORE ERROR OCCURED
ESC    (-721) [$FD2F]                          MONITOR MEMORY LOCATION 'ESC'
ESC1    (-980) [$FC2C] \SE\                    ROUTINE (IF A=@ GO TO HOME; =A GO TO ADVANCE; =B GO TO BS
                                               (BACKSPACE); =C GO TO LF (LINEFEED); =D GO TO UP (INVERSE LINEFEED);
                                               =E GOTO CLREOL; =F GOTO CLREOP; =ANYTHING ELSE RTS & IGNORE ENTRY)
                                               CALLED BY 'RDCHAR' IF ESCAPE KEY IS INPUTTED. CALLS APPROPRIATE
                                               SCROLL WINDOW SERVICE ROUTINE (IF A=@ GO TO HOME; =A GO TO ADVANCE;
                                               =B GO TO BS (BACKSPACE); =C GO TO LF (LINEFEED); =D GO TO UP (INVERSE
                                               LINEFEED); =E GOTO CLREOL; =F GOTO CLREOP; =ANYTHING ELSE RTS &
                                               IGNORE ENTRY) (USES A-REG)
ESDF0    (15975) [$3E67] \SL\                  DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL 'WASD0'
(EVAL EXPR =>INT)    (-7931) [$E105] \SE\      APPLESOFT - EVALUATE EXPRESSION POINTED TO BY TXTPTR ($00B8~$00B9)
                                               AND CONVERT RESULT (WHICH MUST BE NON-NEGATIVE) TOA TWO-BYTE INTEGER
                                               IN FACMO~FACLO ($00A0~$00A1)
EXCNT    (70) [$0046] \P1\                     DOS DISK SYSTEM FORMATTER GENERAL COUNTER
EXP    (-4343) [$EF09] \SE\                    APPLESOFT FP - RAISE E TO THE FAC POWER. RESULT TO FAC. MODIFIES
                                               INDEX CHARAC COMPRTYP XORFPSGN AND MANY OTHER FP LOCNS
~EXP~    (-3215) [$F371] \SE\                  INTEGER BASIC ENTRY TO ROUTINE TO EXPONENTIATE (RAISE TO A POWER)
FAC    (157~163) [$009D~$00A3] \P6\            APPLESOFT MAIN FLOATING-POINT ACCUMULATOR (USES 6-BYTE UNPACKED MATH
                                               PACKAGE FORMAT DESCRIBED BELOW)
(FAC/ARG AND)    (-8363) [$DF55] \SE\          APPLESOFT - LET FAC = FAC 'AND' ARG; I.E. FAC=1 ONLY IF BOTH FAC &
                                               ARG <>0; IF EITHER FAC OR ARG OR BOTH =0 THEN FAC=0
(FAC/ARG COMPARE)    (-8342) [$DF6A] \SE\      APPLESOFT - COMPARERS FAC WITH ARG. TYPE OF COMPARISON CONTROLLED BY
                                               $0016. IF CONDITION MET FAC SET TO ONE; ELSE FAC RESET TO ZERO
(FAC/ARG OR)    (-8369) [$DF4F] \SE\           APPLESOFT - LET FAC = FAC 'OR' ARG; I.E. FAC=1 IF EITHER FAC OR ARG
                                               OR BOTH <>0; FAC=0 ONLY IF BOTH FAC & ARG = 0
FACEXP    (157) [$009D] \P1\                   EXPONENT BYTE OF FAC. SIGNED NUMBER IN EXCESS $80 FORM (SIGNED VALUE
                                               HAS $80 ADDED)
```

FACHO (158) [$009E] \P1\ HIGH ORDER BYTE OF MANTISSA OF FAC
FACLO (161) [$00A1] \P1\ LOW ORDER BYTE OF MANTISSA OF FAC
FACMO (160) [$00A0] \P1\ MIDDLE ORDER BYTE OF MANTISSA OF FAC
FACMOH (159) [$009F] \P1\ MIDDLE ORDER HIGH BYTE OF MANTISSA OF FAC
FACMO~FACLO (160~161) [$00A0~$00A1] \P2\POINTER TO STRING DESCRIPTOR USED IN STRING UTILITIES
(FACSIGN) (162) [$00A2] \P1\ SINGLE BYTE SIGN OF FAC. WHILE IN MATH PKG SIGN IS KEPT IN SGN WHERE ONLY BIT 7 IS SIGNIFICANT
FADD (FPADD) (-6210) [$E7BE] \SE\ APPLESOFT FP - MOVE THE FP NUMBER IN MEMORY POINTED TO BY Y-REG & A-REG INTO ARG AND FALL INTO FADDT (FPADD). MODIFIES INDEX & XORFPSGN
FADD (-2962) [$F46E] \SE\ FLOATING POINT NUMBER IN FP1 ADDED TO THAT IN FP2. NORMALIZED RESULT LEFT IN FP1 {A- X-REGS ALTERED}
FADDH (-6240) [$E7A0] \SE\ APPLESOFT FP - ADD 1/2 TO FAC (1/2 IN $EE64)
FADDT (-6207) [$E7C1] \SE\ APPLESOFT FP - ADD FAC AND ARG. ON ENTRY A-REG AND ZERO FLAG REFLECT FACEXP. RESULT TO FAC
FAKEMON (-2755) [$F53D] MINIASSEMBLER MEMORY LOCATION 'FAKEMON'
FAKEMON2 (-2748) [$F544] MINIASSEMBLER MEMORY LOCATION 'FAKEMON2'
FAKEMON3 (-2760) [$F538] MINIASSEMBER MEMORY LOCATION 'FAKEMON3'
FAKESCT (16192) [$3F40] \SL\ DOS 3.2 DISK FORMATTER INTERIOR LABEL 'FAKESCT' AT BEGINNING OF CODE TO WRITE FAKE SECTOR
FCBFOP ZPGWRK V NPE (64~65) [$0040~$0041] DOS - USED AS GENERAL POINTER BY 1ST LEVEL (COMMAND DECODE) ROUTINES IN DOS
FCOMP (-5198) [$EBB2] \SE\ APPLESOFT FP - COMPARE FAC AND PACKED NUMBER IN MEMORY POINTED TO BY Y-REG & A-REG. ON EXIT A=1 IF MEM<FAC;A=0 IF MEM=FAC;A=$FF IF MEM>FAC
FCOMPL (-2908) [$F4A4] \SE\ VALUE OF FLOATING POINT NUMBER IN FP1 IS NEGATED THEN NORMALIZED {A- X-REGS ALTERED}
FDIV (FPDIV) (-5530) [$EA66] \SE\ APPLESOFT FP - MOVE THE FP NUMBER IN MEMORY POINTED TO BY R-REG & A-REG INTO ARG AND FALL INTO FDIVT. ALTERS INDEX & XORFPSGN
FDIVT (FPDIV2) (-5527) [$EA69] \SE\ APPLESOFT FP - DIVIDE ARG BY FAC. ON ENTRY A-REG AND Z REFLECT FACEXP. RESULT IN FAC. XORFPSGN SHOULD BE COMPUTED BEFORE CALL
FILLCNT - SCTR (75) [$004B] \P1\ DOS DISK SYSTEM FORMATTER GENERAL COUNTER & SECTOR NUMBER
FIN (-5046) [$EC4A] \SE\ APPLESOFT - INPUT FP NUMB INTO FAC FROM CHRGET. ASSUMES 6502 REGS HAVE BEEN SET UP BY CHRGET THAT FETCHED 1ST DIGIT
FINDOP (-2789) [$F51B] MINIASSEMBLER MEMORY LOCATION 'FINDOP'
FIRST (240) [$00F0] \P1\ APPLESOFT - USED BY UTILITY PLOTFNS FOR DESTINATION OF FIRST NUMBER OF LO-RES PLOT COORDINATES
FIX (-2496) [$F640] \SE\ FROMFLOATING POINT NUMBER IN FP1 EXTRACT INTEGER. PJT HIGH-ORDER BYTE IN M1;LOW-ORDER IN M1+1 {A- X-REGS ALTERED}
FLAG (228) [$00E4] INTEGER BASIC MEMORY LOCATION 'FLAG' (GENERAL FLAG BYTE)
FLAGS (2040+S) [$07F8+S] \P1\ EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S OPERATION MODE
FLOAT (-5229) [$EB93] \SE\ APLESOFT FP - FLOAT THE SIGNED INTEGER IN A-REG INTO FAC
FLOAT (-2991) [$F451] \SE\ CONVERT INTEGER (HIGH BYTE IN M1;LOW BYTE IN M1+1;M1+2 CLEARED) TO NORMALIZED FL POINT EQUIV IN FP1 {A-REG ALTERED}
FMT (68) [$0044] \P1\ MINIASSEMBLER MEMORY LOCATION 'FMT'
FMT1 (-1694) [$F962] MONITOR MEMORY LOCATION 'FMT1'
FMT2 (-1626) [$F9A6] MONITOR MEMORY LOCATION 'FMT2'
FMUL (-2932) [$F48C] \SE\ FLOATING POINT MULTIPLY S/R: MUTIPLICAND IN FP1; MULTIPLIER IN FP2; SIGNED NORMALIZED PRODUCT IN FP1 {A- X- Y-REGS ALTERED}
FMUL (-2894) [$F4B2] \SE\ FL PT DIVIDE S/R: NORM DIVIDEND IN FP2;NORM DIVIDER IN FP1;SIGNED NORM FP QUOTIENT TO FP1 {A- X- Y-REGS ALTERED}
FMULT (FPMULT) (-5761) [$E97F] \SE\ APPLESOFT FP - MOVE THE FP NUMBER IN MEMORY POINTED TO BY Y-REG & A -REG INTO ARG AND FALL INTO FMULTT (FPMULT). ALTERS INDEX XORFPSGN
FMULTT (-5758) [$E982] \SE\ APPLESOFT FP - MULTIPLY FAC AND ARG. ON ENTRY A-REG & ZERO FLAG REFLECT FACEXP. RESULT TO FAC. XORFPSGN MUST BE COMPUTED BEFORE CALL

FNDLIN (-10726) [$D61A] \SE\	APPLESOFT - SEARCHES PROGRAM FOR LINE WHOSE NUMBER IS IN LINNUM. ON EXIT IF CARRY SET LOWTR POINTS TO LINK FIELD OF DESIRED LINE; IF NOT LOWTR TO NEXT HIGHER LINE
FNDOP2 (-2787) [$F51D]	MINIASSEMBLER MEMORY LOCATION 'FNDOP2'
FORM1 (-2599) [$F5D9]	MINIASSEMBLER MEMORY LOCATION 'FORM1'
FORM2 (-2597) [$F5DB]	MINIASSEMBLER MEMORY LOCATION 'FORM2'
FORM3 (-2568) [$F5F8]	MINIASSEMBLER MEMORY LOCATION 'FORM3'
FORM4 (-2567) [$F5F9]	MINIASSEMBLER MEMORY LOCATION 'FORM4'
FORM5 (-2566) [$F5FA]	MINIASSEMBLER MEMORY LOCATION 'FORM5'
FORM6 (-2552) [$F608]	MINIASSEMBLER MEMORY LOCATION 'FORM6'
FORM7 (-2547) [$F60D]	MINIASSEMBLER MEMORY LOCATION 'FORM7'
FORM8 (-2526) [$F622]	MINIASSEMBLER MEMORY LOCATION 'FORM8'
FORM9 (-2511) [$F631]	MINIASSEMBLER MEMORY LOCATION 'FORM9'
FORMAT (46) [$002E] \P1\	USED BY MINIASSEMBLER & DISASSEMBLER TO SPECIFY FORMAT OF INSTRUCTION FOR DISPLAY PURPOSES
FORMDSK (-16883~-16625) [$BE0D~$BF0F]	DOS 3.3 - JUMP TO 'DSKFORM' ($BEAF)
FORNDX (251) [$00FB] \P1\	INTEGER BASIC MEMORY LOCATION 'FORNDX' (FOR-NEXT LOOP INDEX)
FORPNT (133~134) [$0085~$0086] \P2\	APPLESOFT GENERAL POINTER. SEE COPY SUBROUTINE FOR EXAMPLE
~FOR~ (-5830) [$E93A] \SE\	INTEGER BASIC ENTRY TO ROUTINE TO HANDLE 'FOR' LOOP INITIALIZATION
FOUT (256~272) [$0100~$0110] \PB\	FOUT BUFFER
FOUT (-4812) [$ED34] \SE\	CREATES A STRING IN FBUFFR EQUIVALENT IN VALUE TO FAC. ON EXIT Y-REG &A-REG POINT TO THE STRING. FAC SCRAMBLED
FP1 (244~247) [$00F4~$00F7] \P4\	MONITOR & FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR 2 (CONTAINS X2 & M2)
FP1 (248~254) [$00F8~$00FE] \P6\	OLD (NON-APPLESOFT) FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR FP1 (CONTAINS X1 M1 AND E (EXTENSION))
FPWRT (FPEXP) (-4457) [$EE97] \SE\	APPLESOFT FP EXPONENTATION (ARG TO FAC POWER) ON ENTRY A-REG & ZERO FLAG SHOULD REFLECT VALUE OF FACEXP. RESULT TO FAC. MODIFIES MANY FP LOCNS
FRESPC (113~114) [$0071~$0072] \P2\	APPLESOFT TEMPORARY POINTER FOR STRING-STORAGE ROUTINES
FRESTR (-6659) [$E5FD] \SE\	APPLESOFT - MAKE SURE THAT LAST FAC RESULT WAS A STRING & FALL INTO FREFAC
FRETMP (-6652) [$E604] \SE\	APPLESOFT - FREE A TEMPORARY STRING. ON ENTRY POINTER TO DESCRIPTOR IS IN Y-REG (MSB) & X-REG (LSB)
FRETMS (-6603) [$E635] \SE\	APPLESOFT - FREE TEMPORARY DESCRIPTOR W/O FREEING UP THE STRING. Y-REG (MSB) & X-REG(LSB) POINT TO DESCRIPTOR TO BE FREED. ON EXIT Z SET IF ANYTHING FREED
FRETOP (111~112) [$006F~$0070] \P2\	APPLESOFT POINTER TO END OF STRING STORAGE OR TOP OF USER-AVAILABLE FREE SPACE. DEFAULTS TO HIMEM - USUALLY $BFFF FOR 48K APPLE)
FRMEVL (-8837) [$DD7B] \SE\	APPLESOFT - EVAL FORMULA AT TXTPTR USING CHRGET & LEAVE RESULT IN FAC. ON ENTRY TXTPTR POINTS TO 1ST CHAR OF FORMULA
FRMEVL (-8837) [$DD7B] \SE\	APPLESOFT - EVAL FORMULA AT TXTPTR USING CHRGET. IF FORMULA IS STRING LITERAL FRMEVL GOBBLES OPENING QUOTE AND EXECUTES STRLIT & ST2TXT
FRMNUM (-8857) [$DD67] \SE\	APPLESOFT - EVALUATE EXPRESSION POINTED TOBY TXTPTR ($0088~$0089) (POINTS TO 1ST CHAR OF FORMULA). PUT RESULT INTO FAC & MAKE SURE IT IS A NUMBER
FRMWSYNC (16096) [$3EE0] \SL\	DOS 3.2 DISK FORMATTER INTERIOR LABEL 'FRMWSYNC'
FSUB (FPSUB) (-6233) [$E7A7] \SE\	APPLESOFT - MOVE FP NUMBER IN MEMORY POINTED TO BY Y-REG &A-REG INTO ARG AND FALL INTO FSUB (FPSUB)T
FSUB (-2968) [$F468] \SE\	FLOATING POINT SUBTRACTION MINUEND IN FP1;SUBTRAHEND IN FP2;NORMALIZED DIFFERENCE TO FP1 (A- X-REGS ALTERED)
FSUBT (-6230) [$E7AA] \SE\	APPLESOFT - FP SUBTRACT FAC FROM ARG. ON ENTRY A-REG & 6502 ZERO FLAG REFLECT FACEXP. RESULT TO FAC
GARBAG (-7036) [$E484] \SE\	APPLESOFT GARBAGE COLLECTOR - MOVES ALL CURRENTLY USED STRINGS UP IN MEMORY AS FAR AS POSSIBLE

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

GBASCALC   (-1977) [$F847] \SE\     COMPUTE GRAPHICS BASE MEMORY ADDRESS FOR LINE IN A-REG (NOTE: 2 LO-RES GRAPHICS
                                    LINES PER TEXT LINE SO (A)= LINE/2); SET GBASL~H (A-REG ALTERED)
GBASL~GBASH   (38~39) [$0026~$0027] \P2\MEMORY ADDRESS OF LEFT END POINT OF DESIRED LINE FOR LO-RES PLOT (SET BY
                                    GBASCALC)
GBCALC   (-1962) [$F856]            MONITOR MEMORY LOCATION 'GBCALC'
GDBUFS   (-10951) [$D539] \SE\      APPLESOFT - PUT ZERO AT END OF INPUT BUFFER (BUF) AND MASK OFF MOST SIGNIFICANT BIT
                                    ON ALL BYTES. ON ENTRY X-REG=END OF INPUT LINE (A- X- Y-REGS ALTERED)
~GET16BIT~   (-6379) [$E715] \SE\   INTEGER BASIC ENTRY TO GET A 16-BIT VALUE
GETADR   (-6318) [$E752] \SE\       APPLESOFT FP - CONVERT FAC (-65535 TO 65535) INTO 2-BYTE INTEGER (0-65535) IN
                                    LINNUM. 'WRAPAROUND' OCCURS IF VALUE IN FAC TOO BIG (A- Y-REGS ALTERED)
GETARYPT   (-2087) [$F7D9] \SE\     APPLESOFT - READ VAR NAME FROM CHRGET & FIND IT IN MEMORY.ON EXIT VAL OF VAR IN
                                    VARPNT AND Y-REG(MSB)&A-REG(LSB)
GETBYT   (-6408) [$E6F8] \SE\       APPLESOFT - EVAL FORMULA AT TXTPTR. LEAVE RESULT IN FAC AND FALL INTO CONINT. AT
                                    ENTRY TXTPTR POINTS TO FIRST CHAR IN FORMULA FOR FIRST NUMBER PLOTFNS PUTS FIRST
                                    NUMBER IN FIRST AND SECOND NUMBER IN H2 AND V2
GETBYT   (-6408) [$E6F8] \SE\       GETBYT S/R. EVALS EXPRESSION (FORMULA) POINTED TO BY TXTPTR ($00B8~$00B9) & CONVTS
                                    TO 1-BYT VAL IN X-REG & FACLO($00A1). A-REG GETS EXPRESSION TERMINAL SIGN (RESETS
                                    Y-REG=0)
~GETCMD~   (-7218) [$E3CE] \SE\     INTEGER BASIC ENTRY POINT TO GET A COMMAND FROM THE KEYBOARD
GETFMT   (-1879) [$F8A9]            MONITOR MEMORY LOCATION GETFMT
GETLN   (-662) [$FD6A] \SE\         PROMPT & GET LINE OF TEXT. ON CALLING A- X- Y-REGS NOT SIGNIFICANT. CV AND BASL~H
                                    SHOULD BE COMPATIBLE POINTING IN THE SCROLL WINDOW. CH INDICATES WHERE ON LINE THE
                                    PROMPT CHARACTER IS TO BE PLACED TO BE FOLLOWED BY ECHOED KEYBOARD INPUT; OUTPUT AS
                                    FOR GETLNZ (X-REG GETS #CHARS READ. DATA TO $200~$200~X (MAX $2FF) $200~X & Y-REG
                                    GET C/R (USES NXTCHAR)) (A- X- Y-REGS ALTERED)
GETLNZ   (-665) [$FD67] \SE\        OUTPUT A C/R (THROUGH COUT). GO TO GETLN TO WRITE PROMPT & GET A LINE OF DATA
                                    (USUALLY FROM KEYBOARD); ON SET-UP A- X- Y-REGS CH AND BASL~H NOT SIGNIFICANT. CV
                                    SHOULD POINT TO A LINE IN SCROLL WINDOW; ON OUTPUT KEYED IN INFO IS IN $200 THRU
                                    $200~X WHERE $200~X CONTAINS A CARRIAGE RETURN;A-REG CONTAINS CARRIAGE RETURN;X-REG
                                    CONTAINS NUMBER OF CHARACTERS READ EXCLUDING TERMINATING CARRIAGE RETURN;Y-REG
                                    CONTAINS CONTENTS OF WNDWDTH; CH CONTAINS ZERO;CV CONTAINS LINE POINTER (CURRENT
                                    VALUE);BASL~H CONTAINS MEMORY ADDRESS CORRESPONDING TO CV AND WNDLFT; SCREEN LINE
                                    IS BLANKS TO THE RIGHT OF THE END OF ECHOED INPUT (A- X- Y-REGS ALTERED)
~GETNEXT~   (-6027) [$E875] \SE\    INTEGER BASIC ENTRY TO 'GETNEXT' (FETCH NEXT STATEMENT FROM TEXT SOURCE)
GETNSP   (-2508) [$F634]            MINIASSEMBLER MEMORY LOCATION 'GETNSP'
GETNUM   (-6330) [$E746] \SE\       APPLESOFT FP - READ 2-BYTE NUM INTO LINNUM FROM TXTPTR. CHECK FOR COMMA. GET SINGLE
                                    BYTE NUMB IN X-REG.
GETNUM   (-89) [$FFA7]              MONITOR & MINIASSEMBLER MEMORY LOCATION 'GETNUM'
GETSPA   (-7086) [$E452] \SE\       APPLESOFT - GET SPACE FOR CHARACTER STRING. MOVES FRESPC & FRETOP DOWN. A-REG = #
                                    OF CHARS. POINTER TO SPC IN Y-REG(MSB) & X-REG(LSB)
~GETVAL255~   (-4352) [$EF00] \SE\  INTEGER BASIC ENTRY TO ROUTINE TO GET A ONE-BYTE VALUE
~GETVAL~   (-4556) [$EE34] \SE\     INTEGER BASIC ENTRY TO ROUTINE TO GET A VALUE WHICH WILL FIT INTO A SINGLE BYTE
                                    (VAL<=255)
~GETVERB~   (-6401) [$E6FF] \SE\    INTEGER BASIC ENTRY TO GET NEXT VERB TO USE
GIVAYF (INT=>FP)   (-7438) [$E2F2] \SE\APPLESOFT - FLOAT THE SIGNED INTEGER W/ LSB IN A-REG MSB IN Y-REG INTO FAC.
                                    RESETS VALTYP. (RESETS Y-REG=0)
GO   (-330) [$FEB6] \SE\            MONITOR MEMORY LOCATION 'GO'
GOCAL   (15809) [$3DC1] \SL\        DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - GO CALCULATE CORRECT TRACK
GOSEEK   (15992) [$3E78] \DL\       DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'GOSEEK'
GOSUBNDX   (252) [$00FC] \P1\       INTEGER BASIC MEMORY LOCATION 'GOSUBNDX' (GOSUB INDEX)
~GOSUB~   (-6084) [$E83C] \SE\      INTEGER BASIC ENTRY TO GOSUB HANDLER
-------------------------------------------------------------------------------------------------------------------------------
GBASCALC - GOSUB                    Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

GOTO (-9922) [$D93E] \SE\ APPLESOFT - USES LINGET & FNDLIN TO UPDATE TXTPTR. GOTO ASSUMES 6502 REGS HAVE BEEN SET UP BY CHRGET THAT FETCHED 1ST DIGIT

~GOTO~ (-6053) [$E85B] \SE\ INTEGER BASIC ENTRY TO 'GOTO' HANDLER

GTBYTC (-6411) [$E6F5] \SE\ APPLESOFT - JSR TO CHRGET TO GOBBLE A CHARACTER AND FALL INTO GETBYT

H2 (44) [$002C] \P1\ RIGHT END POINT OF A HORIZONTAL LINE BEING DRAWN BY HLINE: RANGE 0-39 ($0~$27)

HANDLERR (-3351) [$F2E9] \SE\ APPLESOFT ERROR PROC - SAVE CURLIN IN ERRLIN;TXTPTR IN ERRPOS;X-REG IN ERRNUM; REMSTK IN ERRSTK

HBASL~HBASH (38~39) [$0026~$0027] \P2\HI-RES GRAPHICS ON-THE-FLY BASE ADDRESS (LEFT END POINT OF DESIRED LINE FOR HI-RES PLOT)

HCLR (-12274) [$D00E] \SE\ HI-RES GRAPHICS CLEAR S/R CALL

HCLR (-3090) [$F3EE] \SE\ APPLESOFT HI-RES - CLEAR HI-RES SCREEN TO BLACK

HCOLOR (804) [$0324] \P1\ HI-RES GRAPHICS COLOR FOR HPLOT~ HPOSN

HCOLOR1 (28) [$001C] \P1\ HI-RES RUNNING COLOR MASK (ON-THE-FLY COLOR BYTE)

HEADR (-823) [$FCC9] MONITOR - WRITES SYNCHRONIZATION MONOTONE WHICH IS FIRST PART OF EVERY CASSETTE TAPE RECORD

~HEX/DEC~ (-6885) [$E51B] \SE\ INTEGER BASIC - DECIMAL LPRINT (LINE NUMBER PRINT) S/R; CONVERTS 2-BYTE (16-BIT) BINARY/HEX TO UNSIGNED DECIMAL (0-65535)

HFIND (-11780) [$D1FC] \SE\ HI-RES GRAPHICS FIND S/R CALL: PARAM=SHAPE~ROT~SCALE

HFIND (-2613) [$F5CB] \SE\ APPLESOFT HI-RES HFIND. CONVERT HI-RES CURSOR POSN TO X-Y COORDS. ON EXIT $00E0=HORIZ LSB;$00E1=HORIZ MSB;$00E2=VERT

HFNS (-2375) [$F6B9] \SE\ APPLESOFT - GET HI-RES PLOTTING COORDINATE FROM TXTPTR SETS UP 6502 REGISTERS FOR HPOSN: A-REG=VERT COORD;X-REG LSB OF HORIZ;Y-REG MSB OF HORIZ (A- X- Y-REGS ALTERED)

HGR (-3106) [$F3DE] \SE\ APPLESOFT HI-RES - INITIALIZE & CLEAR PAGE 1 HI-RES REGARDLESS OF SCREEN BEING DISPLAYED

HGR2 (-3116) [$F3D4] \SE\ APPLESOFT HI-RES - INITIALIZE & CLEAR PAGE 2 HI-RES REGARDLESS OF SCREEN BEING DISPLAYED

(HI-RES P1) (8192~16383) [$2000~$3FFF] \HB\HI-RES GRAPHICS PAGE 1

(HI-RES PAGE 2) (16384~24575) [$4000~$5FFF] \HB\HI-RES GRAPHICS PAGE 2

HI-RES (-16297) [$C057] \H1\ POKE TO 0 TO SET TO HI-RES GRAPHICS FROM LO-RES OR TEXT (SAME PAGE)

HIGHDS (148~149) [$0094~$0095] \P2\ USED BY BLOCK TRANSFER UTILITY (BLTU) AS HIGH DESTINATION

HIGHTR (150~151) [$0096~$0097] \P2\ APPLESOFT - USED BY BLOCK TRANSFER UTILITY (BLTU) AS HIGH END OF BLOCK TO BE TRANSFERRED

HIMEML~HIMEMH (76~77) [$004C~$004D] \P2\ADDRESS POINTER TO HIMEM (INTEGER BASIC - END OF BASIC PROGRAM)(APPLESOFT - START OF STRING DATA)

~HIMEM~ (-4019) [$F04D] \SE\ INTEGER BASIC ENTRY TO THE HIMEM FUNCTION

(HIRES P1L000) (8192~8231) [$2000~$2027] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #000

(HIRES P1L001) (9216~9255) [$2400~$2427] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #001

(HIRES P1L002) (10240~10279) [$2800~$2827] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #002

(HIRES P1L003) (11264~11303) [$2C00~$2C27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #003

(HIRES P1L004) (12288~12327) [$3000~$3027] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #004

(HIRES P1L005) (13312~13351) [$3400~$3427] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #005

(HIRES P1L006) (14336~14375) [$3800~$3827] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #006

(HIRES P1L007) (15360~15399) [$3C00~$3C27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #007

(HIRES P1L008) (8320~8359) [$2080~$20A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #008

(HIRES P1L009) (9344~9383) [$2480~$24A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #009

(HIRES P1L010) (10368~10407) [$2880~$28A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #010

(HIRES P1L011) (11392~11431) [$2C80~$2CA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #011

(HIRES P1L012) (12416~12455) [$3080~$30A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #012

(HIRES P1L013) (13440~13479) [$3480~$34A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #013

(HIRES P1L014) (14464~14503) [$3880~$38A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #014

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

(HIRES P1L015)   (15488~15527) [$3C80~$3CA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #015
(HIRES P1L016)   (8448~8487) [$2100~$2127] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #016
(HIRES P1L017)   (9472~9511) [$2500~$2527] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #017
(HIRES P1L018)   (10496~10535) [$2900~$2927] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #018
(HIRES P1L019)   (11520~11559) [$2D00~$2D27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #019
(HIRES P1L020)   (12544~12583) [$3100~$3127] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #020
(HIRES P1L021)   (13568~13607) [$3500~$3527] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #021
(HIRES P1L022   (14592~14631) [$3900~$3927] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #022
(HIRES P1L023)   (15616~15655) [$3D00~$3D27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #023
(HIRES P1L024)   (8576~8615) [$2180~$21A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #024
(HIRES P1L025)   (9600~9639) [$2580~$25A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #025
(HIRES P1L026)   (10624~10663) [$2980~$29A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #026
(HIRES P1L027)   (11648~11687) [$2D80~$2DA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #027
(HIRES P1L028)   (12672~12711) [$3180~$31A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #028
(HIRES P1L029)   (13696~13735) [$3580~$35A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #029
(HIRES P1L030)   (14720~14759) [$3980~$39A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #030
(HIRES P1L031)   (15744~15783) [$3D80~$3DA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #031
(HIRES P1L032)   (8704~8743) [$2200~$2227] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #032
(HIRES P1L033)   (9728~9767) [$2600~$2627] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #033
(HIRES P1L034)   (10752~10791) [$2A00~$2A27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #034
(HIRES P1L035)   (11776~11815) [$2E00~$2E27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #035
(HIRES P1L036)   (12800~12839) [$3200~$3227] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #036
(HIRES P1L037)   (13824~13863) [$3600~$3627] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #037
(HIRES P1L038)   (14848~14887) [$3A00~$3A27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #038
(HIRES P1L039)   (15872~15911) [$3E00~$3E27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #039
(HIRES P1L040)   (8832~8871) [$2280~$22A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #040
(HIRES P1L041)   (9856~9895) [$2680~$26A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #041
(HIRES P1L042)   (10880~10919) [$2A80~$2AA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #042
(HIRES P1L043)   (11904~11943) [$2E80~$2EA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #043
(HIRES P1L044)   (12928~12967) [$3280~$32A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #044
(HIRES P1L045)   (13056~13095) [$3300~$3327] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #045
(HIRES P1L045)   (13952~13991) [$3680~$36A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #045
(HIRES P1L046)   (14976~15015) [$3A80~$3AA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #046
(HIRES P1L047)   (16000~16039) [$3E80~$3EA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #047
(HIRES P1L048)   (8960~8999) [$2300~$2327] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #048
(HIRES P1L049)   (9984~10023) [$2700~$2727] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #049
(HIRES P1L050)   (11008~11047) [$2B00~$2B27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #050
(HIRES P1L051)   (12032~12071) [$2F00~$2F27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #051
(HIRES P1L053)   (14080~14119) [$3700~$3727] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #053
(HIRES P1L054)   (15104~15143) [$3B00~$3B27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #054
(HIRES P1L055)   (16128~16167) [$3F00~$3F27] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #055
(HIRES P1L056)   (9088~9127) [$2380~$23A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #056
(HIRES P1L057)   (10112~10151) [$2780~$27A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #057
(HIRES P1L058)   (11136~11175) [$2B80~$2BA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #058
(HIRES P1L059)   (12160~12199) [$2F80~$2FA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #059
(HIRES P1L060)   (13184~13223) [$3380~$33A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #060
(HIRES P1L061)   (14208~14247) [$3780~$37A7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #061
(HIRES P1L062)   (15232~15271) [$3B80~$3BA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #062
(HIRES P1L063)   (16256~16295) [$3F80~$3FA7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #063
(HIRES P1L064)   (8232~8271) [$2028~$204F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #064
(HIRES P1L065)   (9256~9295) [$2428~$244F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #065
-------------------------------------------------------------------------------------------------------------------------------
(HIRES P1L015) - (HIRES P1L065)      Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------

(HIRES P1L066)   (10280~10319) [$2828~$284F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #066
(HIRES P1L067)   (11304~11343) [$2C28~$2C4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #067
(HIRES P1L068)   (12328~12367) [$3028~$304F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #068
(HIRES P1L069)   (13352~13391) [$3428~$344F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #069
(HIRES P1L070)   (14376~14415) [$3828~$384F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #070
(HIRES P1L071)   (15400~15439) [$3C28~$3C4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #071
(HIRES P1L072)   (8360~8399) [$20A8~$20CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #072
(HIRES P1L073)   (9384~9423) [$24A8~$24CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #073
(HIRES P1L074)   (10408~10447) [$28A8~$28CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #074
(HIRES P1L075)   (11432~11471) [$2CA8~$2CCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #075
(HIRES P1L076)   (12456~12495) [$30A8~$30CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #076
(HIRES P1L077)   (13480~13519) [$34A8~$34CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #077
(HIRES P1L078)   (14504~14543) [$38A8~$38CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #078
(HIRES P1L079)   (15528~15567) [$3CA8~$3CCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #079
(HIRES P1L081)   (9512~9551) [$2528~$254F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #081
(HIRES P1L082)   (10536~10575) [$2928~$294F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #082
(HIRES P1L083)   (11560~11599) [$2D28~$2D4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #083
(HIRES P1L084)   (12584~12623) [$3128~$314F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #084
(HIRES P1L085)   (13608~13647) [$3528~$354F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #085
(HIRES P1L086)   (14632~14671) [$3928~$394F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #086
(HIRES P1L087)   (15656~15695) [$3D28~$3D4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #087
(HIRES P1L088)   (8616~8655) [$21A8~$21CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #088
(HIRES P1L089)   (9640~9679) [$25A8~$25CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #089
(HIRES P1L090)   (10664~10703) [$29A8~$29CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #090
(HIRES P1L091)   (11688~11727) [$2DA8~$2DCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #091
(HIRES P1L092)   (12712~12751) [$31A8~$31CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #092
(HIRES P1L093)   (13736~13775) [$35A8~$35CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #093
(HIRES P1L094)   (14760~14799) [$39A8~$39CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #094
(HIRES P1L095)   (15784~15823) [$3DA8~$3DCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #095
(HIRES P1L096)   (8744~8783) [$2228~$224F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #096
(HIRES P1L097)   (9768~9807) [$2628~$264F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #097
(HIRES P1L098)   (10792~10831) [$2A28~$2A4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #098
(HIRES P1L099)   (11816~11855) [$2E28~$2E4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #099
(HIRES P1L100)   (12840~12879) [$3228~$324F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #100
(HIRES P1L101)   (13864~13903) [$3628~$364F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #101
(HIRES P1L102)   (14888~14927) [$3A28~$3A4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #102
(HIRES P1L103)   (15912~15951) [$3E28~$3E4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #103
(HIRES P1L104)   (8872~8911) [$22A8~$22CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #104
(HIRES P1L105)   (9896~9935) [$26A8~$26CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #105
(HIRES P1L106)   (10920~10959) [$2AA8~$2ACF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #106
(HIRES P1L107)   (11944~11983) [$2EA8~$2ECF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #107
(HIRES P1L108)   (12968~13007) [$32A8~$32CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #108
(HIRES P1L109)   (13992~14031) [$36A8~$36CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #109
(HIRES P1L110)   (15016~15055) [$3AA8~$3ACF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #110
(HIRES P1L111)   (16040~16079) [$3EA8~$3ECF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #111
(HIRES P1L112)   (9000~9039) [$2328~$234F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #112
(HIRES P1L113)   (10024~10063) [$2728~$274F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #113
(HIRES P1L114)   (11048~13871) [$2B28~$362F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #114
(HIRES P1L115)   (12072~12111) [$2F28~$2F4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #115
(HIRES P1L116)   (13096~13135) [$3328~$334F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #116
(HIRES P1L117)   (14120~14159) [$3728~$374F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #117

-------------------------------------------------------------------------------------------------------------------
(HIRES P1L066) - (HIRES P1L117)      Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------

(HIRES P1L118)    (15144~13871) [$3B28~$362F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #118
(HIRES P1L119)    (16168~16207) [$3F28~$3F4F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #119
(HIRES P1L120)    (9128~9167) [$23A8~$23CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #120
(HIRES P1L121)    (10152~10191) [$27A8~$27CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #121
(HIRES P1L122)    (11176~11215) [$2BA8~$2BCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #122
(HIRES P1L123)    (12200~12239) [$2FA8~$2FCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #123
(HIRES P1L124)    (13224~13263) [$33A8~$33CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #124
(HIRES P1L125)    (14248~14287) [$37A8~$37CF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #125
(HIRES P1L126)    (15272~15311) [$3BA8~$3BCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #126
(HIRES P1L127)    (16296~16335) [$3FA8~$3FCF] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #127
(HIRES P1L128)    (8272~8311) [$2050~$2077] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #128
(HIRES P1L129)    (9296~9335) [$2450~$2477] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #129
(HIRES P1L130)    (10320~10359) [$2850~$2877] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #130
(HIRES P1L131)    (11344~11383) [$2C50~$2C77] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #131
(HIRES P1L132)    (12368~12407) [$3050~$3077] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #132
(HIRES P1L133)    (13392~13431) [$3450~$3477] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #133
(HIRES P1L134)    (14416~14455) [$3850~$3877] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #134
(HIRES P1L135)    (15440~15479) [$3C50~$3C77] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #135
(HIRES P1L136)    (8400~8423) [$20D0~$20E7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #136
(HIRES P1L137)    (9424~9447) [$24D0~$24E7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #137
(HIRES P1L138)    (10448~10471) [$28D0~$28E7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #138
(HIRES P1L139)    (11472~11495) [$2CD0~$2CE7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #139
(HIRES P1L140)    (12496~12519) [$30D0~$30E7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #140
(HIRES P1L141)    (13520~13543) [$34D0~$34E7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #141
(HIRES P1L142)    (14544~14567) [$38D0~$38E7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #142
(HIRES P1L143)    (15568~15591) [$3CD0~$3CE7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #143
(HIRES P1L144)    (8528~8575) [$2150~$217F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #144
(HIRES P1L145)    (9552~9599) [$2550~$257F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #145
(HIRES P1L146)    (10576~10623) [$2950~$297F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #146
(HIRES P1L147)    (11600~11647) [$2D50~$2D7F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #147
(HIRES P1L148)    (12624~12671) [$3150~$317F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #148
(HIRES P1L149)    (13648~13695) [$3550~$357F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #149
(HIRES P1L150)    (14672~14719) [$3950~$397F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #150
(HIRES P1L151)    (15696~15743) [$3D50~$3D7F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #151
(HIRES P1L152)    (8656~8695) [$21D0~$21F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #152
(HIRES P1L153)    (9680~9719) [$25D0~$25F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #153
(HIRES P1L154)    (10704~10743) [$29D0~$29F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #154
(HIRES P1L155)    (11728~11767) [$2DD0~$2DF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #155
(HIRES P1L156)    (12752~12791) [$31D0~$31F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #156
(HIRES P1L157)    (13776~13815) [$35D0~$35F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #157
(HIRES P1L158)    (14800~14839) [$39D0~$39F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #158
(HIRES P1L159)    (15824~15863) [$3DD0~$3DF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #159
(HIRES P1L160)    (8784~8823) [$2250~$2277] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #160
(HIRES P1L161)    (9808~9847) [$2650~$2677] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #161
(HIRES P1L162)    (10832~10871) [$2A50~$2A77] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #162
(HIRES P1L163)    (11856~11895) [$2E50~$2E77] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #163
(HIRES P1L164)    (12880~12919) [$3250~$3277] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #164
(HIRES P1L165)    (13904~13943) [$3650~$3677] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #165
(HIRES P1L166)    (14928~14967) [$3A50~$3A77] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #166
(HIRES P1L167)    (15952~15991) [$3E50~$3E77] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #167
(HIRES P1L168)    (8912~8951) [$22D0~$22F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #168

-------------------------------------------------------------------------------
(HIRES P1L118) - (HIRES P1L168)      Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------

(HIRES P1L169)    (9936~9975) [$26D0~$26F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #169
(HIRES P1L170)    (10960~10999) [$2AD0~$2AF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #170
(HIRES P1L171)    (11984~12023) [$2ED0~$2EF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #171
(HIRES P1L172)    (13008~13047) [$32D0~$32F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #172
(HIRES P1L173)    (14032~14071) [$36D0~$36F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #173
(HIRES P1L174)    (15056~15095) [$3AD0~$3AF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #174
(HIRES P1L175)    (16080~16119) [$3ED0~$3EF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #175
(HIRES P1L176)    (9040~9087) [$2350~$237F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #176
(HIRES P1L177)    (10064~10111) [$2750~$277F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #177
(HIRES P1L178)    (11088~11135) [$2B50~$2B7F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #178
(HIRES P1L179)    (12112~12159) [$2F50~$2F7F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #179
(HIRES P1L180)    (13136~13183) [$3350~$337F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #180
(HIRES P1L181)    (14160~14207) [$3750~$377F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #181
(HIRES P1L182)    (15184~15231) [$3B50~$3B7F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #182
(HIRES P1L183)    (16208~16255) [$3F50~$3F7F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #183
(HIRES P1L184)    (9168~9207) [$23D0~$23F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #184
(HIRES P1L185)    (10192~18423) [$27D0~$47F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #185
(HIRES P1L186)    (11216~11255) [$2BD0~$2BF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #186
(HIRES P1L187)    (12240~12279) [$2FD0~$2FF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #187
(HIRES P1L188)    (13264~13303) [$33D0~$33F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #188
(HIRES P1L189)    (14288~14327) [$37D0~$37F7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #189
(HIRES P1L190)    (15312~15351) [$3BD0~$3BF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #190
(HIRES P1L191)    (16336~16375) [$3FD0~$3FF7] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #191
(HIRES P1L80)    (8488~8527) [$2128~$214F] \HB\HI-RES GRAPHICS: PAGE 1 - LINE #80
(HIRES P2L000)    (16384~16423) [$4000~$4027] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #000
(HIRES P2L001)    (17408~17447) [$4400~$4427] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #001
(HIRES P2L002)    (18432~18471) [$4800~$4827] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #002
(HIRES P2L003)    (19456~19495) [$4C00~$4C27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #003
(HIRES P2L004)    (20480~20519) [$5000~$5027] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #004
(HIRES P2L005)    (21504~21543) [$5400~$5427] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #005
(HIRES P2L006)    (22528~22567) [$5800~$5827] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #006
(HIRES P2L007)    (23552~23591) [$5C00~$5C27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #007
(HIRES P2L008)    (16512~16551) [$4080~$40A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #008
(HIRES P2L009)    (17536~17575) [$4480~$44A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #009
(HIRES P2L010)    (18560~18599) [$4880~$48A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #010
(HIRES P2L011)    (19584~19623) [$4C80~$4CA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #011
(HIRES P2L012)    (20608~20647) [$5080~$50A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #012
(HIRES P2L013)    (21632~21671) [$5480~$54A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #013
(HIRES P2L014)    (22656~22695) [$5880~$58A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #014
(HIRES P2L015)    (23680~23719) [$5C80~$5CA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #015
(HIRES P2L016)    (16640~16679) [$4100~$4127] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #016
(HIRES P2L017)    (17664~17703) [$4500~$4527] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #017
(HIRES P2L018)    (18688~18727) [$4900~$4927] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #018
(HIRES P2L019)    (19712~19751) [$4D00~$4D27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #019
(HIRES P2L020)    (20736~20775) [$5100~$5127] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #020
(HIRES P2L021)    (21760~21799) [$5500~$5527] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #021
(HIRES P2L022)    (22784~22823) [$5900~$5927] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #022
(HIRES P2L023)    (23808~23847) [$5D00~$5D27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #023
(HIRES P2L024)    (16768~16807) [$4180~$41A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #024
(HIRES P2L025)    (17792~17831) [$4580~$45A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #025
(HIRES P2L026)    (18816~18855) [$4980~$49A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #026

-------------------------------------------------------------------------------------------------------------------
(HIRES P1L169) - (HIRES P2L026)      Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

(HIRES P2L027) (19840~19879) [$4D80~$4DA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #027
(HIRES P2L028) (20864~20903) [$5180~$51A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #028
(HIRES P2L029) (21888~21927) [$5580~$55A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #029
(HIRES P2L030) (22912~22951) [$5980~$59A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #030
(HIRES P2L031) (23936~23975) [$5D80~$5DA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #031
(HIRES P2L032) (16896~16935) [$4200~$4227] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #032
(HIRES P2L033) (17920~17959) [$4600~$4627] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #033
(HIRES P2L034) (18944~18983) [$4A00~$4A27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #034
(HIRES P2L035) (19968~20007) [$4E00~$4E27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #035
(HIRES P2L036) (20992~21031) [$5200~$5227] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #036
(HIRES P2L037) (22016~22055) [$5600~$5627] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #037
(HIRES P2L038) (23040~23079) [$5A00~$5A27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #038
(HIRES P2L039) (24064~24103) [$5E00~$5E27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #039
(HIRES P2L040) (17024~17063) [$4280~$42A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #040
(HIRES P2L041) (18048~18087) [$4680~$46A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #041
(HIRES P2L042) (19072~19111) [$4A80~$4AA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #042
(HIRES P2L043) (20096~20135) [$4E80~$4EA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #043
(HIRES P2L044) (21120~21159) [$5280~$52A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #044
(HIRES P2L045) (21248~21287) [$5300~$5327] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #045
(HIRES P2L045) (22144~22183) [$5680~$56A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #045
(HIRES P2L046) (23168~23207) [$5A80~$5AA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #046
(HIRES P2L047) (24192~24231) [$5E80~$5EA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #047
(HIRES P2L048) (17152~17191) [$4300~$4327] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #048
(HIRES P2L049) (18176~18215) [$4700~$4727] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #049
(HIRES P2L050) (19200~19239) [$4B00~$4B27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #050
(HIRES P2L051) (20224~20263) [$4F00~$4F27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #051
(HIRES P2L053) (22272~22311) [$5700~$5727] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #053
(HIRES P2L054) (23296~23335) [$5B00~$5B27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #054
(HIRES P2L055) (24320~24359) [$5F00~$5F27] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #055
(HIRES P2L056) (17280~17319) [$4380~$43A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #056
(HIRES P2L057) (18304~18343) [$4780~$47A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #057
(HIRES P2L058) (19328~19367) [$4B80~$4BA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #058
(HIRES P2L059) (20352~20391) [$4F80~$4FA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #059
(HIRES P2L060) (21376~21415) [$5380~$53A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #060
(HIRES P2L061) (22400~22439) [$5780~$57A7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #061
(HIRES P2L062) (23424~23463) [$5B80~$5BA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #062
(HIRES P2L063) (24448~24487) [$5F80~$5FA7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #063
(HIRES P2L064) (16424~16463) [$4028~$404F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #064
(HIRES P2L065) (17448~17487) [$4428~$444F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #065
(HIRES P2L066) (18472~18511) [$4828~$484F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #066
(HIRES P2L067) (19496~19535) [$4C28~$4C4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #067
(HIRES P2L068) (20520~20559) [$5028~$504F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #068
(HIRES P2L069) (21544~21583) [$5428~$544F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #069
(HIRES P2L070) (22568~22607) [$5828~$584F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #070
(HIRES P2L071) (23592~23631) [$5C28~$5C4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #071
(HIRES P2L072) (16552~16591) [$40A8~$40CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #072
(HIRES P2L073) (17576~17615) [$44A8~$44CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #073
(HIRES P2L074) (18600~18639) [$48A8~$48CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #074
(HIRES P2L075) (19624~19663) [$4CA8~$4CCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #075
(HIRES P2L076) (20648~20687) [$50A8~$50CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #076
(HIRES P2L077) (21672~21711) [$54A8~$54CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #077

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION

(HIRES P2L078) (22696~22735) [$58A8~$58CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #078
(HIRES P2L079) (23720~23759) [$5CA8~$5CCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #079
(HIRES P2L080) (16680~16719) [$4128~$414F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #080
(HIRES P2L081) (17704~17743) [$4528~$454F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #081
(HIRES P2L082) (18728~18767) [$4928~$494F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #082
(HIRES P2L083) (19752~19791) [$4D28~$4D4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #083
(HIRES P2L084) (20776~20815) [$5128~$514F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #084
(HIRES P2L085) (21800~21839) [$5528~$554F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #085
(HIRES P2L086) (22824~22863) [$5928~$594F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #086
(HIRES P2L087) (23848~23887) [$5D28~$5D4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #087
(HIRES P2L088) (16808~16847) [$41A8~$41CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #088
(HIRES P2L089) (17832~17871) [$45A8~$45CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #089
(HIRES P2L090) (18856~18895) [$49A8~$49CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #090
(HIRES P2L091) (19880~19919) [$4DA8~$4DCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #091
(HIRES P2L092) (20904~20943) [$51A8~$51CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #092
(HIRES P2L093) (21928~21967) [$55A8~$55CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #093
(HIRES P2L094) (22952~22991) [$59A8~$59CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #094
(HIRES P2L095) (23976~24015) [$5DA8~$5DCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #095
(HIRES P2L096) (16936~16975) [$4228~$424F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #096
(HIRES P2L098) (18984~19023) [$4A28~$4A4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #098
(HIRES P2L099) (20008~20047) [$4E28~$4E4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #099
(HIRES P2L100) (21032~21071) [$5228~$524F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #100
(HIRES P2L101) (22056~22095) [$5628~$564F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #101
(HIRES P2L102) (23080~23119) [$5A28~$5A4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #102
(HIRES P2L103) (24104~24143) [$5E28~$5E4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #103
(HIRES P2L104) (17064~17103) [$42A8~$42CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #104
(HIRES P2L105) (18088~18127) [$46A8~$46CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #105
(HIRES P2L106) (19112~19151) [$4AA8~$4ACF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #106
(HIRES P2L107) (20136~20175) [$4EA8~$4ECF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #107
(HIRES P2L108) (21160~21199) [$52A8~$52CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #108
(HIRES P2L109) (22184~22223) [$56A8~$56CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #109
(HIRES P2L110) (23208~23247) [$5AA8~$5ACF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #110
(HIRES P2L111) (24232~24271) [$5EA8~$5ECF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #111
(HIRES P2L112) (17192~17231) [$4328~$434F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #112
(HIRES P2L113) (18216~18255) [$4728~$474F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #113
(HIRES P2L114) (19240~22063) [$4B28~$562F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #114
(HIRES P2L115) (20264~20303) [$4F28~$4F4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #115
(HIRES P2L116) (21288~21327) [$5328~$534F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #116
(HIRES P2L117) (22312~22351) [$5728~$574F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #117
(HIRES P2L118) (23336~22063) [$5B28~$562F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #118
(HIRES P2L119) (24360~24399) [$5F28~$5F4F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #119
(HIRES P2L120) (17320~17359) [$43A8~$43CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #120
(HIRES P2L121) (18344~18383) [$47A8~$47CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #121
(HIRES P2L122) (19368~19407) [$4BA8~$4BCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #122
(HIRES P2L123) (20392~20431) [$4FA8~$4FCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #123
(HIRES P2L124) (21416~21455) [$53A8~$53CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #124
(HIRES P2L125) (22440~22479) [$57A8~$57CF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #125
(HIRES P2L126) (23464~23503) [$5BA8~$5BCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #126
(HIRES P2L127) (24488~24527) [$5FA8~$5FCF] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #127
(HIRES P2L128) (16464~16503) [$4050~$4077] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #128
(HIRES P2L129) (17488~17527) [$4450~$4477] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #129

NAME (DEC LOCN)	[HEX LOCN] \USE-TYPE\ - DESCRIPTION
(HIRES P2L130)	(18512-18551) [$4850-$4877] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #130
(HIRES P2L131)	(19536-19575) [$4C50-$4C77] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #131
(HIRES P2L132)	(20560-20599) [$5050-$5077] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #132
(HIRES P2L133)	(21584-21623) [$5450-$5477] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #133
(HIRES P2L134)	(22608-22647) [$5850-$5877] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #134
(HIRES P2L135)	(23632-23671) [$5C50-$5C77] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #135
(HIRES P2L136)	(16592-16615) [$40D0-$40E7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #136
(HIRES P2L137)	(17616-17639) [$44D0-$44E7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #137
(HIRES P2L138)	(18640-18663) [$48D0-$48E7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #138
(HIRES P2L139)	(19664-19687) [$4CD0-$4CE7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #139
(HIRES P2L140)	(20688-20711) [$50D0-$50E7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #140
(HIRES P2L141)	(21712-21735) [$54D0-$54E7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #141
(HIRES P2L142)	(22736-22759) [$58D0-$58E7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #142
(HIRES P2L143)	(23760-23783) [$5CD0-$5CE7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #143
(HIRES P2L144)	(16720-16767) [$4150-$417F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #144
(HIRES P2L145)	(17744-17791) [$4550-$457F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #145
(HIRES P2L146)	(18768-18815) [$4950-$497F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #146
(HIRES P2L147)	(19792-19839) [$4D50-$4D7F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #147
(HIRES P2L148)	(20816-20863) [$5150-$517F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #148
(HIRES P2L149)	(21840-21887) [$5550-$557F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #149
(HIRES P2L150)	(22864-22911) [$5950-$597F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #150
(HIRES P2L151)	(23888-23935) [$5D50-$5D7F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #151
(HIRES P2L152)	(16848-16887) [$41D0-$41F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #152
(HIRES P2L153)	(17872-17911) [$45D0-$45F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #153
(HIRES P2L154)	(18896-18935) [$49D0-$49F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #154
(HIRES P2L155)	(19920-19959) [$4DD0-$4DF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #155
(HIRES P2L156)	(20944-20983) [$51D0-$51F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #156
(HIRES P2L157)	(21968-22007) [$55D0-$55F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #157
(HIRES P2L158)	(22992-23031) [$59D0-$59F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #158
(HIRES P2L159)	(24016-24055) [$5DD0-$5DF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #159
(HIRES P2L160)	(16976-17015) [$4250-$4277] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #160
(HIRES P2L161)	(18000-18039) [$4650-$4677] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #161
(HIRES P2L162)	(19024-19063) [$4A50-$4A77] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #162
(HIRES P2L163)	(20048-20087) [$4E50-$4E77] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #163
(HIRES P2L164)	(21072-21111) [$5250-$5277] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #164
(HIRES P2L165)	(22096-22135) [$5650-$5677] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #165
(HIRES P2L166)	(23120-23159) [$5A50-$5A77] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #166
(HIRES P2L167)	(24144-24183) [$5E50-$5E77] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #167
(HIRES P2L168)	(17104-17143) [$42D0-$42F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #168
(HIRES P2L169)	(18128-18167) [$46D0-$46F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #169
(HIRES P2L170)	(19152-19191) [$4AD0-$4AF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #170
(HIRES P2L171)	(20176-20215) [$4ED0-$4EF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #171
(HIRES P2L172)	(21200-21239) [$52D0-$52F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #172
(HIRES P2L173)	(22224-22263) [$56D0-$56F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #173
(HIRES P2L174)	(23248-23287) [$5AD0-$5AF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #174
(HIRES P2L175)	(24272-24311) [$5ED0-$5EF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #175
(HIRES P2L176)	(17232-17279) [$4350-$437F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #176
(HIRES P2L177)	(18256-18303) [$4750-$477F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #177
(HIRES P2L178)	(19280-19327) [$4B50-$4B7F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #178
(HIRES P2L179)	(20304-20351) [$4F50-$4F7F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #179
(HIRES P2L180)	(21328-21375) [$5350-$537F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #180

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

(HIRES P2L181)    (22352~22399) [$5750~$577F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #181
(HIRES P2L182)    (23376~23423) [$5B50~$5B7F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #182
(HIRES P2L183)    (24400~24447) [$5F50~$5F7F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #183
(HIRES P2L184)    (17360~17399) [$43D0~$43F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #184
(HIRES P2L185)    (18384~18423) [$47D0~$47F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #185
(HIRES P2L186)    (19408~19447) [$4BD0~$4BF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #186
(HIRES P2L187)    (20432~20471) [$4FD0~$4FF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #187
(HIRES P2L188)    (21456~21495) [$53D0~$53F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #188
(HIRES P2L189)    (22480~22519) [$57D0~$57F7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #189
(HIRES P2L190)    (23504~23543) [$5BD0~$5BF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #190
(HIRES P2L191)    (24528~24567) [$5FD0~$5FF7] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #191
(HIRES P2L97)    (17960~17999) [$4628~$464F] \HB\HI-RES GRAPHICS: PAGE 2 - LINE #97
HISCR    (-16299) [$C055] \H1\              POKE TO 0 TO DISPLAY PAGE 2  (DOES NOT CLEAR SCREEN)
HLIN    (-2768) [$F530] \SE\                APPLESOFT HI-RES HORIZ LINE DRAWING FROM LAST POINT PLOTTED TO X-COORD =
                                            X-REG(MSB)&A-REG(LSB);Y-COORD=Y-REG
HLINE    (-2023) [$F819] \SE\               LO-RES S/R TO DRAW HORIZONTAL LINE AT Y-COORD = (A-REG) WITH X-COORDS FROM
                                            (A-REG) THRU (H2)($002C) (A- Y-REGS ALTERED)
HLINE1    (-2020) [$F81C] \SE\              LO-RES S/R. DRAW HORZ LINE AT Y-COORD ESTAB BY GBASL~H & MASK. X-CORDS FROM
                                            (Y-REG) THRU ($002C) (A- Y-REGS ALTERED)
~HLIN~    (-4432) [$EEB0] \SE\              INTEGER BASIC ENTRY POINT TO DRAW A LO-RES HORIZONTAL LINE
HMASK    (48) [$0030] \P1\                  HI-RES GRAPHICS ON-THE-FLY BIT MASK
HNDLERR    (15913) [$3E29] \SL\             DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF ERROR
                                            HANDLING MODULE
HNDLERR    (-16824~-16816) [$BE48~$BE50]    DOS 3.3 - SET CARRY; STORE A-REG IN IOB AS RETURN CODE. TURN OFF MOTOR. RETURN
                                            TO CALLER
HNDX    (805) [$0325] \P1\                 HI-RES  ON-THE-FLY BYTE INDEX FROM BASE ADDRESS TO CURRENT PLOT BYTE (FUNCTION
                                            OF CURRENT X-COORD)
HOME    (-936) [$FC58] \SE\                CLEAR SCROLL WINDOW TO BLANKS. SET CURSOR TO TOP LEFT CORNER (A- Y-REGS
                                            ALTERED)
HPAG    (230) [$00E6] \P1\                 HI-RES PAGE TO PLOT ON REGARDLESS OF WHICH PAGE BEING DISPLAYED - $20 FOR PG1;
                                            $40 FOR PG2
HPAG    (806) [$0326] \P1\                 HI-ORDER BYTE OF START ADDR OF CURRENT HI-RES DISPLAY MEM PG (POKE 32 FOR
                                            HI-RES PG1 ~ 64 FOR PG2)
HPAG    (806) [$0326] \P1\                 HI-RES GRAPHICS MEM PAGE FOR PLOTTING GRAPHICS $20 FOR PG1 ~$40 FOR PG2
HPLOT    (-2989) [$F453] \SE\              APPLESOFT HI-RES - CALL HPOSN THEN PLOT DOT THERE. NO DOT MAY BE PLOTTED IS
                                            PLOTTING NON-WHITE AT COMPLEMENTARY COLOR X COORD
HPOSN    (-3059) [$F40D] \SE\               APPLESOFT HI-RES - POSN HI-RES CURSOR W/O PLOTTING. HPAG DETERMINES WHICH
                                            PAGE; HORIZ = Y-REG(MSB)&X-REG(LSB);VERT= A-REG
(I/O HOOK TBLS)    (54~57) [$0036~$0039] \PB\MONITOR OUTPUT & INPUT HOOKS (VECTORS TO DOS OUTPUT & INPUT ROUTINES)
IEVEN    (-1893) [$F89B]                   MONITOR MEMORY LOCATION 'IEVEN'
~IF/THEN~    (-6104) [$E828] \SE\          INTEGER BASIC ENTRY TO IF/THEN ROUTINE
IFSKIP    (212) [$00D4] \P1\                INTEGER BASIC MEMORY LOCATION 'IFSKIP' (IF\THEN FAIL FLAG)
ILEAV    (-16456~-16441) [$BFB8~$BFC7]      DOS 3.3 - SECTOR TRANSLATE TABLE. SECTOR INTERLEAVING DONE WITH SOFTWARE
(ILLDIRPRT)    (-7413) [$E30B] \SE\         PRINT "ILLEGAL DIRECT" THEN HALT AT APPLESOFT (]) LEVEL
(ILLEGAL QTY PRT)    (-7783) [$E199] \SE\   APPLESOFT - PRINT "ILLEGAL QUANTITY" AND HALT AT APPLESOFT LEVEL (])
~IN#S~    (-3046) [$F41A] \SE\              INTEGER BASIC ENTRY TO ROUTINE TO SET INPUT PORT
IN    (512) [$0200]                       MONITOR & MINIASSEMBLER MEMORY LOCATION 'IN'
INCHR    (-10925) [$D553] \SE\             APPLESOFT - GET ONE CHAR FROM CURRENT INPUT DEVICE IN A-REG & MASK OF MSB.
                                            USES MAIN APPLE INPUT ROUTINES & SUPPORTS HANDSHAKING
INDEX    (94~95) [$005E~$005F] \P2\        APPLESOFT TEMPORARY (STACK) POINTER FOR MOVING STRINGS
INIT    (-1233) [$FB2F] \SE\               MONITOR S/R- SCREEN INITIALIZATION (RESET TEXT MODE)
-------------------------------------------------------------------------------------------------------------------------------
(HIRES P2L181) - INIT                Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

INITAN    (-1425) [$FA6F]                  AUTOSTART MONITOR MEMORY LOCATION 'INITAN'
INITBL    (-1263) [$FB11]                  MONITOR MEMORY LOCATION 'INITBL'
(INITFACMANT)    (-5056) [$EC40] \SE\      APPLESOFT FP - INITIALIZED MANTISSA OF FAC (EXCEPT EXTENSION BYTE) TO VALUE IN
                                          A-REGISTER
INLIN    (-10964) [$D52C] \SE\            APPLESOFT - INPUT LINE OF TEXT FROM CURRENT INPUT DEVICE INTO INPUT BUFFER
                                          (BUF) & FALL INTO GDBUFS. NO PROMPT!
INLIN+2    (-10962) [$D52E] \SE\          APPLESOFT - INPUT LINE OF TEXT FROM CURRENT INPUT DEVICE INTO INPUT BUFFER
                                          (BUF) & FALL INTO GDBUFS. CHAR IN X-REG USED AS PROMPT
(INP SOURCE PTR)    (127~128) [$007F~$0080] \P2\APPLESOFT - PTR TO CURRENT SOURCE OF INPUT. $201 DURING INPUT STATEMENT
                                          IF STANDARD BUFFER IN USE
INPORT    (-373) [$FE8B]                  MONITOR MEMORY LOCATION  'INPORT'
INPRT    (-4839) [$ED19] \SE\             APPLESOFT - PRINT 'IN' & CURRENT LINE # FROM CURLIN. USES LPRINT
INPRT    (-371) [$FE8D]                   MONITOR MEMORY LOCATION 'INPRT'
~INPUTSTR~    (-7823) [$E171] \SE\        INTEGER BASIC ENTRY POINT TO 'INPUT A STRING' ROUTINE
~INPUT~    (-5206) [$EBAA] \SE\           INTEGER BASIC ENTRY TO INPUT ROUTINE
INSDS1    (-1918) [$F882]                 MONITOR MEMORY LOCATION 'INSDS1'
INSDS2    (-1906) [$F88E]                 MONITOR S/R - DISASSEMBLER ENTRY
INSTDSP    (-1840) [$F8D0]                MONITOR & MINIASSEMBLER MEMORY LOCATION 'INSTDSP' (INSTRUCTION DISPLAY)
INSTDSP    (-640) [$FD80]                 MONITOR S/R TO DISASSEMBLE INSTRUCTION AT PCH/PCL (A- X- Y-REGS ALTERED)
INT (FPINT)    (-5085) [$EC23] \SE\       APPLESOFT FP - COMPUTES GREATES INT (FPINT)EGER VALUE OF FAC. MODIFIES CHARAC
                                          ($000D). USES QINT (FPINT). RESULT TO FAC. MODIFIES CHARAC ($000D)
(INT=>FP)    (-8471) [$DEE9] \SE\        APPLESOFT - PULL INTEGER (%) VARIABLE POINTED TO BY FACMO~FACLO ($00A0~$00A1)
                                          INTO A-REG & Y-REG AND CONVERT TO FP IN FAC. RESETS VALTYP (RESETS Y-REG TO 0)
INTOIT    (16198) [$3F46] \SL\            DOS 3.2 DISK FORMATTER INTERIOR LABEL 'INTOIT'
INVFLG    (50) [$0032] \P1\               VIDEO FORMAT CONTROL: 255($FF)=NORMAL;127($7F)=FLASHING;63($3F)=INVERSE
IOBPL~H    (72~73) [$0048~$0049] \P2\     DOS READ-WRITE-TRACK-SECTOR (RWTS)  'IOBPL~H' (INPUT-OUTPUT CONTROL BLOCK
                                          POINTER)
IOPRT    (-357) [$FE9B]                   MONITOR MEMORY LOCATION 'IOPRT'
IOPRT1    (-345) [$FEA7]                  MONITOR MEMORY LOCATION 'IOPRT1'
IOPRT2    (-343) [$FEA9]                  MONITOR MEMORY LOCATION 'IOPRT2'
IORTS    (-168) [$FF58]                   JSR HERE TO FIND OUT WHERE ONE IS. SETS OVERFLOW FLAG
IRQ    (-1472) [$FA40] \SE\               AUTOSTART ROM MONITOR S/R - IRQ HANDLER
IRQ    (-1402) [$FA86] \SE\               MONITOR S/R- IRQ HANDLER. NOTE: MOVED TO $FA40 IN AUTOSTART ROM
IRQADR~IRQLOC    (1022~1023) [$03FE~$03FF] \P2\IRQ'S VECTORED BY POINTER HERE TO SUBROUTINE TO HANDLE INTERRUPT REQUESTS
ISCNTC    (-10152) [$D858] \SE\          APPLESOFT - CHECK KEYBOARD FOR CONTROL-C ($83). EXECUTES BREAK ROUTINE IF
                                          THESE IS
ISDRV0    (15989) [$3E75] \DL\            DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'ISDRV0'
ISLETC (CHARCHEK)    (-8067) [$E07D] \SE\ APPLESOFT - CHECKS A-REG FOR ASCII LETTER OTHERWISE CLEAR IT TO ZERO ('A' TO
                                          'Z'). SET C (CARRY FLAG) TO 1 IF A IS A LETTER OTHERWISE CLEAR IT TO ZERO (A-
                                          X- Y-REGS NOT ALTERED)
ITSGOOD    (16286) [$3F9E] \SL\           DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF CONTINUATION IF GOOD
                                          CONDITION DETECTED
JJTOER    (15893) [$3E15] \SL\            DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'JJTOER'
JMPTO1    (15841) [$3DE1] \SL\            DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'JMPTO1'
JMPTOERR    (15842) [$3DE2] \SL\          DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'JMPTOERR' (JUMP TO
                                          ERROR HANDING ROUTINE HNDLERR)
KBD ~ IOADR  [$C000~] \H1\                MONITOR I/O - PEEK TO READ KEYBOARD. IF VAL>127 KEY HAS BEEN PRESSED SINCE
                                          LAST STROBED AT $C010.
KBDSTB    (-16368) [$C010] \H1\           KEYBOARD STROBE- REACTIVATES KEYBOARD SO THAT VALUE OF PRESSED KEY GOES TO
                                          $C000. SETS HIGH BITTO ZERO..-4
-------------------------------------------------------------------------------------------------------------------------------
INITAN - KBDSTB                    Prof. Luebbert's "What's Where in the Apple"                    ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

KEYIN   (-741) [$FD1B] \SE\             GETS NEXT KEY INPUT FROM KEYBOARD HARDWARE. REQUIRES LOOP TO TEST THAT KEY HAS
                                        INDEED BEEN READ; BY PRESENCE OF $80 BIT. ALSO REQUIRES KEYBOARD STROBE TO BE
                                        HIT BEFORE NEXT KEYBOARD INPUT. AUXILLIARY ACTIONS TAKEN BY KEYIN INCLUDE
                                        RESTORING TO THE SCREEN AREA THE CHARACTER MODIFIED  BY RDKEY TO REMOVE BLINK
                                        INSERTED BY RDKEY AND COUNTING UP THE RANDOM NUMBER FIELD~ IGNORING OVERFLOW.
                                        SET-UP: X-REG NOT SIGNIFICANT & NOT AFFECTED; A-REG INPUT TO THIS ROUTINE STORED
                                        AT (BASL)~Y WHEN A KEY IS PRESSED BEFORE THE A-REG IS FILLED FROM THE KEYBOARD
                                        REGISTER; Y-REG USED FOR STORING A-REG IN SCREEN AREA TO (BASL)~Y; CH AND CV NOT
                                        REFEENCED;BASL~H ARE USED AS INDICATED IN RDKEY. RESJLT: A-REG CONTAINS INPUT
                                        FROM KEYBOARD REGISTER; IT IS ONLY ITEM CHANGED (A-REG ALTERED)
KEYIN2   (-735) [$FD21]                 MONITOR MEMORY LOCATION KEYIN2
KSWL~KSWH   (56~57) [$0038~$0039] \P2\  DOS INPUT HOOK; I.E. ADDRESS OF THE USER INPUT ROUTINE. CONTROLLED BY CURRIN
                                        PORT IN# & KEYIN. RESET ~ 0 CTRL-K & IN#0 SET THIS LOCN TO $FD1B (MONITOR
                                        KEYBOARD INPUT ROUTINE); S CTRL-K &IN#S SET THIS LOCN TO $CS00(SLOT S ROM)
                                        (MONITOR INPUT REG)
L   (53) [$0035] \P1\                   MINIASSEMBER MEMORY LOCATION 'L'
(LAST CHAR PTR)   (184~185) [$00B8~$00B9] \P2\APPLESOFT PTR TO LAST CHAR OBTAINED THRU CHRGET ROUTINE
(LAST VBL NAME)   (129~130) [$0081~$0082] \P2\APPLESOFT - HOLDS LAST-USED VARIABLE'S NAME
LASTIN   (47) [$002F] \P1\              USED IN CASSETTE INPUT BY RDBIT AS WORK AREA TO DETERMINE WHETHER INPUT HAS
                                        CHANGED
LASTPT   (83) [$0053] \P1\              APPLESOFT LAST USED TEMPORARY STRING POINTER
LEAD2R   (250) [$00FA] \P1\             INTEGER BASIC MEMORY LOCATION 'LEADZR' (LEADING ZEROS INDEX)
LEADBL   (201) [$00C9]                  INTEGER BASIC MEMORY LOCATION 'LEADBL' (LEADING BLANKS INDEX)
LENGTH   (47) [$002F] \P1\              USED BY DISASSEMBLER TO INDICATE LENGTH OF THE INSTRUCTION. ALSO BY TRACE
~LEN~   (-4574) [$EE22] \SE\            INTEGER BASIC ENTRY TO FUNCTION TO OBTAIN LENGTH OF A STRING
LET   (-9658) [$DA46] \SE\              APPLESOFT LET - USES CHRGET TO GET ADDRESS OF '=';EVALUATES FORMULA & STORES IT.
                                        ON ENTRY TXTPTR POINTS TO FIRST CHAR OF VARIABLE NAME
LF   (-922) [$FC66] \SE\                MONITOR S/R TO TO PERFORM A LINE FEED; I.E. INCREMENT CV; COMPARE CV TO WNDBTM
                                        IF CV<WNDBTM GOTO VTABZ TO SET BASL~H AND RETURN ELSE DECREMENT CV AND DO SCROLL
                                        (A-REG ALTERED)
LINGET   (-9716) [$DA0C] \SE\           READ 16BIT INTEGER LINE # FROM TXTPTR INTO LINNUM. SEE APPLE ORCHARD V1#1P13 FOR
                                        DETAILS
LINNUM   (80~81) [$0050~$0051] \P2\     APPLESOFT GENERAL PURPOSE 16 BIT NUMBER LOCATION (USES INCLUED LOCATION FOR LINE
                                        NUMBER)
LINPRT   (-4828) [$ED24] \SE\           APPLESOFT - PRINTS 2-BYTE UNSIGNED NUMBER IN X-REG (MSB) & A-REG (LSB)
LIST   (-418) [$FE5E] \SE\              CALL TO DISASSEMBLE 20 INSTRUCTIONS
LIST2   (-413) [$FE63]                  MONITOR MEMORY LOCATION 'LIST2'
LMNEM~RMNEM   (44~45) [$002C~$002D] \P2\ADDRESS POINTER USED BY DISASSEMBLER FOR INDEX TO MNEMONICS TABLE
(LN(2))   (-5828~-5824) [$E93C~$E940] \P5\APPLESOFT FP CONSTANT (LN(2) = .30103...
LNAL~LNAH   (228~229) [$00E4~$00E5] \P2\INTEGER BASIC MEMORY LOCATIONS 'LNAL~LNAH' (LINE NUMBER ADDRESS)(NEXT LINE
                                        NUMBER)
(LO-RES PAGE 2)   (2048~3071) [$0800~$0BFF] \HB\SECONDARY SCREEN BUFFER (TEXT & LOW-RES GRAPHICS PAGE 2)
LO-RES   (-16298) [$C056] \H1\            POKE TO 0 TO SET FROM HI-RES TO SAME PAGE # OF LO-RES OR TEXT
(LO-RESLNS0/1)   [$C400-$0427] \BB\       VIDEO SCREEN BUFFER LO-RES LINES 0 AND 1
(LO-RESLNS10/11)   [$0680-$06A7] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 10 AND 11
(LO-RESLNS12/13)   [$0700-$0727] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 12 AND 13
(LO-RESLNS14/15)   [$0780-$07A7] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 14 AND 15
(LO-RESLNS16/17)   [$0428-$044F] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 16 AND 17
(LO-RESLNS18/19)   [$04A8-$04CF] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 18 AND 19
(LO-RESLNS2/3)   [$0480-$04A7] \BB\       VIDEO SCREEN BUFFER LO-RES LINES 2 AND 3
(LO-RESLNS20/21)   [$0528-$054F] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 20 AND 21
-------------------------------------------------------------------------------------------------------------------------------
KEYIN - (LO-RES                    Prof. Luebbert's "What's Where in the Apple"                    ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

(LO-RESLNS22/23)  [$05A8-$05CF] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 22 AND 23
(LO-RESLNS24/25)  [$0628-$064F] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 24 AND 25
(LO-RESLNS26/27)  [$06A8-$06CF] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 26 AND 27
(LO-RESLNS28/29)  [$0728-$074F] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 28 AND 29
(LO-RESLNS30/31)  [$07A8-$07CF] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 30 AND 31
(LO-RESLNS32/33)  [$0450-$0477] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 32 AND 33
(LO-RESLNS34/35)  [$04D0-$04F7] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 34 AND 35
(LO-RESLNS36/37)  [$0550-$0577] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 36 AND 37
(LO-RESLNS38/39)  [$05D0-$05F7] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 38 AND 39
(LO-RESLNS4/5)  [$0500-$0527] \BB\       VIDEO SCREEN BUFFER LO-RES LINES 4 AND 5
(LO-RESLNS40/41)  [$0650-$0677] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 40 AND 41
(LO-RESLNS42/43)  [$06D0-$06F7] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 42 AND 43
(LO-RESLNS44/45)  [$0750-$0777] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 44 AND 45
(LO-RESLNS46/47)  [$07D0-$07F7] \BB\     VIDEO SCREEN BUFFER LO-RES LINES 46 AND 47
(LO-RESLNS6/7)  [$0580-$05A7] \BB\       VIDEO SCREEN BUFFER LO-RES LINES 6 AND 7
(LO-RESLNS8/9)  [$0600-$0627] \BB\       VIDEO SCREEN BUFFER LO-RES LINES 8 AND 9
(LOAD DOS 3.2 REGS)   (1002) [$03EA] \SE\RECONNECT DOS 3.2 VIA APPLE MONITOR REGS. PREVIOUS CONTENTS OF MONITOR I/O
                                         REGS ($0036~$0039) TO DOS 3.2 INPUT & OUTPUT REGS (DOS 3.2 REGS ALTERED)
LOAD   (-10039) [$D8C9] \SE\             APPLESOFT CASSETTE - LOAD A PROGRAM FROM CASSETTE TAPE
~LOAD~   (-3873) [$F0DF]                 INTEGER BASIC ENTRY TO LOAD SUBROUTINE (LOAD A PROGRAM FROM CASSETTE TAPE)
LOC0   (0) [$0000] \P1\                  MONITOR MEMORY LOCATION 'LOC0'. PRESET TO $4C (JMP) - (JUMP ADDRESS IN
                                         $001~$002)
LOC1   (1~2) [$0001~$0002] \P2\          MONITOR MEMORY LOCATION 'LOC1' - POINTER PRESET TO ADDRESS OF APPLESOFT SOFT
                                         ENTRY
(LOG(E)2)   (-4389~-4385) [$EED8~$EEDF] \P5\APPLESOFT FP CONSTANT LOG(E)2
LOMEML~LOMEMH   (74~75) [$004A~$004B] \P2\POINTER TO LOMEM (CONTAINS 'START OF BASIC VARIABLES' FOR INTEGER BASIC -
                                         START OF PROGRAM FOR APPLESOFT BASIC)
~LOMEM~   (-3895) [$F0C9] \SE\           INTEGER BASIC ENTRY TO LOMEM ROUTINE
LOWSCR   (-16300) [$C054] \H1\           POKE TO 0 TO DISPLAY PAGE 1 (DOES NOT CLEAR SCREEN)
LOWTR   (155~156) [$009B~$009C] \P2\     APPLESOFT GENERAL PURPOSE REGISTER USED BY GETARYPT~FNDLN~BLTU (E.G. LOW END OF
                                         BLOCK TO BE TRANSFERRED IN BLTU)
LT   (-480) [$FE20]                      MONITOR MEMORY LOCATION 'LT'
LT2   (-478) [$FE22]                     MONITOR MEMORY LOCATION 'LT2'
M   (-16384~-16369) [$C000~$C00F] \H1\   EQUIVALENT ADDRESSES - ALL FOR KEYBOARD INPUT BYTE. WHEN KEY PRESSED ASCII
                                         VALUE GOES THERE AND HIGH BIT SET
M1   (249~251) [$00F9~$00FB] \P3\        FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR FP1 MEMORY LOC 'M1'
                                         (MANTISSA)
M2   (245~247) [$00F5~$00F7] \P3\        MONITOR & OLD (NON-APPLESOFT) FLOATING POINT ACCUMULATOR 2 MEMORY LOC 'M2'
                                         (MANTISSA - 3 BYTES)
(MACROLINE0)   (1024~1143) [$0400~$0477] \HB\TEXT VIDEO SCREEN DISPLAY PAGE 1 - MACROLINE ORSUBPAGE CONSISTING OF LINES
                                         0 ~ 8 & 16
(MACROLINE1)   (1152~1271) [$0480~$04F7] \HB\TEXT PAGE 1 - MACROLINE OR SUBPAGE CONSISTING OF 3 TEXT LINES OF 40 BYTES
                                         (CHARACTERS) EACH PLUS A BLOCK OF 8 I-O PERIPHERAL BYTES. SUBSEQUENT
                                         MACROLINES WILL BE OMITTED FROM DATABASE
~MAINLINE~   (-7501) [$E2B3] \SE\        INTEGER BASIC ENTRY POINT TO MAIN LINE OF COMPILE/EXECUTE CODE
~MAN~   (-4524) [$EE54] \SE\             INTEGER BASIC ENTRY TO MANUAL LINE NUMBER FUNCTION
MASK   (46) [$002E] \P1\                 LOW-RES COLOR GRAPHICS MASK. $0F OR $F0 TO SELECT HIGH OR LOW NIBBLE TO SPECIFY
                                         WHICH OF 2 PLOT LINES REP BY GBASL~H POINTER
MD1   (-1116) [$FBA4]                    MONITOR 16-BIT MULITIPLY/DIVIDE SIGN-PROCESSOR. SETS ABSOLUTE VALUES OF ACL~H
                                         MEMORY LOCATION 'MD1' AUXL~H LEAVING RESULTING SIGN IN LSB OF SIGN ($002F)
MD2   (-1105) [$FBAF]                    MONITOR MEMORY LOCATION 'MD2'
-------------------------------------------------------------------------------------------------------------------------------
(LO-RESLNS22/23) - MD2          Prof. Luebbert's "What's Where in the Apple"          ALPHABETICAL GAZETTEER
```

MD3 (-1100) [$FBB4] MONITOR MEMORY LOCATION 'MD3'

MDRTS (-1088) [$FBC0] MONITOR MEMORY LOCATION 'MDRTS'

MEMFUL (-7317) [$E36B] \SE\ INTEGER BASIC MEMORY FULL ERROR

MEMSIZE (115~116) [$0073~$0074] \P2\ APPLESOFT HIMEM (HIGHEST LOC IN MEM AVAIL + 1). INIT TO HIGHEST RAM - $BFFF FOR 48K APPLE IF DOS NOT ACTIVE BEGINNING OF DOS IF DOS ACTIVE

MINASM (-2458) [$F666] TURN ON MINIASSEMBLER (KEYBOARD INPUT WILL BE INTERPRETED AS A SEMBLY-LANGUAGE INSTRUCTION)

(MINUS.ONE.HALF) (-5833~-5813) [$E937~$E94B] \P5\APPLESOFT FP CONSTANT MINUS ONE HALF (-1/2)

MIXCLR (-16302) [$C052] \H1\ POKE TO 0 TO RESET FROM MIXED GRAPHICS (W/4 LINES TEXT) TO FULL-SCREEN GRAPHICS

MIXSET (-16301) [$C053] \H1\ POKE=0 TO SET TEXT/GRAPHICS MIX (BOTTOM 4 LINES TEXT)

MNEML (-1600) [$F9C0] MONITOR & MINIASSEMBLER MEMORY LOCATION 'MNEML'

MNEMR (-1536) [$FA00] MONITOR & MINIASSEMBER MEMORY LOCATION 'MNEMR'

MNNDX1 (-1858) [$F8BE] MONITOR MEMORY LOCATION 'MNNDX1'

MNNDX2 (-1854) [$F8C2] MONITOR MEMORY LOCATION 'MNNDX2'

MNNDX3 (-1847) [$F8C9] MONITOR MEMORY LOCATION 'MNNDX3'

MOD8CHK (-595) [$FDAD] MONITOR MEMORY LOCATION 'MOD8CHK'

MODE (49) [$0031] \P1\ USED BY MONITOR COMMAND PROCESSING TO INDICATE DISPOSITION OF HEX INFO IN THE INPUT LINE

~MOD~ (-7558) [$E27A] \SE\ INTEGER BASIC ENTRY POINT TO MODULO FUNCTION

MON (-155) [$FF65] \SE\ MONITOR S/R- NORMAL ENTRY TO 'TOP' OF MONITOR WHEN RUNNING (BEEPS!)

(MONITOR RESVD) (32~85) [$0020~$0055] \PB\APPLE II SYSTEM MONITOR RESERVED LOCATIONS ($0050~$0055 USED ONLY BY MULTIPLY-DIVIDE ROUTINES AND THUS AVAILABLE IN MANY SITUATIONS)

MONTIME (70) [$0046] \P1\ DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER 'MONTIME'

MONZ (-151) [$FF69] \SE\ MONITOR S/R TO RESET AND ENTER MONITOR (NO BEEP)

MOTOF (15741) [$3D7D] \SL\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL - STARTSCODE TO DELAY UNTIL MOTOR UP TO SPEED

MOTOROFF (-16248) [$C088] \P1\ DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'MOTOROFF'

MOTORON (-16247) [$C089] \P1\ DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'MOTORON'

MOV1F (-5343) [$EB21] \SE\ APPLESOFT FP - PACK FAC AND MOVE IT INTO TEMP1 ($0093~$0097). USES MOVMF. ON EXIT A-REG & Z FLAG REFLECT FACEXP. MODIFIES INDEX ($005E~$005F) (RESET Y-REG=0)

MOV2F (-5346) [$EB1E] \SE\ APPLESOFT FP - PACK FAC AND MOVE IT INTO TEMP2 ($0098~$009C). USES MOVMF. ON EXIT A-REG & Z FLAG REFLECT FACEXP (RESET Y-REG=)

MOVAF (TR1=>2) (-5277) [$EB63] \SE\ APPLESOFT FP - PACK EXTENSION BYTE INTO FAC & MOVE FAC INTO ARG. ON EXIT A-REG = FACEXP AND ZERO FLAG IS SET. RESET EXTENSION BYTE = 0 (RESET X-REG=0)

MOVE (-468) [$FE2C] \SE\ MONITOR S/R TO PERFORM A MEMORY MOVE (A1~A2 TO A4)(Y-REG MUST =0 AT CALL) (A-REG ALTERED)

MOVFA (TR2=>1) (-5293) [$EB53] \SE\ APPLESOFT FP - MOVE ARG INTO FAC. ON EXIT A-REG = FACEXP AND ZERO FLAG IS SET

MOVFM (FPLOAD) (-5383) [$EAF9] \SE\ APPLESOFT FP MOVE MEMORY POINTED TO BY Y-REG & A-REG INTO FAC. ON EXIT A-REG & ZERO FLAG REFLECT FACEXP. RESET EXTENSION BYTE=0 (RESET Y-REG=0)

MOVINS (-6700) [$E5D4] \SE\ APPLESOFT - MOVE STRING WHOSE DESCRIPTOR IS POINTED TO BY STRNG1 TO MEM LOC POINTED TO BY FORPNT

MOVMF (FPSTR) (-5333) [$EB2B] \SE\ APPLESOFT FP - PACK FAC AND MOVE IT INTO MEMORY POINTED TO BY Y-REG (MSB) & X-REG (LSB). ON EXIT A-REG & ZERO FLAG REFLECT FACEXP. MODIFIES INDEX ($005E~$005F)

MOVML (-5341) [$EB23] \SE\ APPLESOFT FP - PACK FAC AND MOVE IT INTO ZERO PAGE AREA POINTED TO BY X-REG. USES MOVMF. ON EXIT A-REG & Z FLAG REFLECT FACEXP

MOVSTR (-6686) [$E5E2] \SE\ APPLESOFT - MOVE STRING POINTED TO BY Y-REG (MSB) & X-REG (LSB) WITH LENGH IN A-REG TO MEMORY POINTED TO BY FRESPA

MSWAIT (-17920) [$BA00] \SB\ DOS 3.3 RWTS OPERATION TIMER ROUTINE

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

MUL    (-1181) [$FB63] \SE\                    MONITOR - UNSIGNED 16-BIT MULTIPLY S/R (NOT AVAILABLE WITH AUTOSTART ROM).
                                               SAME AS MULPM ($FB60) EXCEPT UNSIGNED. SEE 'SIGN' AT $002F (A- X- Y-REGS
                                               ALTERED)
MUL10    (-5575) [$EA39] \SE\                  APPLESOFT FP - MULTIPLY FAC BY 10. WORKS FOR BOTH POSITIVE & NEGATIVE NUMBERS
MUL2    (-1179) [$FB65]                        MONITOR MEMORY LOCATION 'MUL2'
MUL3    (-1171) [$FB6D]                        MONITOR MEMORY LOCATION 'MUL3'
MUL4    (-1162) [$FB76]                        MONITOR MEMORY LOCATION 'MUL4'
MUL5    (-1160) [$FB78]                        MONITOR MEMORY LOCATION 'MUL5'
MULPM    (-1184) [$FB60] \SE\                  MONITOR - SIGNED 16-BIT MULTIPLY LEAVING SIGN IN LSB OF 'SIGN' (A- X- Y-REGS
                                               ALTERED)
MULPM    (-1184~-1152) [$FB60~$FB80] \SB\      MONITOR 16-BIT MULTIPLY S/R (NOT IN AUTOSTART ROM). MULTIPLIER IN AUXL~AUXH
                                               ($0054~$0055); MULTIPLICAND IN ACL~ACH ($0050~$0051);XTNDL~XTNDH
                                               ($0052~$0053) CLEARED TO ZEROS; RESULT GOES TO EXTENDED AC ($0050~$0053).
                                               ALSO SEE 'SIGN' AT $002F. (A- X-REGSY-REG ALTERED)
~MULT~    (-7646) [$E222] \SE\                 INTEGER BASIC ENTRY POINT TO MULTIPLY ROUTINE
MYSEEK    (15931) [$3E3B] \SE\                 DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF ROUTINE
                                               WHICH SEEKS TRACK 'N' IN SLOT #X/$10. (IF DRIVENO IS - THEN DRIVE 0;IF
                                               DRIVENO IS + THEN DRIVE 1
MYSEEK    (-16806~-16755) [$BE5A~$BE8D]        DOS 3.3 - HOUSEKEEPING BEFORE 'SEEKABS'. DETERMINES NUMBER OF PHASES PER
                                               TRACK & STORES TRACK INFO IN APPROPRIATE SLOT-DEPENDENT LOCN
NBITS    (1912+S) [$0778+S] \P1\               EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S NUMBER OF DATA BITS PLUS 1 FOR
                                               START BIT
NBRNCH    (-1269) [$FB0B]                      MONITOR MEMORY LOCATION 'NBRNCH'
NEGOP    (-4400) [$EED0] \SE\                  APPLESOFT FP - LET FAC = -FAC (X- Y-REGS NOT ALTERED)
NEWMON    (-1407) [$FA81]                      AUTOSTART MONITOR MEMORY LOCATION 'NEWMON'
NEWPCL    (-1331) [$FACD]                      MONITOR MEMORY LOCATION 'NEWPCL'
NEWSTT    (-10286) [$D7D2] \SE\                APPLESOFT - EXECUTE A NEW STATEMENT. ON ENTRY TXTPTR POINTS TO THE':'
                                               PRECEDING THE STMT OR ZERO AT END OF PREVIOUS LIN. USE NEWSTT TO RESTART THE
                                               PROGRAM WITH CONT. THIS ROUTINE DOES NOT RETURN
~NEW~    (-6739) [$E5AD] \SE\                  INTEGER BASIC ENTRY POINT TO CLEAR OUT OLD PROGRAM AND RESET POINTERS FOR A
                                               NEW PROGRAM
(NEXT W/O FOR PRT)    (-8949) [$DD0B] \SE\     APPLESOFT - PRINT ERROR MESSAGE "NEXT WITHOUT FOR" THEN HALT AT APPLESOFT
                                               (]) LEVEL
NEXTOP    (-2692) [$F57C]                      MINIASSEMBLER MEMORY LOCATION 'NEXTOP'
~NEXT~    (-5930) [$E8D6] \SE\                 INTEGER BASIC ENTRY TO ROUTINE TO HANDLE 'NEXT' LOOP END
NMI    (1019) [$03FB]                         NMI'S VECTORED TO THIS LOCATION
~NODSP~    (-3360) [$F2E0] \SE\                INTEGER BASIC ENTRY TO ROUTINE TO TURN OFF DISPLAY FUNCTION
NOGOOD    (16276) [$3F94] \SL\                 DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF CLEAN UP IF NOGOOD
                                               CONDITION DETECTED
NORM    (-2973) [$F463] \SE\                   NORMALIZE FLOATING POINT NUMBER IN FP1 (A-REG ALTERED)
NOTCR    (-707) [$FD3D]                        MONITOR MEMORY LOCATION 'NOTCR'
NOTCR1    (-673) [$FD5F]                       MONITOR MEMORY LOCATION 'NOTCR1'
(NOTFAC)    (-8552) [$DE98] \SE\               APPLESOFT - LET FAC = NOT(FAC); I.E. RETURNS FAC=1 IF FAC=0 OR FAC=0 IF
                                               FAC<>0
~NOTRACE~    (-3722) [$F176] \SE\              INTEGER BASIC ENTRY TO ROUTINE TO TURN OFF TRACE MODE
NOTSURE    (15651) [$3D23] \SL\                DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - AT THIS POINT
                                               PROGRAM NOT SURE WHETHER MOTOR IS RUNNING (STABLE LONG ENOUGH)
~NOT~    (-6346) [$E736] \SE\                  INTEGER BASIC ENTRY TO 'NOT' (NOT A VALUE FUNCTION)
NOUNSTKC    (160~191) [$00A0~$00BF]            INTEGER BASIC MEMORY LOCATION 'NOUNSTKC' (NOUN STACK COUNTER)
NOUNSTKH    (120~151) [$0078~$0097]            INTEGER BASIC MEMORY LOCATION 'NOUNSTKH' (NOUN STACK HI BYTE)
NOUNSTKL    (80~87) [$0050~$0057] \P8\         INTEGER BASIC MEMORY LOCATION 'NOUNSTKL'
-------------------------------------------------------------------------------------------------------------------------------
MUL - NOUNSTKL                          Prof. Luebbert's "What's Where in the Apple"                    ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

NREL   (-2696) [$F578]                  MINIASSEMBLER MEMORY LOCATION 'NREL'
NXTA1   (-838) [$FCBA] \SE\             MONITOR S/R TO INCREMENT A1 (16 BITS). SET CARRY IF RESULT >=A2. (A-REG ALTERED)
NXTA4   (-844) [$FCB4] \SE\             MONITOR S/R TO INCREMENT A4 (16 BITS) THEN DO NXTA1 (A-REG ALTERED)
NXTBAS   (-104) [$FF98]                 MONITOR MEMORY LOCATION 'NXTBAS'
NXTBIT   (-112) [$FF90]                 MONITOR MEMORY LOCATION 'NXTBIT'
NXTBS2   (-94) [$FFA2]                  MONITOR MEMORY LOCATION 'NXTBS2'
NXTBYT   (-1337) [$FAC7]                AUTOSTART MONITOR MEMORY LOCATION 'NXTBYT'
"NXTBYTE"   (-8150) [$E02A] \SE\        INTEGER BASIC ENTRY POINT TO GET NEXT BYTE 16-BIT POINTER
NXTCHAR   (-651) [$FD75] \SE\           TOP POINT IN CHAR INPUT LOOP. SAME EFFECT AS GETLN EXCEPT BYPASS PRINT OF PROMPT
                                        CHARACTER; ON SET-UP X-REG SHOULD BE SET TO ZERO TO BEGIN STORING OF INPUT AT
                                        $200; A- Y-REGS NOT SIGNIFICANT;CV AND BASL"H SHOULD BE COMPATIBLE POINTING IN
                                        THE SCROLL WINDOW; CH INDICATES WHERE ECHOING OF KEYBOARD INPUT IS TO START &
                                        SHOULD BE LESS THAN WNDWDTH; RESULTS SAME AS FOR GETLNZ (A- X- Y-REGS ALTERED)
NXTCHR   (-83) [$FFAD]                  MONITOR - TOP POINT IN GETLN CHARACTER INPUT LOOP;RDCHAR CALLED TO GET CHAR INTO
                                        A-REG; ON RETURN A-REG TESTED FOR PRESENCE OF CTRL-U (RIGHT ARROW); IF SO A-REG
                                        LOADED FROM SCREEN MEMORY ASSUMING Y-REG CONTAINS SAME VALUE AS CH; IF A-REG
                                        VAL >$DF LOWER-CASE LETTER CONVERTED TO UPPER CASE; IF CHAR IS A C/R IT IS
                                        PRINTED THROUGH COUT AND RTS EXIT OF COUT WILL GIVE CONTROL BACK TO CALLING
                                        PROGRAM W/ X-REG INDICATING INPUT CHAR COUNT +1; THAT IS INPUT IS IN LOCNS $200
                                        THRU $200"X WHERE $200"X CONTAINS A C/R; ON SET-UP A- X- Y-REGS NOT SIGNIFICANT;
                                        CV & BASL"H SHOULD BE COMPATIBLE (POINTING IN THE SCROLL WINDOW);CH INDICATES
                                        HORIZ POSN IN SCROLL WINDOW WHERE CURSOR WILL BE INDICATED BY BLINKING. ON
                                        RETURN CALLER A-REG WILL CONTAIN KEY VALUE;Y-REG WILL CONTAIN CONTENTS OF
                                        CH;X-REG WILL CONTAIN SAME VALUE AS INPUT; CV CH & BASL"H WILL HAVE CHANGE ONLY
                                        IF AN ESCAPE KEY SEQUENCE HAS BEEN PERFORMED
NXTCOL   (-1953) [$F85F] \SE\           MONITOR LO-RES S/R. CHANGE COLOR TO (COLOR)+3 (A-REG ALTERED)
NXTCOL   (-1803) [$F8F5]                AUTOSTART MONITOR MEMORY LOCATION 'NXTCOL'
NXTITM   (-141) [$FF73]                 MONITOR MEMORY LOCATION 'NXTITM'
NXTLINE   (-2667) [$F595]               MINIASSEMBLER MEMORY LOCATION 'NXTLINE'
NXTL"NXTH   (230"231) [$00E6"$00E7] \P2\INTEGER BASIC MEMORY LOCATIONS 'NXTL"NXTH' (NEXT POINTER)
NXTM   (-2624) [$F5C0]                  MINIASSEMBLER MEMORY LOCATION 'NXTM'
NXTM2   (-2613) [$F5CB]                 MINIASSEMBLER MEMORY LOCATION 'NXTM2'
NXTMN   (-2627) [$F5BD]                 MINIASSEMBLER MEMORY LOCATION 'NXTMN'
NXTPRT   (16086) [$3ED6] \SL\           DOS 3.2 DISK FORMATTER - LABEL AT POINT WHERE CHECK IS MADE TO SEE IF TRACK DONE
NXTTRY   (16208) [$3F50] \SL\           DOS 3.2 DISK FORMATTER INTERIOR LABEL 'NXTTRY'
OK   (15710) [$3D5E] \SL\               DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE THAT IT IS
                                        OKAY TO CONTINUE
(OLD TEXT PTR)   (121"122) [$0079"$007A] \P2\APPLESOFT OLD TEXT PTR. PTS TO LOC IN MEM FOR NEXT STMT TO BE EXE
OLDBRK   (-1447) [$FA59]                AUTOSTART MONITOR MEMORY LOCATION 'OLDBRK'
OLDLIN   (119"120) [$0077"$0078] \P2\   APPLESOFT - LAST LINE EXECUTED -  LINE # AT WHICH EXECUTION INTERRUPED BY CTRL-C
                                        STOP ETC.
ONDRV0   (16027) [$3E9B] \DL\           DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'ONDRV0'
(ONE)   (-5869"-5865) [$E913"$E917] \P5\APPLESOFT FP CONSTANT ONE =1.
(ONE-QUARTER)   (-3984"-3979) [$F070"$F075] \P5\APPLESOFT5-BYTE FLOATING POINT CONSTANT 1/4 (0.25)
(ONE.BILLION)  [$ED14"$$ED18] \P5\      APPLESOFT 5-BYTE FLOATING POINT CONSTANT 1000000000 (1E9)
(ONE.HALF)   (-4508"-4504) [$EE64"$EE68] \P5\APPLESOFT 5-BYTE FP CONSTANT ONE HALF (1/2)
ONEDLY   (-798) [$FCE2]                 MONITOR MEMORY LOCATION 'ONEDLY'
ORMASK   (243) [$00F3] \P1\             MASK FOR OUTPUT CONTROL: NORMAL/FLASHING/INVERSE
(OUT OF MEM PRT)   (-11248) [$D410]     APPLESOFT - PRINT "OUT OF MEMORY" THEN HALT AT APPLESOFT (]) LEVEL
OUTDO   (-9380) [$DB5C] \SE\            APPLESOFT - PRINT THE CHARACTER IN A-REG. INVERSE"FLASH"NORMAL OPTIONS IN EFFECT
OUTPORT   (-363) [$FE95]                MONITOR MEMORY LOCATION 'OUTPORT'
-------------------------------------------------------------------------------------------------------------------------------
NREL - OUTPORT                  Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

OUTPRT (-361) [$FE97] MONITOR MEMORY LOCATION 'OUTPRT'

OUTQST (-9382) [$DB5A] \SE\ APPLESOFT - PRINT A QUESTION MARK

OUTSPC (-9385) [$DB57] \SE\ APPLESOFT - PRINT A SPACE

OUTVAL (200) [$00C8] INTEGER BASIC MEMORY LOCATION 'OUTVAL' (OUTPUT VALUE TEMPORARY)

(OVERFLOWPRT) (-5931) [$E8D5] \SE\ PRINT "OVERFLOW" THEN HALT AT THE APPLESOFT (]) LEVEL

P1L~P1H (50~227) [$0032~$00E3] \P2\ INTEGER BASIC MEMORY LOCATIONS 'P1L~P1H' (AUXILIARY POINTER ONE)

P2L~P2H (228~229) [$00E4~$00E5] \P2\ INTEGER BASIC MEMORY LOCATIONS 'P2L~P2H' (AUXILIARY POINTER TWO)

P3L~P3H (230~231) [$00E6~$00E7] \P2\ INTEGER BASIC MEMORY LOCATIONS 'P3L~P3H' (AUXILIARY POINTER THREE)

PADDL0 (-16284) [$C064] \H1\ MONITOR MEMORY LOCATION PADDL0; HARDWARE INDISTINGUISHABLE FROM $C06C; STATE OF TIMER OUTPUT FOR PADDLE 0 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PADDL0 (-16276) [$C06C] \H1\ MONITOR MEMORY LOCATION PADDL0; HARDWARE INDISTINGUISHABLE FROM $C064; STATE OF TIMER OUTPUT FOR PADDLE 0 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PADDL1 (-16283) [$C065] \H1\ MONITOR MEMORY LOCATION PADDL1; HARDWARE INDISTINGUISHABLE FROM $C06D; STATE OF TIMER OUTPUT FOR PADDLE 1 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PADDL1 (-16275) [$C06D] \H1\ MONITOR MEMORY LOCATION PADDL1; HARDWARE INDISTINGUISHABLE FROM $C065; STATE OF TIMER OUTPUT FOR PADDLE 1 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PADDL2 (-16282) [$C066] \H1\ MONITOR MEMORY LOCATION PADDL2; HARDWARE INDISTINGUISHABLE FROM $C06E; STATE OF TIMER OUTPUT FOR PADDLE 2 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PADDL2 (-16274) [$C06E] \H1\ MONITOR MEMORY LOCATION PADDL2; HARDWARE INDISTINGUISHABLE FROM $C066; STATE OF TIMER OUTPUT FOR PADDLE 2 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PADDL3 (-16281) [$C067] \H1\ MONITOR MEMORY LOCATION PADDL3; HARDWARE INDISTINGUISHABLE FROM $C06F; STATE OF TIMER OUTPUT FOR PADDLE 3 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PADDL3 (-16273) [$C06F] \H1\ MONITOR MEMORY LOCATION PADDL3; HARDWARE INDISTINGUISHABLE FROM $C067; STATE OF TIMER OUTPUT FOR PADDLE 3 APPEARS IN BIT 7 (NEGATIVE UNTIL TIMER EXPIRES)

PARCHK (-8526) [$DEB2] \SE\ APPLESOFT PARENTHESIS CHECK - CHECK FOR '('; EVALUATE FORMULA; CHECK FOR ')'. USES CHKOPN & FRMEVL THEN FALLS INTO CHKCLS

PCADJ (-1709) [$F953] MINIASSEMBLER MEMORY LOCATION 'PCADJ' (PROGRAM COUNTER ADJUST: 0=1 BYTE; 1=2 BYTES; 2=3 BYTES)

PCADJ2 (-1708) [$F954] MONITOR & MINIASSEMBLER MEMORY LOCATION 'PCADJ2'

PCADJ3 (-1706) [$F956] MONITOR MEMORY LOCATION 'PCADJ3'

PCADJ4 (-1700) [$F95C] MONITOR MEMORY LOCATION 'PCADJ4'

PCINC2 (-1363) [$FAAD] MONITOR MEMORY LOCATION 'PCINC2'

PCINC3 (-1361) [$FAAF] MONITOR MEMORY LOCATION 'PCINC3'

PCL~PCH (58~59) [$003A~$003B] \P2\ SAVE AND CONTROL AREA FOR PROGRAM COUNTER. USED IN BREAK PROCESSING AND MINIASSEMBLER. SET BY MONITOR CMDS L G S &T (PC SAVED HERE BY MONITOR)

~PDL~ (-3269) [$F33B] \SE\ INTEGER BASIC ENTRY TO ROUTINE TO READ A PADDLE

~PEEK~ (-4362) [$EEF6] \SE\ INTEGER BASIC ENTRY TO ROUTINE TO 'PEEK' AT THE CONTENTS OF A MEMORY LOCATION

PHASON (-16255) [$C081] \P1\ DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'PHASON'

PHSOFF (-16254) [$C082] \P1\ DOS 3.2 READ\WRITE TRACK~SECTOR (RWTS) PACKAGE PARAMETER 'PHSOFF'

PHSOFF~PHSON (-16256~-16255) [$C080~$C081] \P4\DOS 3.2 READ\WRITE TRACK\SECTOR PACKAGE PARAMETER STATEMACHINE CONTROLS TABLE: LO LO=READ; HI LO=SENSE WRITE PROTECT; LO HI=WRITE; HI HI=WRITE LOAD

(PI/2) (-3997~-3993) [$F063~$F067] \P5\APPLESOFT 5-BYTE FLOATING POINT CONSTANT PI/2 = 1.508..

PLOT (-2048) [$F800] \SE\ LO-RES PLOT POINT AT X-COORD=(Y-REG) Y-COORD=(A-REG) LEAVING GBASL~H AND MASK SET (SEE CALL-APPLE DEC 78) (A-REG ALTERED)

PLOT1 (-2034) [$F80E] \SE\ LO-RES PLOT A POINT X-COORD=(Y-REG) Y-COORD PER GBASL~H & MASK (A-REG ALTERED)

PLOTFNS (-3604) [$F1EC] \SE\ APPLESOFT - GET 2 LO-RES PLOTTING COORDS SEPARATED BY COMMA FM TXTPTR. PUT FIRST # IN FIRST AND SECOND # IN H2 & V2

~PLOT~ (-4545) [$EE3F] \SE\ INTEGER BASIC ENTRY TO ROUTINE TO DO A LO~RES PLOT (I.E. PLOT A COLORED SQUARE ON LO-RES SCREEN)

PNL~PNH (222~223) [$00DE~$00DF] \P2\ INTEGER BASIC MEMORY LOCATIONS 'PNL~PNH' (CURRENT NOUN POINTER)

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------

"POP"   (-3737) [$F167] \SE\              INTEGER BASIC ENTRY TO ROUTINE TO POP THE RETURN STACK FOR GOSUB
POSTNB16   (-18238"-18213) [$B8C2"$B8DB] \SB\DOS 3.3 POSTNIBBLE ROUTINE. CONVERTS 342 6-BIT NIBBLES OF FORM 00XXXXXX TO
                                         256 8-BIT BYTES. NIBLES STORED AT PRIMARY ($BB00-$BBFF) AND SECONDARY
                                         ($BC00"$BC55) BUFFERS. POINTER TO DATA PARGE STORED AT 'BUFPTR'
                                         ($003E"$003F). ON ENTRY X-REG= SLOT*16; CSW ($0036"$0037) POINTS TO USER
                                         DATA; $0026= BYTE COUNT IN SECONDARY BUFFER. ON EXIT CARRY SET 'BUFPTR'
                                         Y-REG CONTAINS BYTE COUNT IN SECONDARY BUFFER
POSTNIBL (DOS 3.2)  [$B9C1"BA1D] \SB\    DOS 3.1"3.2"3.2.1 (SEE $B8C2 FOR DOS 3.3) RWTS (READ-WRITE TRACK SECTOR)
                                         POSTNIBL (DOS 3.2) MODULE. CONVERTS A BUFFER OF 410 ($19A) LEFT-JUSTIFIED
                                         5-BIT NIBBLES TO 256 ($100) REAL BYTES. $003E"$003F POINTS TO BUFFER TO PUT
                                         THEM INTO
POSTNIBL (DOS 3.3)   (-18238) [$B8C2] \SB\DOS 3.3 'POSTNIBL'
PPL"PPH   (202"203) [$00CA"$00CB] \P2\   INTEGER BASIC PROGRAM POINTER (START-OF-PROGRAM EQUAL TO HIMEM IF NO PROGRAM)
"PR#S"   (-3127) [$F3C9] \SE\            INTEGER BASIC ENTRY TO ROUTINE TO SET OUTPUT PORT
PRA1   (-622) [$FD92] \SE\               PRINT CARRIAGE RET; THEN HEX OF A1H"A1L; THEN MINUS SIGN (A- X- Y-REGS ALTERED)
PRADR1   (-1776) [$F910]                 MONITOR MEMORY LOCATION 'PRADR1' (PRINT ADDRESS)
PRADR2   (-1772) [$F914]                 MONITOR MEMORY LOCATION 'PRADR2'
PRADR3   (-1754) [$F926]                 MONITOR MEMORY LOCATION 'PRADR3'
PRADR4   (-1750) [$F92A]                 MONITOR MEMORY LOCATION 'PRADR4'
PRADR5   (-1744) [$F930]                 MONITOR MEMORY LOCATION 'PRADR5'
PRBL2   (-1716) [$F94C] \SE\             MONITOR S/R- PRINT BLANKS: X REG CONTAINS NUMBER TO PRINT. CLOBBERS AC"X (A-
                                         X-REGS ALTERED)
PRBL3   (-1716) [$F94C] \SE\             PRINT A-REG FOLLOWED BY (X-REG)-1 BLANKS (A- X-REGS ALTERED)
PRBLNK   (-1720) [$F948] \SE\            PRINT THREE BLANKS THROUGH COUT (A- X-REGS ALTERED)
PRBYTE   (-550) [$FDDA] \SE\             MONITOR S/R TO PRINT CONTENTS OF A-REG AS 2 HEX DIGITS (A-REG ALTERED)
PREAD   (-1250) [$FB1E] \SE\             MONITOR S/R TO READ PADDLE. X-REG CONTAINS PADDLE NUMBER (0-3) OF PADDLE TO BE
                                         READ. PADDLE VALUE TO Y-REG (A- Y-REGS ALTERED)
PREAD2   (-1243) [$FB25]                 MONITOR MEMORY LOCATION 'PREAD2'
PRENIBL-PRENIB16   (-18432"-18327) [$B800"$B869] \SB\DOS 3.1"3.2"3.3 RWTS (READ-WRITE TRACK-SECTOR) PRENIBL MODULE.
                                             CONVERTS A PAGE OF 256 OF REAL BYTES TO A SECTOR OF 410 ($19A)
                                             RIGHT JUSTIFIED 5 BIT NIBBLES (EXCEPT DOS 3.3 CONVERTS TO 342 6
                                             BIT NIBBLES OF THE FORM 00XXXXXX). POINTER TO PAGE TO CONVERT AT
                                             $003E"$003F; DATA STORED AT PRIMARY XXX) SECONDARY BUFFERS; ON
                                             EXIT X-REG XXX) Y-REG CONTAIN $FF & CARRY SET.
PRERR   (-211) [$FF2D] \SE\              MONITOR S/R TO PRINT "ERR" AND SOUND BELL. (A- Y-REGS(?) ALTERED)
PRGEND   (175"176) [$00AF"$00B0] \P2\    APPLESOFT POINTER TO END OF PROGRAM. NOT CHANGED BY LOMEM:
PRHEX   (-541) [$FDE3] \SE\              MONITOR S/R TO PRINT RIGHT NIBBLE OF A-REG AS A SINGLE HEX DIGIT (A-REG
                                         ALTERED)
PRHEXZ   (-539) [$FDE5]                  MONITOR MEMORY LOCATION 'PRHEXZ'
PRINOW   (215) [$00D7] \P1\              INTEGER BASIC MEMORY LOCATION 'PRINOW' (PRINT IT NOW FLAG)
"PRINT"   (-4397) [$EED3] \SE\           INTEGER BASIC ENTRY POINT TO PRINT ERROR MESSAGE/BELL
PRL"PRH   (220"221) [$00DC"$00DD] \P2\   INTEGER BASIC MEMORY LOCATIONS 'PRL"PRH' (CURRENT LINE VALUE)
PRMN1   (-1803) [$F8F5]                  MONITOR MEMORY LOCATION 'PRMN1' (PRINT MNEMONIC)
PRMN2   (-1799) [$F8F9]                  MONITOR MEMORY LOCATION 'PRMN2'
PRNTAX   (-1727) [$F941] \SE\            MONITOR S/R-PRINT CONTENTS OF A-REG & X-REG AS HEX DIGITS (A- X-REGS
                                         ALTERED)
PRNTBL   (-1829) [$F8DB]                 MONITOR MEMORY LOCATION 'PRNTBL'
PRNTFAC   (-4818) [$ED2E] \SE\           APPLESOFT - PRINTS & DESTROYS CURRENT VALUE OF FAC. USES FOUT & STROUT
PRNTOP   (-1836) [$F8D4]                 MONITOR MEMORY LOCATION 'PRNTOP' (PRINT OPERATION CODE)
"PRNTSTR"   (-4605) [$EE03] \SE\         INTEGER BASIC ENTRY TO FUNCTION WHICH PRINTS A STRING
PRNTX   (-1724) [$F944] \SE\             PRINT CONTENTS OF X-REG AS HEX DIGITS (A- X-REGS ALTERED)
-------------------------------------------------------------------------------------------------------------------
POP - PRNTX                    Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

PRNTYX    (-1728) [$F940] \SE\                  MONITOR S/R- PRINT CONTENTS OF Y AND X AS 4 HEX DIGITS (A- X-REGS ALTERED)
PROGIO    (-9983) [$D901] \SE\                  APPLESOFT CASSETTE - SET UP A1 & A2 TO SAVE PROGRAM TEXT ON CASSETTE
PROMPT    (51) [$0033] \P1\                     PROMPT CHARACTER WRITTEN TO SCREEN WHENEVER A LINE OF INPUT IS CALLED FOR
                                                BY GETLN ROUTINE
"PRTERR"    (-3743) [$F161] \SE\                INTEGER BASIC ENTRY TO ROUTINE TO PRINT AN ERROR MESSAGE
PRYX2    (-618) [$FD96] \SE\                    MONITOR S/R TO PRINT CAR RET THEN HEX OF Y-REG & X-REG THEN A DASH (A-REG
                                                ALTERED)
PTRGET    (-8221) [$DFE3] \SE\                  APPLESOFT - READ VAR NAME FROM CHRGET AND FIND IT IN MEMORY (OR CREATE
                                                APPROPRIATE SIMPLE VARIABLE OR ARRAY). DOES MUCH HOUSEKEEPING
PTRIG    (-16272"-16257) [$C070"$C07F] \H1\    ALL 16 ADDRESSES DECODE TO SINGLE SWITCH WHICH TRIGGERS PADDLE TIMERS
                                                DURING PHI-2
PTRIG    (-16272"-16257) [$C070"$C07F] \H1\    GAME CONTROLLER STROBE. WHEN READ CAUSES FALG INPUTS OF GAME CONTROLLERS
                                                TO GO OFF & TIMING LOOPS RESTARTED
PTRMOV    (15684) [$3D44] \SL\                  DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE TO
                                                MOVE OUT ALL POINTERS FROM IOB (IN-OUT-BLOCK) TO ZERO PAGE
PUTNEW    (-7126) [$E42A] \SE\                  APPLESOFT -- STRING FUNCTION RETURNING WITH RESULT INDSCTMP. MOVE DSCTMP TO
                                                TEMP DESCRIPTOR & PUT POINTER TO DESCRIPTOR IN FACMO"FACLO & FLAG RESULT
                                                AS STRING
PVL"PVH    (204"205) [$00CC"$00CD] \P2\        INTEGER BASIC CURRENT VARIABLE POINTER (END OF CURRENT VARIABLE EQUAL TO
                                                LOMEM IF NO ACTIVE CURRENT VARIABLE )
PWDTH    (1784+S) [$06F8+S] \P1\                EXAMPLE:APPLE SERIAL INTERFACE CARD IN SLOT #S - PRINTER WIDTH ('PWDTH')
PWRCON    (-1283) [$FAFD]                       AUTOSTART MONITOR MEMORY LOCATION 'PWRCON'
PWREDUP    (1012) [$03F4] \P1\                  AUTOSTART ROM POWER UP MASK. SET BY SETPWRC TO EXCLUSIVE 'OR' OF $03F3 &
                                                $00A5
PWRUP    (-1370) [$FAA6]                        AUTOSTART MONITOR MEMORY LOCATION 'PWRUP'
PXL"PXH    (224"225) [$00E0"$00E1] \P2\         INTEGER BASIC MEMORY LOCATIONS 'PXL"PXH' (CURRENT VERB POINTER)
Q6L\Q6H    (-16244"-16243) [$C08C"$C08D] \P2\   DOS 3.2 READ"WRITE TRACK\SECTOR PACKAGE PARAMETER 'Q6L"Q6H' (Q6 LOW CAUSES
                                                DOS 3.2 TO READ A BYTE)
Q7L\Q7H    (-16242"-16241) [$C08E"$C08F] \P2\  DOS 3.2 READ"WRITE TRACK\SECTOR PACKAGE PARAMETER 'Q7L"Q7H' (Q7 LOW SETS
                                                DOS 3.2 FOR READ MODE)
QDRNT    (83) [$0053] \P1\                      HI-RES GRAPHICS QDRNT: 2 LSB'S ARE ROTATION QUADRANT FOR DRAW
QINT    (-5134) [$EBF2] \SE\                    APPLESOFT QUICK GREATEST INTEGER FUNCTION. LEAVE INT(FAC)IN FAC MANTISSA
                                                (HO"MO"LO SIGNED). ASUMES FAC<2"23 (RESET Y-REG=0)
(R0-R15)    (0"31) [$0000"$001F] \PB\           'SWEET-16' REGISTERS R0 THRU R15 OF 'SWEET-16' (16-BIT INTERPRETER IN
                                                MONITOR)
R0L"R0H    (0"1) [$0000"$0001] \P2\             'SWEET-16' REGISTER R0 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R1)    (2"3) [$0002"$0003] \P2\                'SWEET-16' REGISTER R1 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R10)    (20"21) [$0014"$0015] \P2\             'SWEET-16' REGISTER R10 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R11)    (22"23) [$0016"$0017] \P2\             'SWEET-16' REGISTER R11 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R12)    (24"25) [$0018"$0019] \P2\             'SWEET-16' REGISTER R12 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R13)    (26"27) [$001A"$001B] \P2\             'SWEET-16' REGISTER R13 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R14)    (28"29) [$001C"$001D] \P2\             'SWEET-16' REGISTER R14 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
R15L"R15H    (30"31) [$001E"$001F] \P2\         'SWEET-16' REGISTER R15 (USED AS PROGRAM COUNTER IN 16-BIT PSEUDOMACHINE
                                                IN APPLE SYSTEM MONITOR) (REG-R15 )
(R2)    (4"5) [$0004"$0005] \P2\                'SWEET-16' REGISTER R2 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R3)    (6"7) [$0006"$0007] \P2\                'SWEET-16' REGISTER R3 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R4)    (8"9) [$0008"$0009] \P2\                'SWEET-16' REGISTER R4 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R5)    (10"11) [$000A"$000B] \P2\              'SWEET-16' REGISTER R5 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R6)    (12"13) [$000C"$000D] \P2\              'SWEET-16' REGISTER R6 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R7)    (14"15) [$000E"$000F] \P2\              'SWEET-16' REGISTER R7 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
(R8)    (16"17) [$0010"$0011] \P2\              'SWEET-16' REGISTER R8 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
-------------------------------------------------------------------------------------------------------------------------------
PRNTYX - (R8)                    Prof. Luebbert's "What's Where in the Apple"                    ALPHABETICAL GAZETTEER
```

```
(R9)    (18~19) [$0012~$0013] \P2\    'SWEET-16' REGISTER R9 (IN 16-BIT PSEUDOMACHINE IN APPLE SYSTEM MONITOR)
RD2     (-246) [$FF0A]                MONITOR MEMORY LOCATION 'RD2'
RD2BIT    (-774) [$FCFA]              MONITOR TWO-EDGE TAPE SENSE; I.E. LOOPS DECREMENTING Y-REG UNTIL HARDWARE HAS
                                      INDICATED TWO TRANSITIONS OF TAPE INPUT REGISTER.  CONTENTS OF Y-REG ON RETURN
                                      COMPARED WITH CONTENTS ON ENTRY MEASURE TIME REQUIRED FOR TRANSITIONS. CALLS RDBIT
RD3     (-234) [$FF16]                MONITOR MEMORY LOCATION 'RD3'
RDADR16   [B944]                      DOS 3.3 SYNONYM FOR READADR
RDBIT     (-771) [$FCFD]              MONITOR - LOOPS DECREMENTING Y-REG UNTIL CASSETTE TAPE INPUT REGISTER CHANGES
                                      (EITHER 0=>1 OR 1=>0). BIT VALUE RETURNED IS DETERMINED FROM RESIDUAL COUNT OF
                                      Y-REG.  CALLED BY RD2BIT AND READ
RDBYT2    (-786) [$FCEE]              MONITOR MEMORY LOCATION 'RDBYT2'
RDBYTE    (-788) [$FCEC]              MONITOR - READS BITS FROM CASSETTE TAPE UNTIL BYTE ACCUMULATED (CALLED BY MONITOR
                                      READ MEMORY LOCATION 'RDBYTE' SHAPE TABLE LOAD)
RDCHAR    (-715) [$FD35] \SE\         CALLS RDKEY TO GET NEXT CHAR PLACED INTO A-REG. IF ESCAPE KEY PRESSED CALLS 'ESC1'
                                      FOR ESCAPE KEY PROCESSING; AFTER ESCAPE KEY AND KEY FOLLOWING HAVE BEEN READ &
                                      PROCESSED CONTROL RETURNS TO RDCHAR ROUTINE AS IF IT WERE JUST BEING ENTERED (A- X-
                                      Y-REGS ALTERED)
RDKEY     (-756) [$FD0C] \SE\         SAME AS RDCHAR EXCEPT BYPASSES ESCAPE KEY MONITOR SUPPORT; PICKS UP AND SAVE THE
                                      CHARACTER IN THE SCREEN AREA AT BASL~H CH (LEAVING Y-REG CONTAINING CONTENTS OF CH)
                                      IT THEN CHANGES THAT CHARACTER TO BLINKING TO INDICATE CURRENT CURSOR POSN; ASKS
                                      FOR NEXT INPUT CHAR TO BE PLACED IN A-REG BY DOING AN INDIRECT JUMP VIA KSWL~H
                                      WHICH IS NORMALLY POINTING AT KEYIN. RETURN IS THEREFORE TO THE CALLER OF RDKEY -
                                      NOT TO RDKEY ROUTINE ITSELF. SET-UP: A- X- Y-REGS NOT SIGNIFICANT; CV AND BASL~H
                                      SHOULD BE COMPATABLE POINTING IN THE SCROLL WINDOW; CH INDICATES HORIZONTAL
                                      POSITION WHERE CURSOR WILL BLINK. RESULTS: A-REG CONTAINS THE INPUT CHARACTER
                                      (WHICH MAY BE ANY CHARACTER INCLUDING ANY CONTROL KEY OR ESCAPE KEY);X-REG IS
                                      UNCHANGED; Y-REG CONTAINS CONTENTS OF CH; CV CH BASL~H REMAIN UNCHANGED (A- X-
                                      Y-REGS ALTERED)
~RDKEY~   (-3247) [$F351] \SE\        INTEGER BASIC ENTRY TO ROUTINE TO READ AN INPUT FOR BASIC FROM KEYBOARD
RDRIGHT   (15815) [$3DC7] \SL\        DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL WHICH STARTS CODE TO
                                      DETERMINE IF ONE IS READING CORRECT TRACK SECTOR AND VOLUME
RDRIGHT   (-16964~-16916) [$BDBC~$BDEC] \SB\DOS 3.3 - INITIALIZE MAX RETRIES AT 48. READ ADDRESS FIELD VIA 'RDADR16'
                                      ($B944). IF GOOD READ BRANCH TO 'RDRIGHT' ($BDED). IF BAD TRY AGAIN
                                      DECREMENTING RETRIES. IF NONE LEFT PREPARE TO RECALIBRATE. DECREMENT RECAL
                                      COUNT. IF NO MORE THEN 'DRVERR' ($BE04). OTHERWISE RESET RESEEKS AT 4 AND
                                      RECALIBRATE ARM. TRY AGAIN
RDRIGHT   (-16915~-16893) [$BDED~$BE03] \SE\DOS 3.3 - VERIFY TRACK. IF CORRECT BRANCH TO 'RTTRK' ($BE10) OTHERWISE GOTO
                                      'SETTRK' ($BE95) AND DECREMENT RESEEK COUNT. IF ZERO RECAL OTHERWISE RESEEK TRACK
RDSP1     (-1308) [$FAE4]             MONITOR MEMORY LOCATION 'RDSP1'
READ      (-18179~-18076) [$B8FD~$B964] \SB\DOS 3.1~3.2~3.2.1 (SEE $B8DC FOR DOS 3.3 'READ') RWTS (READ-WRITE TRACK-SECTOR
                                      READ MODJLE. READS A SECTOR OFF THE DISK FORMING 410 ($19A) 5-BIT
                                      RIGHT-JUSTIFIED NIBBLES
READ      (-259) [$FEFD] \SE\         READS DATA FROM CASSETTE TAPE PUTTING FIRST DATA READ INTO LOCATION POINTED TO BY
                                      A1L~H ($003C~$003D) AND CONTINUING TO READ UNTIL DATA GOES TO LOCATION POINTED TO
                                      BY A2L~H ($003E~$003F). ALSO COMPUTES A RUNNING EXCLUSIVE OR CHECKSUM IN 'CHECKSUM'
                                      ($002E)
READ16    (-18212~-18109) [$B8DC~$B943] \SB\DOS 3.3 'READ' IN RWTS (READ-WRITE TRACK-SECTOR). READS A SECTOR OFF THE
                                      DISK INTO SECONDARY BUFFER ($BC00-$BC55) HIGH TO LOW THEN INTO PRIMARY
                                      ($BB00-$BBFF) LOW TO HIGH EN ROUTE TO OVERALL PROCESS OF  FORMING $153
                                      RIGHT-JUSTIFIED 6-BIT NIBBLES
```

READADR (DOS 3.2) (-18075~-17984) [$B965~$B9C0] \SB\DOS 3.1~3.2~3.2.1 (SEE $B944 FOR DOS 3.3 'READADR (DOS 3.2)')
RWTS (READ-WRITE TRACK SECTOR) READ ADDRESS MODULE. READS ADDRESSES ON THE SECTORS OF CURRENT TRACK UNTIL IT FINDS A SECTOR. THEN IT RETURNS PUTTING CHECKSUM INTO $002C;SECTOR INTO $002D;TRACK INTO $002E; AND VOLUME INTO $002F. CARRY IS SET ON ERROR

READADR-RDADR16 (DOS 3.3) [.3)$B944~$B99F] \SB\DOS 3.3 READADR. FUNCTION SAME AS READADR-RDADR16 (DOS 3.2)

READX1 (-254) [$FF02]
HI-RES GRAPHICS - READ WITHOUT HEADER

REASON (-11293) [$D3E3] \SE\
CHECKS FOR ENOUGH ROOM IN MEMORY; CHECKS THAT ADDR Y-REG(MSB)&A-REG(LSB) LESS THAN FRETOP. MAY CAUSE GARBAGE COLLECTION. CAUSE OMERR IF NO ROOM

REGDSP (-1321) [$FAD7] \SE\
DISPLAY SAVED REGISTER CONTENTS FROM MEMORY LOCNS $0045~$0049 WITH PRECEDING CARRIAGE RETURN (SEE 'SAVE' ROUTINE AT $FF4A) (A- X-REGS ALTERED)

REGZ (-321) [$FEBF] \SE\
MONITOR S/R TO DISPLAY REGISTERS

REL (-2816) [$F500]
MINIASSEMBLER MEMORY LOCATION 'REL'

REL2 (-2804) [$F50C]
MINIASSEMBLER MEMORY LOCATION 'REL2'

REL3 (-2794) [$F516]
MINIASSEMBLER MEMORY LOCATION 'REL3'

RELADR (-1736) [$F938]
MONITOR MEMORY LOCATION 'RELADR' (RELATIVE ADDRESS)

REMN (-9818) [$D9A6] \SE\
APPLESOFT - CALCULATE OFFSET IN Y-REG FROM TXTPTR TO NEXT COL(0)

REMSTK (248) [$00F8] \P1\
APPLESOFT STACK POINTER SAVED BEFORE EACH STATEMENT

(RESET) (-6066) [$E84E] \SE\
RESET FACEXP($009D) AND $00A2 (FACSIGN) & A-REG TO ZERO (A-REG=>0;X-Y-REG NOT ALTERED)

RESET (-1438) [$FA62]
AUTOSTART MONITOR MEMORY LOCATION 'RESET'

RESET (-167) [$FF59] \SE\
CALL HERE HAS SAME EFFECT AS PUSHING RESET BUTTON

RESETZ (-2670) [$F592]
MINIASSEMBLER MEMORY LOCATION 'RESETZ'

RESTOR (-10167) [$D849] \SE\
APPLESOFT RESTORE FUNCTION - SET DATA POINTER (DATPTR) TO BEGINNING OF THE PROGRAM

RESTORE (-193) [$FF3F] \SE\
RESTORE 6502 REGISTERS: ($0045)=>A-Reg; ($0046)=>X-Reg; ($0047)=>Y-Reg; ($0048)=>P-Reg; (A- X- Y- P-REGS ALTERED)

RESTR1 (-188) [$FF44]
MONITOR MEMORY LOCATION 'RESTR1'

RESUME (-3305) [$F317] \SE\
APPLESOFT ERROR PROC - RESTORE CURLIN FROM ERRLIN & TXTPTR FROM ERRPOS. TRANSFER ERRSTK INTO 6502 STACK POINTER

(RET W/O GOSUB) (-9863) [$D979]
APPLESOFT - PRINT "RETURN WITHOUT GOSUB" THEN HALT AT APPLESOFT (]) LEVEL

~RETURN~ (-5979) [$E8A5] \SE\
INTEGER BASIC ENTRY TO ROUTINE FOR RETURN FROM GOSUB

RGDSP1 (-1318) [$FADA] \SE\
DISPLAY SAVED REGISTER CONTENTS FROM MEMORY LOCNS $0045~$0049 WITHOUT PRECEDING CARRIAGE RETURN (SEE 'SAVE' ROUTINE AT $FF4A) (A- X-REGS ALTERED)

RGTIM (16094) [$3EDE] \SL\
DOS 3.2 DISK FORMATTER INTERIOR LABEL 'RGTIM'

RND (201~205) [$00C9~$00CD] \P5\
APPLESOFT FLOATING POINT RANDOM NUMBER (5-BYTE FLOATING POINT PACKED FORMAT C9=EXP CA-CD=MANTISSA)

RND (-4178) [$EFAE] \SE\
APPLESOFT FP - FORM A 'RANDOM' NUMBER IN FAC USING ORIGINAL VALUE IN FAC AS PARAMETER 'KEY' OR 'SEED'. MODIFIES MANY FP LOCNS

RNDL~RNDH (78~79) [$0C4E~$004F] \P2\
16 BIT NO. RANDOMIZED WITH EACH KEY ENTRY DONE BY MONITOR KEYIN ROUTINE (AND BY MANY OTHER ROUTINES SUCH AS SERIAL & COMM CARD WHICH ARE USED TO REPLACE KEYIN). RANDOMIZATION ACCOMPLISHED BY CONTINUOUSLY INCREMENTING WHILE AWAITING KEYBOARD INPUT. HIGH ORDER BYTE $4F

~RND~ (-4274) [$EF4E] \SE\
INTEGER BASIC ENTRY TO RANDOM NUMBER GENERATOR

RNGERR (-4504) [$EE68] \P1\
INTEGER BASIC RANGE ERROR

RTAR (-2947) [$F47D] \SE\
DENORMALIZE FP1 BY SHIFTING M1(&E) RIGHT 1 BIT POSN & INCREMENTING X1 (A-X-REGS ALTERED)

RTBL (-1255) [$FB19]
MONITOR MEMORY LOCATION 'RTBL'

RTMASK (-2036) [$F80C]
MONITOR MEMORY LOCATION 'RTMASK'

```
RTMSKZ    (-1921) [$F87F]              MONITOR MEMORY LOCATION 'RTMSKZ'
RTNJMP    (-1327) [$FAD1]              MONITOR MEMORY LOCATION 'RTNJMP'
RTNL~RTNH    (44~45) [$002C~$002D] \P2\MONITOR RETURN POINTER (POINTS TO SAVE AREA USED BY INSTRUCTION TRACE ROUTINE)
RTS1    (-1999) [$F831]               MONITOR MEMORY LOCATION 'RTS1'
RTS2    (-1695) [$F961]               MONITOR MEMORY LOCATION 'RTS2'
RTS2B    (-1041) [$FBEF]              MONITOR MEMORY LOCATION 'RTS2B'
RTS2D    (-1234) [$FB2E]              MONITOR MEMORY LOCATION 'RTS2D'
RTS3    (-1028) [$FBFC]               MONITOR MEMORY LOCATION 'RTS3'
RTS4    (-981) [$FC2B]                MONITOR MEMORY LOCATION 'RTS4'
RTS4B    (-824) [$FCC8]               MONITOR MEMORY LOCATION 'RTS4B'
RTS4C    (-571) [$FDC5]               MONITOR MEMORY LOCATION 'RTS4C'
RTS5    (-489) [$FE17]                MONITOR MEMORY LOCATION 'RTS5'
RTTRK    (15856) [$3DF0] \SL\         DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL WHICH ASSUMES RIGHT TRACK
                                      SELECTED AND BEGINS CHECK OF CORRECT VOLUME NUMBER ON DISKETTE
RTTRK    (-16880~-16859) [$BE10~$BE25] DOS 3.3 -CHECK VOL# FOUND VS VOL# WANTED. IF NO VOL SPECIFIED NO ERROR OTHERWISE
                                      IF MISMATCH LOAD A-REG WITH $20 (VOLUME MISMATCH ERROR) AND EXIT VIA 'HNDLERR'
                                      ($BE48)
~RUN #N~    (-4110) [$EFF2] \SE\      INTEGER BASIC ENTRY TO ROUTINE TO RUN FROM LINE #N
RUN    (-10906) [$D566] \SE\          APPLESOFT - RUN THE PROGRAM IN MEMORY. THIS ROUTINE DOES NOT RETURN
RUNMODE    (217) [$00D9] \P1\         INTEGER BASIC MEMORY LOCATION 'RUNMODE' USED AS RUN MODE FLAG BYTE
RUNMODE    (217) [$00D9] \P1\         USED BY DOS TO TEST FOR DIRECT-DEFERRED MODE USAGE. IF $AAB6 CONTAINS 0 AND BIT 7
                                      OF THIS LOCATION IS CLEAR DOS ASSUMES DIRECT MODE AND WILL NOT DO OPEN OR OTHER
                                      DIRECT MODE COMMANDS
~RUN~    (-4116) [$EFEC] \SE\         APPLE INTEGER BASIC RUN ROUTINE (RUN FROM BEGINNING)
RWTS    (15616) [$3D00] \SE\          DOS 3.1/3.2 READ\WRITE A TRACK & SECTOR. UPON ENTRY A- & Y-REGS POINT AT I/O
                                      CONTROL BLOCK (IOB)
RWTS    (15616~16027) [$3D00~$3E9B] \SB\DOS 3.1/3.2 RWTS SUBROUTINE
S16PAG    (247) [$00F7] \P\           SWEET-16 MEMORY LOCATION 'S16PAG'
SAMESLOT    (15661) [$3D2D] \SL\      DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - STARTS CODE TO DETERMINE IF
                                      SAME SLOT BEING USED
SAMESLOT    (-17100~-17069) [$BD34~$BD53] \SB\ENTER READ MODE AND READ WITH DELAYS TO SEE IF DISK IS SPINNING. SAVE
                                      RESULTS OF TEST AND TURN ON MOTOR ANYHOW
SAV1    (-180) [$FF4C]                MONITOR MEMORY LOCATION 'SAV1'
SAVE    (-10064) [$D8B0] \SE\         APPLESOFT CASSETTE - SAVE THE PROGRAM IN MEMORY TO CASSETTE TAPE
SAVE    (-182) [$FF4A] \SE\           MONITOR S/R TO SAVE 6502 REGISTERS: (A-REG)=>$0045; (X-REG)=>$0046;
                                      (Y-REG)=>$0047; (P-REG)=>$0048; (S-REG)=>$0049 (NONE)
~SAVE~    (-3776) [$F140] \SE\        INTEGER BASIC ENTRY TO ROUTINE TO SAVE A PROGRAM TO CASSETTE TAPE
SB    (6912~7423) [$1B00~$1CFF]       TEMPORARY LOCATION OF DOS 3.2 RELOCATION CODE DURING DOS 3.2 BOOT (SB)
SCALE    (231) [$00E7] \P1\           HI-RES GRAPHICS SCALE FACTOR
SCALE    (807) [$0327] \P1\           ON-THE-FLY SCALE FACTOR FOR DRAW~ SHAPE~ MOVE
~SCRATCH~    (-4096) [$F000] \SE\     INTEGER BASIC ENTRY TO SCRATCH EVERYTHING ROUTINE
SCRL1    (-906) [$FC76]               MONITOR MEMORY LOCATION 'SCRL1'
SCRL2    (-884) [$FC8C]               MONITOR MEMORY LOCATION 'SCRL2'
SCRL3    (-875) [$FC95]               MONITOR - CLEAR LINE (BASL~H) (WHOLE LINE) THEN SET NEW BASL~H FROM CV & WNDLFT
SCRN    (-1935) [$F871] \SE\          GET (LOAD TO A-REG) LO-RES GRAPHICS COLOR OF POINT Y-COORD = (A-REG); X-COORD =
                                      (X-REG)  (A-REG ALTERED)
SCRN2    (-1927) [$F879]              MONITOR MEMORY LOCATION 'SCRN2'
~SCRN~    (-7542) [$E28A] \SE\        INTEGER BASIC ENTRY POINT TO SCREEN X~ Y~ COLOR VALUE FUNCTION
SCROLL    (-912) [$FC70] \SE\         MONITOR S/R TO SCROLL UP 1 LINE.  (A- Y-REGS ALTERED)
SCRTCH    (-10677) [$D64B] \SE\       APPLESOFT INITIALIZATION - THE 'NEW' COMMAND. CLEARS PROGRAM VARIABLES & STACK
SECT    (45) [$002D] \P1\             DOS RWTS (READ-WRITE TRACK-SECTOR) PARAMETER FOR CURRENT DISK SECTOR
```

SEEK (15948) [$3E4C] \SE\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT SOFT ENTRY POINT OF SEEK SUBROUTINE

SEEKABS (DOS 3.2) (-18016) [$B9A0] \SB\DOS 3.2 'SEEKABS'

SEEKABS (DOS 3.3) (-18016~-17924) [$B9A0~$B9FC] \SB\DOS 3.3 - MOVES DISK AREM TO DESIRED TRACK. CALLS ARM MOVE DELAY SUBROUTINE ($B9FD). ON ENTRY $0478 CONTAINS CURRENT TRACK; X-REG CONTAINS SLOT*16; A-REG DESIRED TRACK. ON EXIT X-REG UNCHANGED; A-REG Y-REG CLOBBERED; $0478 &$002A: FINAL TRACK;$27 PRIOR TRACK (IF SEEK NEEDED). USES $0026;$0027;$002A;$002B. EXITS TO CALLER

SEEKABS (-17890~-17777) [$BA1E~$BA8F] \SB\ DOS 3.1~3.2~3.2.1 (SEE $B9A0 FOR DOS 3.3) RWTS (READ-WRITE TRACK SECTOR) SEEKABS MODULE. MOVES HEAD TO TRACK SPECIFIED BY A-REG. $0478 IS CURRENT. RWTS DOES PHASE OFF FOR ALL FOUR BEFORE CALL

SEEKCNT (1275) [$04FB] \P1\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) SEEK COUNTER PARAMETER

SETAN0 (-16296) [$C058] \FF\ VALUE<>0 WHEN GAME AN0 IS SET. POKE 0 TO CLEAR GAME I/O OUTPUT AN0 (3.5V AT PIN 15)

SETAN1 (-16294) [$C05A] \FF\ POKE 0 TO CLEAR GAME I/O OUTPUT AN1 (3.5V AT PIN 14)

SETAN2 (-16292) [$C05C] \FF\ POKE 0 TO CLEAR GAME I/O OUTPUT AN2 (3.5V AT PIN 13)

SETAN3 (-16290) [$C05E] \FF\ POKE 0 TO CLEAR GAME I/O OUTPUT AN3 (3.5V AT PIN 12)

~SETBUF~ (-3796) [$F12C] \SE\ INTEGER BASIC ENTRY TO ROUTINE TO SET UP PROGRAM SAVE/LOAD PARAMETERS

SETCOL (-1948) [$F864] \SE\ SET LO-RES COLOR TO COLOR CODE SPECIFIED BY A-REG FOR FUTURE PLOTTING (A-REG ALTERED)

SETGR (-1216) [$FB40] \SE\ MONITOR S/R- SET GRAPHIC MODE (GR). THIS INCLUDES SETTING TO MIXED MODE;CLEARING GRAPHICS PART OF SCREEN; AND RESETTING WNDTOP~WNDLFT~WNDWDTH~WNDBTM & TABV (A-REG ALTERED)

SETHCOL (-2324) [$F6EC] \SE\ APPLESOFT HI-RES - SET COLOR TO CONTENTS OF X-REG (MUST BE LESS THAN 8)

~SETHDR~ (-3810) [$F11E] \SE\ INTEGER BASIC ENTRY TO SET UP HEADER FOR SAVE/LOAD PARAMETERS

SETHRL (-12288) [$D000] \SE\ HI-RES GRAPHICS INIT S/R CALL (ROM VERSION)

SETIFLG (-378) [$FE86] MONITOR MEMORY LOCATION 'SETIFLG'

SETINV (-384) [$FE80] \SE\ MONITOR S/R TO SET VIDEO OUTPUT TO INVERSE

SETKBD (-375) [$FE89] MONITOR MEMORY LOCATION 'SETKBD'

SETMDZ (-483) [$FE1D] MONITOR MEMORY LOCATION 'SETMDZ'

SETMODE (-488) [$FE18] MONITOR MEMORY LOCATION 'SETMODE'

SETNORM (-380) [$FE84] \SE\ MONITOR S/R TO SET VIDEO OUTPUT TO NORMAL (NOT INVERSE)

SETPG3 (-1367) [$FAA9] AUTOSTART MONITOR MEMORY LOCATION 'SETPG3'

~SETPRMPT~ (-8186) [$E006] \SE\ INTEGER BASIC ENTRY POINT TO SET UP '>' PROMPT

SETPWRC (-1169) [$FB6F] \SE\ SET POWER CONDITION (AUTOSTART ROM ONLY)

SETTRK (16002) [$3E82] \SM\ DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL - CODE SETS THE SLOT-DEPENDENT TRACK LOCATION

SETTRK (-16747~-16722) [$BE95~$BEAE] DOS 3.3 - SET TRACK #

SETTRK2 (16015) [$3E8F] \SM\ DOS 3.2 RWTS (READ-WRITE INTERIOR LABEL 'SETTRK2'

SETTXT (-1223) [$FB39] \SE\ MONITOR S/R- SET SCREEN TO TEXT MODE. CLOBBERS ACCUMULATOR (A-REG ALTERED)

SETVID (-365) [$FE93] MONITOR MEMORY LOCATION'SETVID'

SETWND (-1205) [$FB4B] \SE\ MONITOR S/R- SET NORMAL LOW-RESOLUTION GRAPHICS WINDOW

SGN (FPSGN) (-5232) [$EB90] \SE\ APPLESOFT FP - CALLS SIGN AND FLOATS THE RESULT IN THE FAC. FAC=+1 IF FAC WAS +;=0 IF FAC WAS 0;=-1 IF FAC WAS -

~SGN~ (-6308) [$E75C] \SE\ INTEGER BASIC ENTRY POINT TO GET SIGN OF A NUMBER

SHAPEL~SHAPEH (26~27) [$001A~$001B] \P2\ HI-RES POINTER TO SHAPE LIST (ON-THE-FLY SHAPE POINTER)

SHAPEX (81) [$0051] \P1\ HI-RES GRAPHICS SHAPE TEMP.

SHAPXL~SHAPXH (808~809) [$0328~$0329] \P2\ START-OF-SHAPE-TABLE POINTER

SHLOAD (-11335) [$D3B9] \SE\ HI-RES GRAPHICS SHLOAD S/R CALL

SHLOAD (-2187) [$F775] \SE\ APPLESOFT HI-RES. LOADS SHAPE TABLE INTO MEMORY FROM TAPE ABOVE MEMSIZ (HIMEM) AND SETS POINTER AT $00E8

SIGN (47) [$002F] \P1\	$01 BIT SET AFTER CALL TO MULPM OR DIVPM (SIGNED 16 BIT MULT OR DIV) TO SPECIFY WHETHER COMPLEMENT NEEDED (NOTE MULPM & DIVPM IN OLD MONITOR ONLY ~ NOT IN AUTOSTART)
SIGN (243) [$00F3] \P1\	MONITOR & FLOATING POINT ROUTINES MEMORY LOC 'SIGN'
SIGN (-5246) [$EB82] \SE\	APPLESOFT FP - SETS A-REG ACCORDING TO VALUE OF FAC. ON EXIT A-REG=1 IF FAC +;A-REG=0 IF FAC=0;A-REG=$FF IF FAC - (X- Y-REGS NOT ALTERED)
SIN (-4111) [$EFF1] \SE\	APPLESOFT FP - COMPUTE THE SINE OF THE NUMBER IN FAC. RESULT TO FAC. MODIFIES INDEX CHARAC COMPTRTYP XORFPSGN & MANY OTHER FP LOCNS
SLOOP (-1351) [$FAB9]	AUTOSTART MONITOR MEMORY LOCATION 'SLOOP'
(SLOT #) (2040) [$07F8]	CONTAINS SLOT NUMBER (IN THE FORMAT $CS) OF THE PERIPHERAL CARD CURRENTLY ACTIVE - PRINT PEEK(2040)-192 YIELDS SLOT # IN DECIMAL FORMAT
SLOT (1528+S) [$C5F8+S] \P1\	DOS READ-WRITE-TRACK-SECTOR (RWTS) 'SLOT' = HOLDS SLOT NUMBER USED
SNGFLT (-7423) [$E301] \SE\	APPLESOFT - FLOAT THE UNSIGNED INTEGER IN Y-REG INTO FAC. RESETS VALTYP. (RESET Y-REG=0)
SOFTEV (1010~1011) [$03F2~$03F3] \P2\	AUTOSTART ROM RESET VECTOR USED FOR SOFT ENTRY TO LANGUAGE IN USE - DEFAULT VALUE $E003 FOR APPLESOFT
SPACE (-2631) [$F5B9]	MINIASSEMBLER MEMORY LOCATION 'SPACE'
SPDBYT (241) [$00F1] \P1\	USED FOR SPEED CONTROL OF OUTPUT & DISPLAY. SPEED 0-255 ($00-$FF) CONTROLS INSERTED DELAY)
SPKR (-16336) [$C030] \H1\	PEEK TO TOGGLE SPEAKER (PRODUCES A 'CLICK')
SPKR (-16336~-16321) [$C030~$C03F] \H1\	SPEAKER TOGGLE FLIP FLOP. READ ONLY - DO NOT WRITE TO THES ADDRESSES WHICH ARE DECODED AS SAME SINGLE BIT LOCN
SPNT (73) [$0049] \P1\	USER STACK POINTER (S-REGISTER) SAVED HERE BY MONITOR 'SAVE' ROUTINE ON BRK & DURING TRACE
SQR (FPSQR) (-4467) [$EE8D] \SE\	APPLESOFT FP - TAKE SQUARE ROOT OF FAC. RESULT TO FAC. MODIFIES CHARAC INDEX AND MANY OTHER FP LOCNS
(SQR(.5)) (-5843~-5839) [$E92D~$E931] \P5\	APPLESOFT FP CONSTANT SQR(.5) = .707..
(SQR(2)) (-5838~-5834) [$E932~$E936] \P5\	APPLESOFT FP CONSTANT SQR(2) = 1.414...
SRCH2L~SRCH2H (210~211) [$00D2~$00D3] \P2\	INTEGER BASIC MEMORY LOCATION 'SRCH2L' (SECOND VARIABLE SEARCH POINTER)
SRCHL~SRCHH (208~209) [$00D0~$00D1] \P2\	INTEGER BASIC MEMORY LOCATION 'SRCHL' (POINTER TO SEARCH VARIABLE TABLE)
STAT (2040+S) [$07F8+S] \P1\	APPLE COMMUNICATIONS INTERFACE CARD IN SLOT #S - STATUS (SEE ACIC MANUAL PG 17). E.G. POKE 2040+S~17
STATUS (72) [$0048] \P1\	USER STATUS REGISTER (P-REGISTER) SAVED HERE ON BRK TO MONITOR & DURING TRACE. WARNING: INITIALIZE BEFORE G FUNCTION TO AVOID DECIMAL MODE IF DOS HAS BEEN USED
STATUS (1400+S) [$0578+S] \P1\	EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S: PARITY CHECKSUM OPTIONS (SEE MANUAL)
STBITS (1272+S) [$04F8+S] \P1\	EXAMPLE: APPLE SERIAL INTERFACE IN SLOT #S: CONTAIN NUMBER OF STOP BITS (INCLUDING 1 PARITY BIT)
STEP (-1469) [$FA43]	MONITOR S/R- PERFORM A SINGLE STEP (NOT AVAILABLE WITH AUTOSTART ROM). EXECUTES ONE INSTRUCTION AT (PCL~H) WITH REGISTER RESTORE BEFORE; REGISTER SAVE AFTER; UPDATE OF PCL~H; DISPLAY OF INSTRUCTION & DISPLAY OF RESULT REGISTERS
STEPZ (-316) [$FEC4]	MONITOR MEMORY LOCATION 'STEPZ'
~STEP~ (-3463) [$F279] \SE\	INTEGER BASIC ENTRY TO ROUTINE TO HANDLE STEP FUNCTION FOR FOR/NEXT LOOP
STILLON (15646) [$3D1E] \SL\	DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL STARTS CODE WHICH SENSES IF MOTOR STILL ON
STITLE (-1179) [$FB65]	AUTOSTART MONITOR MEMORY LOCATION 'STITLE'
STKINI (-10621) [$D683] \SE\	APPLESOFT STACK INITIALIZATION - CLEARS THE STACK
STOADV (-1040) [$FBF0] \SE\	MONITOR - LOAD Y FROM CH; STORE A-REG TO SCREEN AT (BASL)~Y; AND GOTO ADVANCE ($FBF4) (A- Y-REG ALTERED)

```
NAME  (DEC LOCN)  [HEX LOCN] \USE-TYPE\ - DESCRIPTION
-------------------------------------------------------------------------------------------------------------------------------

"STOPPED AT"   (-5949) [$E8C3] \SE\ INTEGER BASIC ENTRY TO ROUTINE TO PRINT 'STOPED AT LINE #'
STOR   (-501) [$FE0B]                MONITOR MEMORY LOCATION 'STOR'
STREND   (109"110) [$006D"$006E] \P2\APPLESOFT STORAGE END POINTER (POINTS TO TOP OF ARRAY STORAGE I.E. TO END OF
                                     NUMERIC STORAGE IN USE)
STRINI   (-7211) [$E3D5] \SE\        APPLESOFT - GET SPACE FOR CREATION OF A STRING & CREATE DISCRIPTOR FOR IT IN
                                     DSCTMP. ON ENTRY A-REG = LEN OF STRING.
STRLIT   (-7193) [$E3E7] \SE\        APPLESOFT - STORE A QUOTE IN ENDCHR AND CHARAC SO THAT STRLT2 WILL STOP ON IT
STRLT2   (-7187) [$E3ED] \SE\        APPLESOFT - BUILD DESCRIPTOR FOR STRING LITERAL WHOSE 1ST CHAR POINTED TO BY Y-REG
                                     (MSB) & X-REG (LSB). PUT INTO TEMPORARY & POINTER TO IT IN FACMO"FACLO.
STRNG1   (171"172) [$00AB"$00AC] \P2\APPLESOFT POINTER TO A STRING USED IN 'MOVINS' STRING UTILITY
STRNG2   (173"174) [$00AD"$00AE] \P2\APPLESOFT POINTER TO A STRING USED IN STRLT2 STRING UTILITY
STROUT   (-9414) [$DB3A] \SE\        APPLESOFT - PRINT STRING POINTED TO BY Y-REG (MSB) & A-REG (LSB). STRING MUST END
                                     WITH A ZERO OR QUOTE
STRPRT   (-9411) [$DB3D] \SE\        APPLESOFT - PRINT A STRING WHOSE DESCRIPTOR IS POINTED TO BY FACMO"FACLO
STRSPA   (-7203) [$E3DD] \SE\        APPLESOFT - JSR TO GETSPA. STORE THE POINTER & LENGTH IN DSCTMP.
STRTXT   (-8575) [$DE81] \SE\        APPLESOFT - SET Y-REG (MSB) & X-REG(LSB) TO TXTPTR + CARRY BIT AND FALL INTO STRLIT
STXTPT   (-10601) [$D697] \SE\       APPLESOFT INITIALIZATION - SET TXTPTR TO BEGINNING OF PROGRAM
SUBFLG   (20) [$0014]                APPLESOFT SUBSCRIPT FLAG: $00= SUBSCRIPTS ALLOWED;$80= SUBSCRIPTS NOT ALLOWED
SUBTBL   (-29) [$FFE3]               'SUBTBL' L.S.B. ADDRESS-1 OF BASCONT SUBROUTINE
SUBTBL   (-29"-23) [$FFE3"$FFE9] \PB\TABLE OF SUBROUTINE ADDRESSES -1 (INDEX PC WITH TBL ITEM FOR S/R ENTRY): (ADDRESS
                                     MSB = $FE; LSB = TABLE ENTRY +1)
"SUBTRACTION"   (-6270) [$E782] \SE\INTEGER BASIC ENTRY POINT TO SUBTRACTION FUNCTION
SYNCHR   (-8512) [$DEC0] \SE\        APPLESOFT SYNTAX CHARACTER CHECK - CHECKS TO VERIFY TXTPTR POINTS TO SAME CHARACTER
                                     AS THAT IN A-REG. NORMAL EXIT THRU CHGET TO GET NEX CHAR FROM INPUT BUFFER OTHERWISE
                                     SYNTAX ERROR. TXTPTR NOT MODIFIED. (Y-REG RESET TO ZERO)
SYNPAGL"SYNPAGH   (254"255) [$00FE"$00FF] INTEGER BASIC SYNTAX PAGE POINTER. IF $00FF NOT ZERO THEN ERROR CONDITION
                                     EXISTS
SYNSTKDX   (253) [$00FD] \P1\        INTEGER BASIC MEMORY LOCATION 'SYNSTKDX' (SYNTAX STACK INDEX VALUE)
SYNSTKH   (88) [$0058]               INTEGER BASIC MEMORY LOCATION 'SYNSTKH'
SYNSTKL   (128"159) [$0080"$009F]    INTEGER BASIC MEMORY LOCATION 'SYNSTKL' (SYNTAX STACK LOCATION)
"SYNTABL"   (-5120"-4609) [$EC00"$EDFF] \PB\INTEGER BASIC SYNTAX TABLE
(TABLE1 DOS 3.2.1)   (-17780) [$BA8C] \SB\DOS 3.2.1 RWTS OPERATION TIMER ROUTINE TABLE1
TABV   (-1189) [$FB5B] \SE\          PLACE CURSOR AT LINE (A-REG) COLUMN (CH) SETTING CV AND BASL"H FROM A-REG
                                     (A-REG ALTERED)
"TAB"   (-6236) [$E7A4] \SE\         INTEGER BASIC ENTRY POINT TO HORIZONTAL TAB FUNCTION
TAN   (-4038) [$F03A] \SE\           APPLESOFT FP - COMPUTE THE TANGENT OF THE NUMBER IN FAC. RESULT TO FAC.
                                     MODIFIES CHARAC INDEX XORFPSGN AND MANY OTHER FP LOCNS
TAPEIN   (-16288) [$C060]            MONITOR MEMORY LOCATION 'TAPEIN'
TAPEIN  [$C060/8] \H1\               STATE OF 'CASSETE DATA IN' APPEARS IN BIT 7
TAPEOUT   (-16352) [$C020] \H1\      PEEK TO TOGGLE CASSETTE OUTPUT (CREATE A 'CLICK' ON RECORDING)
TAPEOUT   (-16352"-16337) [$C020"$C02F] \H1\CASSETTE OUTPUT TOGGLE FLIP FLOP. READ ONLY DO NOT WRITE TO THESE ADDRESSES
                                     WHICH ARE DECODED AS SAME SINGLE BIT LOCN
TEMP   (44"45) [$002C"$002D] \P2\    DOS RWTS (READ-WRITE TRACK-SECTOR TEMPORARY STORAGE FOR ADDRESS INFORMATION
TEMP1   (147"151) [$0093"$0097] \P5\ APPLESOFT REGISTER TEMP1 FOR FLOATING POINT MATH PACKAGE (PACKED 5-BYTE FORMAT)
TEMP2   (152"156) [$0098"$009C] \P5\ APPLESOFT FLOATING POINT MATH PACKAGE REGISTER TEMP2 (PACKED 5-BYTE FORMAT)
TEMP3   (138"142) [$008A"$008E] \P5\ APPLESOFT REGISTER TEMP3 FOR FLOATING POINT MATH PACKAGE (PACKED 5-BYTE FORMAT)
TEMPPT   (82) [$0052] \P1\           APPLESOFT TEMPORARY POINT - LAST USED TEMPORARY STRING DESCRIPTOR (SEE DSCTMP)
(TEXTLN0)   [$0400-$0427] \BB\       VIDEO SCREEN BUFFER TEXT LINE 0
(TEXTLN1)   [$0480-$04A7] \BB\       VIDEO SCREEN BUFFER TEXT LINE 1
(TEXTLN10)   [$0528-$054F] \BB\      VIDEO SCREEN BUFFER TEXT LINE 10
(TEXTLN11)   [$05A8-$05CF] \BB\      VIDEO SCREEN BUFFER TEXT LINE 11
-------------------------------------------------------------------------------------------------------------------------------
STOPPED AT - (TEXTLN                 Prof. Luebbert's "What's Where in the Apple"                ALPHABETICAL GAZETTEER
```

```
(TEXTLN12)  [$0628-$064F] \BB\          VIDEO SCREEN BUFFER TEXT LINE 12
(TEXTLN13)  [$06A8-$06CF] \BB\          VIDEO SCREEN BUFFER TEXT LINE 13
(TEXTLN14)  [$0728-$074F] \BB\          VIDEO SCREEN BUFFER TEXT LINE 14
(TEXTLN15)  [$07A8-$07CF] \BB\          VIDEO SCREEN BUFFER TEXT LINE 15
(TEXTLN16)  [$0450-$0477] \BB\          VIDEO SCREEN BUFFER TEXT LINE 16
(TEXTLN17)  [$04D0-$04F7] \BB\          VIDEO SCREEN BUFFER TEXT LINE 17
(TEXTLN18)  [$0550-$0577] \BB\          VIDEO SCREEN BUFFER TEXT LINE 18
(TEXTLN19)  [$05D0-$05F7] \BB\          VIDEO SCREEN BUFFER TEXT LINE 19
(TEXTLN2)  [$0500-$0527] \BB\           VIDEO SCREEN BUFFER TEXT LINE 2
(TEXTLN20)  [$06=0-$0677] \BB\          VIDEO SCREEN BUFFER TEXT LINE 20
(TEXTLN21)  [$06D0-$06F7] \BB\          VIDEO SCREEN BUFFER TEXT LINE 21
(TEXTLN22)  [$0750-$0777] \BB\          VIDEO SCREEN BUFFER TEXT LINE 22
(TEXTLN23)  [$07D0-$07F7] \BB\          VIDEO SCREEN BUFFER TEXT LINE 23
(TEXTLN3)  [$0580-$05A7] \BB\           VIDEO SCREEN BUFFER TEXT LINE 3
(TEXTLN4)  [$0600-$0627] \BB\           VIDEO SCREEN BUFFER TEXT LINE 4
(TEXTLN5)  [$0680-$06A7] \BB\           VIDEO SCREEN BUFFER TEXT LINE 5
(TEXTLN6)  [$0700-$0727] \BB\           VIDEO SCREEN BUFFER TEXT LINE 6
(TEXTLN7)  [$0780-$07A7] \BB\           VIDEO SCREEN BUFFER TEXT LINE 7
(TEXTLN8)  [$0428-$044F] \BB\           VIDEO SCREEN BUFFER TEXT LINE 8
(TEXTLN9)  [$04A8-$04CF] \BB\           VIDEO SCREEN BUFFER TEXT LINE 9
(TEXTMACROLINE2)   (1280~1399) [$0500~$0577] \HB\TEXTVIDEO DISPLAY - SUBPAGE 2. CONSISTS OF TEXT LINES 2~ 10 & 18
                                        FOLLOWED BY AN 8-BYTE BLOCK FOR I-O PERIPHERALS
TEXTTAB   (103~104) [$0067~$0068] \P2\  APPLESOFT TEXT TABLE POINTER (POINTS TO TO BEGINNING OF PROGRAM TEXT . DEFAULT
                                        VALUE $0801
(TIMER DOS 3.1~3.2)   (-17793) [$BA7F] \SB\DOS 3.1~3.2 RWTS OPERATION TIMER ROUTINE
(TIMER DOS 3.2.1)   (-17797) [$BA7B] \SB\DOS 3.2.1 RWTS OPERATION TIMER ROUTINE
TITLE   (-1271) [$FB09]                 AUTOSTART MONITOR MEMORY LOCATION 'TITLE'
~TO/FOR~   (-5808) [$E950] \SE\         INTEGER BASIC ENTRY POINT TO ROUTINE TO HANDLE LOOP COUNTER # TO # STEP #
TOKNDX   (241) [$00F1] \P1\             INTEGER BASIC MEMORY LOCATION 'TOKNDX' (TOKEN INDEX VALUE)
TOKNDXSTK   (209~240) [$00D1~$00F0]     INTEGER BASIC MEMORY LOCATION 'TOKNDXSTK' ('TOKEN INDEX STACK?)
(TOOCOMPLEX)   (-7120) [$E430] \SE\     APPLESOFT - PRINT "FORMULA TOO COMPLEX" THEN HALT AT APPLESOFT (]) LEVEL
TOSUB   (-66) [$FFBE]                   MONITOR & MINIASSEMBER MEMORY LOCATION 'TOSUB'
TRACE   (-318) [$FEC2] \SE\             CALL TO PERFORM MONITOR TRACE
~TRACEIT~   (-3715) [$F17D] \SE\        INTEGER BASIC ENTRY TO ROUTINE TO EXECUTE THE TRACE FUNCTION
~TRACE~   (-3727) [$F171] \SE\          INTEGER BASIC ENTRY TO ROUTINE TO SET TRACE MODE FOR EXECUTION
TRACK - TRKN   (46) [$002E] \P1\        DOS RWTS (READ-WRITE TRACK-SECTOR) TRACK NUMBER
TRKCNT   (65) [$0041] \P1\              DOS DISK SYSTEM FORMATTER SPECIAL TRACK COUNTER
TRKDON   (16243~16340) [$3F73~$3FD4] \SB\DOS 3.2 DISK FORMATTER CHECK TRACK FORMATTING ROUTINE
TRKDON   (-4237) [$EF73] \SE\           DOS 3.2 DISK FORMATTER INTERIOR LABEL AT POINT WHERE TRACK FORMATTING IS DONE
                                        AND CHECKING OF THAT FORMATTING BEGINS
TRKFRM   (16046) [$3EAE] \SL\           DOS 3.2 DISK FORMATTER LABEL AT POINT WHERE TRACK FORMATTING BEGINS
TRYADR   (15776) [$3DA0] \SL\           DOS 3.2 RWTS (READ-WRITE TRACK SECTOR) INTERIOR LABEL 'TRYADR'
TRYADR2   (15784) [$3DA8] \SL\          DOS 3.2 RWTS (READ-WRITE TRACK SECTOR) INTERIOR LABEL 'TRYADR2
TRYNEXT   (-2724) [$F55C]               MINIASSEMBLER MEMORY LOCATION 'TRYNEXT'
TRYTRK   (15754) [$3D8A] \SL\           DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL - TRY DISK TRACK AS PART
                                        OF LOCATING CORRECT SECTOR FOR READ
TRYTRK   (-16981~-16965) [$BDAB~$BDBB] \SB\DOS 3.3 - GET COMMAND CODE.  IF NULL EXIT VIA 'ALLDONE' ($BE46) TURNING OFF
                                        DRIVE & RETURNING TO CALLER. IF COMMAND CODE=4 BRANCH TO 'FORMDSK' ($BE0D);
                                        OTHERWISE MOVE LOW BIT INTO CARRY (SET=READ;CLEAR=WRITE) AND SAVE VALUE ON
                                        STATUS REG. IF WRITE OPN DATA IS PRENIBBILIZED VIA 'PRENIB16' ($B800)
TRYTRK2   (15771) [$3D9B] \SL\          DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR INTERIOR LABEL 'TRYTRK2'
```

(TWO PI) (-3989~-3985) [$F06B~$F06F] \P5\APPLESOFT 5-BYTE FLOATING POINT CONSTANT 2*PI = 6.2832...
TXTCLR (-16304) [$C050] \H1\ POKE TO 0 TO SET FROM TEXT TO GRAPHICS MODE W/O CLEARING SCREEN
TXTNDX (200) [$00C8] INTEGER BASIC MEMORY LOCATION 'TXTNDX' (TEXT INDEX VALUE)
TXTNDXSTK (168~199) [$00A8~$00C7] INTEGER BASIC MEMORY LOCATION 'TXTNDXSTK' (TEXT INDEX STACK)
TXTPTR (184~185) [$00B8~$00B9] \P2\ TXTPTR - POINTS AT NEXT CHAR OR TOKEN FROM PROG (C/A DEC 78)
TXTSET (-16303) [$C051] \H1\ POKE TO 0 TO SET FROM GRAPHICS TO TEXT MODE W/O RESETTING SCROLLING WINDOW
(UNDEF'D STMT PRT) (-9860) [$D97C] APPLESOFT - PRINT "UNDEF'D STATEMENT" THEN HALT AT APPLESOFT (]) LEVEL
~UNPACK~ (-8083) [$E06D] \SE\ INTEGER BASIC ENTRY POINT TO UNPACK TOKENED CODE TO MNEMONICS
UP ~ CURSUP (-998) [$FC1A] \SE\ MONITOR S/R TO MOVE CURSOR UPWARD (IF POSSIBLE) (A-REG ALTERED)
USR (-310) [$FECA] MONITOR MEMORY LOCATION 'USR'
USRADR (1016) [$03F8] IN MONITOR MODE KEYBOARD ENTRY OF CTL-Y WILL CAUSE JSR HERE
V2 (45) [$002D] \P1\ BOTTOM PT OF LO-RES VERT LINE DRAWN BY VLINE. RANGE: 0-19(~$21) FOR MIXED SCR; 0-23(~$17) FOR FULL SCR
~VALGETL~VALGETH~ (206~207) [$00CE~$00CF] \P2\INTEGER BASIC PRIMARY EVALUATOR TEMPORARY LOCATION
~VALL~VALH~ (206~207) [$00CE~$00CF] \P2\INTEGER BASIC 16-BIT TEMPORARY VALUE FOR MATHEMATICAL OPERATIONS
VALTYP (17) [$0011] APPLESOFT FLAG FOR LAST FAC (FLOATING ACCUMULATOR) OPERATION: $00 = NUMBER; $FF=STRING
VARPNT (131~132) [$0083~$0084] \P2\ APPLESOFT POINTER TO THE LAST-USED VARIABLE'S VALUE (USED BY PTRGET)
VARTAB: (105~106) [$0069~$006A] \P2\ APPLESOFT VARIABLE TABLE POINTER - POINTS TO TO START OF SIMPLE VARIABLE SPACE (AT END OF APPLESOFT PROGRAM TEXT)
VARTIO (-10000) [$D8FC] \SE\ APPLESOFT CASSETTE - SET UP A1 & A2 TO SAVE 3 BYTES ($0050~$0052) FOR LENGTH
~VERBADL~ (-5616~-5497) [$EA10~$EA87] \PB\INTEGER BASIC VERB DISPATCH TABLE LOW BYTE
~VERBADRH~ (-5496) [$EA88] \PB\ INTEGER BASIC VERB DISPATCH TABLE HI BYTE
VERBNOW (214) [$00D6] \P1\ INTEGER BASIC MEMORY LOCATION 'VERBNOW' (VERB CURRENTLY IN USE)
VFY (-458) [$FE36] \SE\ MONITOR S/R TO PERFORM A MEMORY VERIFY (A1-A2 TO A4)
VFYOK (-424) [$FE58] MONITOR MEMORY LOCATION 'VFYOK'
VIDOUT (-1027) [$FBFD] \SE\ MONITOR S/R- OUTPUT A-REGISTER AS ASCII ON TEXT SCREEN OR PROCESS CONTROL CHARACTER. IF (A)<$80 GOTO STOADV; =$87 SOUND BELL; =$88 GOTO BS; =$8A GOTO LF; =$8D GOTO CR; >$9F GOTO STOADV; OTHERWISE IGNORE ENTRY SCREEN RTS 1
VIDWAIT (-1160) [$FB78] AUTOSTART MONITOR MEMORY LOCATION 'VIDWAIT'
VLINE (-2008) [$F828] \SE\ LO-RES PLOT VERT LINE AT X-COORD = (Y-REG) AND Y-COORD FROM (A-REG) THRU ($002D) (A-REG ALTERED)
VLINEZ (-2010) [$F826] \SE\ LO-RES PLOT VERTICAL LINE AT X-COORD = (Y-REG) AND Y-COORD FROM (A-REG)+1+CARRY THRU ($002D) (A-REG ALTERED)
~VLIN~ (-4410) [$EEC6] \SE\ INTEGER BASIC ENTRY POINT TO DRAW A LO-RES VERTICAL LINE
VOLUME (47) [$002F] \P1\ DOS RWTS (READ-WRITE TRACK-SECTOR) DISK VOLUME NUMBER
VTAB (-990) [$FC22] \SE\ PERFORM A VERTICAL TAB TO ROW SPECIFIED IN A-REG ($0~$17). SET BASL~H FROM CV (AND WNDLFT) (A-REG ALTERED)
VTABZ (-988) [$FC24] \SE\ SET BASL~H FROM (A-REG) AND WNDLFT WITHOUT REGARD TO CV (A-REG ALTERED)
~VTAB~ (-4521) [$EE57] \SE\ INTEGER BASIC ENTRY TO VERTICAL TAB FUNCTION
WAIT (-856) [$FCA8] \SE\ CALL FOR WAIT LOOP. WAIT ESTIMATED AT 2.5A~2+13.5A+13 WAIT CYCLES OF 1.02 MICROSECONDS WHERE A IS CONTENTS OF A-REG WHEN S/R CALLED
WAIT2 (-855) [$FCA9] MONITOR MEMORY LOCATION 'WAIT2'
WAIT3 (-854) [$FCAA] MONITOR MEMORY LOCATION 'WAIT3'
WBYTE (16315) [$3FBB] \SL\ DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF TIGHT TIMING ROUTINE
WINBLB2 (16331) [$3FCB] \SL\ DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WINBLB2'
WINBLC (-4147) [$EFCD] \SL\ DOS 3.2 DISK FORMAT INTERIOR LABEL 'WINBLC'
WLOOP (16256) [$3F80] \SL\ DOS 3.2 DISK FORMATTER INTERIOR LABEL AT BEGINNING OF 26 MICROSECOND WAIT LOOP
WNDBTM (35) [$0023] \P1\ BOTTOM LINE OF SCROLL WINDOW: RANGE (WNDTOP)+1 TO 24($18).
WNDLFT (32) [$0020] \P1\ LEFT COLUMN OF SCROLL WINDOW: RANGE 0-39 OR $0~$27. USED ONLY IN VTABZ.
WNDTOP (34) [$0022] \P1\ TOP LINE OF SCROLL WINDOW: RANGE 0-22($16) FOR FULL TEXT SCREEN 20-22($14~$16) FOR MIXED SCREEN

```
WNDWDTH    (33) [$0021] \P1\              WIDTH OF THE SCROLL WINDOW: RANGE:1 TO 40-(WNDLFT) OR $1 TO $28 - (WNDLFT)
WNIBLA   (16330) [$3FCA] \SL\             DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WNIBLA'
WR1    (-300) [$FED4]                     MONITOR MEMORY LOCATION 'WR1'
WRBIT    (-810) [$FCD6]                   MONITOR - WRITES A BIT TO CASSETTE TAPE (CALLED BY WRBYTE AND HEADR)
WRBYT2   (-273) [$FEEF]                   MONITOR MEMORY LOCATION 'WRBYT2'
WRBYTE   (-275) [$FEED]                   MONITOR - USES WRBIT TO WRITE 10 BITS  TO CASSETTE TAPE
WRIT   (15922) [$3E32] \SL\               DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL AT START OF CODE TO WRITE
                                          NIBBLES TO DISK IF NOT WRITE PROTECTED
WRIT   (-16815~-16807) [$BE51~$BE59]      DOS 3.3 - WRITE A SECTOR USING 'WRITE16' ($B82A); IF GOOD WRITE EXIT VIA
                                          'ALLDONE' ($BE46) OTHERWISE LOAD A-REG WITH $10 (WRITE PROTECT ERROR) AND EXIT
                                          VIA 'HNDLERR' ($BE48)
WRIT2   (16098) [$3EE2] \SL\              DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WRIT2'
WRIT3   (16103) [$3EE7] \SL\              DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WRIT3'
WRITE   (-18326~-18180) [$B86A~$B8FC] \SB\DOS 3.1~3.2~3.2.1 (SEE $B82A FOR DOS 3.3 'WRITE') RWTS (READ-WRITE
                                          TRACK-SECTOR) WRITE MODULE. WRITES A BUFFER OF 410 ($19A) 5-BIT
                                          RIGHT-JUSTIFIED NIBBLES ONTO THE DISK SURFACE AS A SECTOR CONVERTING THEM TO
                                          A 8-BIT 'DISK BYTE' FORMAT FIRST
WRITE   (-307) [$FECD] \SE\               MONITOR S/R TO WRITE DATA FROM MEMORY TO CASSETTE TAPE - FIRST MEMORY LOCATION
                                          POINTED TO BY A1L~H ($003C~$003D); LAST BY A2L~H ($003E~$003F).  CASSETTE TAPE
                                          GETS 10 SECONDS OF TONE HEADER THEN THE DESIGNATED DATA BITS AND ONE CHECKSUM
                                          BYTE
WRITE16 (DOS 3.3)   (-18390~-18249) [$B82A~$B8B7] \SB\DOS 3.3 'WRITE'. WRITES PRENIBBILIZED DATA FROM PRIMARY &
                                                SECONDARY BUFFERS TO DISK; CALLS WRITE-A-BYTE S~R; WRITES 5 BYTES
                                                AUTSYNC~ STARTING DATA MARKS ($D5~$AA~$AD)~ 342 BYTES DATA~ ONE
                                                BYTE CHECKSUM~ AND CLOSING DATA MARKS ($DE~$AA~$EB). USES WRITE
                                                TRANSLATE TABLE ($BA29). ON ENTRY X-REG CONTAINS SLOT#*16. ON
                                                EXIT X-REG UNCHANGED;Y-REG $00; CARRY CLEAR. USES $0026~ $0027~ $678
WRITSF   (16102) [$3EE6] \SL\             DOS 3.2 DISK FORMATTER INTERIOR LABEL 'WRITSF'
WRNIBL   (-4146) [$EFCE] \SL\             DOS 3.2 DISK FORMAT INTERIOR LABEL 'WRNIBL'
WRTAPE   (-795) [$FCE5]                   MONITOR MEMORY LOCATION 'WRTAPE'
WRTRK   (16068) [$3EC4] \SL\              DOS 3.2 DISK FORMATTER - LABEL AT POINT WHERE WRITE OF FORMATTING INFO
                                          ONTO TRACK BEGINS -- A HIGHLY TIMING-SENSITIVE AREA OF CODE
XOL~XOH   (800~801) [$0320~$0321] \P2\    HI-RES GRAPHICS- PRIOR X-COORD SAVE AFTER HLIN OR HPLOT
X1   (248) [$00F8]                        OLD (NON-APPLESOFT) FLOATING POINT ROUTINES FLOATING POINT ACCUMULATOR
                                          FP1 MEMORY LOC 'X1' (EXPONENT)
X2   (244) [$00F4] \P1\                   MONITOR & OLD (NON-APPLESOFT) FLOATING POINT ROUTINES FLOATING POINT
                                          ACCUMULATOR 2 MEMORY LOC 'X2' (EXPONENT)
XAM    (-589) [$FDB3] \SE\                MONITOR S/R TO EXAMINE CONTENTS OF MEMORY FROM (A1L~A1H) TO(A2L~A2H).
                                          Y-REG=0 BEFORE CALL (A-REG ALTERED)
XAM8    (-605) [$FDA3] \SE\               MONITOR S/R TO EXAMINE 8 MEM LOCNS. PRINTS HEX OF MEMORY FROM XXXX TO
                                          XXX7 WHERE XXXX IS CONTENTS OF A1L~A1H; Y-REG MUST =0 ON ENTRY (A-REG
                                          ALTERED )
XAMPM    (-570) [$FDC6]                   MONITOR MEMORY LOCATION 'XAMPM'
XBASIC    (-336) [$FEB0] \SE\             MONITOR S/R TO JUMP TO BASIC
XBRK    (-1380) [$FA9C]                   MONITOR MEMORY LOCATION 'XBRK'
XDRAW    (-2467) [$F65D] \SE\             APPLESOFT HI-RES - DRAW SHAPE POINTED TO BY Y-REG(MSB)&X-REG(LSB) BY
                                          INVERTING EXISTING COLOR OF DOTS SHAPE DRAWS OVER. A-REG = ROT FACTOR
XJMP    (-1340) [$FAC4]                   MONITOR MEMORY LOCATION 'XJMP'
XJMPAT    (-1339) [$FAC5]                 MONITOR MEMORY LOCATION 'XJMPAT'
XJSR    (-1351) [$FAB9]                   MONITOR MEMORY LOCATION 'XJSR'
XQ1    (-1416) [$FA78]                    MONITOR MEMORY LOCATION 'XQ1'
```

NAME	(DEC LOCN)	[HEX LOCN] \USE-TYPE\	DESCRIPTION
XQ2	(-1414)	[$FA7A]	MONITOR MEMORY LOCATION 'XQ2'
XQINIT	(-1458)	[$FA4E]	MONITOR MEMORY LOCATION 'XQINIT'
XQT/XQTNZ	(60~67)	[$003C~$0043] \PB\	8 BYTE WORK AREA FOR INSTRUCTION STEP/TRACE. NEXT INSTRUCTION SOMETIMES MOVED HERE
XREG	(70)	[$0046] \P1\	USER X-REG SAVED HERE ON BRK TO MONITOR & DURING TRACE
XRTI	(-1371)	[$FAA5]	MONITOR MEMORY LOCATION 'XRTI'
XRTS	(-1367)	[$FAA9]	MONITOR MEMORY LOCATION 'XRTS'
XSAVE	(216)	[$00D8] \P1\	INTEGER BASIC MEMORY LOCATION 'XSAVE' (TEMPORARY STORAGE FOR CONTENTS OF X-REGISTER)
XTNDL~XTNDH	(82~83)	[$0052~$0053] \P2\	OLD MONITOR (NOT AUTOSTART) - USED IN 16-BIT MULT & DIVIDE AS ACCUMULATOR EXTENSION (TO 32 BITS)
XTOY	(15995)	[$3E7B] \DL\	DOS 3.2 RWTS (READ-WRITE TRACK-SECTOR) INTERIOR LABEL 'XTOY'
XTOY	(-16754~-16748)	[$BE8E~$BE94]	DOS 3.3 - X-REG/16 =>Y-REG. USED TO PUT SLOT INTO Y-REG
YO	(802)	[$0322] \P1\	HI-RES GRAPHICS YO - MOST RECENT Y-COORDINATE
YCNT	(71)	[$0047] \P1\	DOS DISK SYSTEM FORMATTER NYBBLE COUNTER (ALSO COUNTER FOR DISK-DRIVE MOTOR-ON TIME?)
YREG	(71)	[$0047] \P1\	USER Y-REG SAVED HERE ON BRK TO MONITOR & DURING TRACE {Y-REG SAVED HERE ON BRK}
YSAV	(52)	[$0034] \P1\	USED BY MONITOR COMMAND PROCESSOR TO SAVE CONTENTS OF Y-REGISTER DURING PROCESSOR {Y-REGISTER SAVE LOCN FOR MONITOR}
YSAV1	(53)	[$0035] \P1\	USED TO SAVE CONTENTS OF Y-REGISTER ACROSS A CALL TO SCREEN OUTPUT ROUTINES. {Y-REGISTER SAVE LOCN FOR COUT1}
YTEMP	(201)	[$00C9]	INTEGER BASIC MEMORY LOCATION 'YTEMP' (TEMPORARY STORAGE FOR Y-REGISTER)
ZERDLY	(-805)	[$FCDB]	MONITOR MEMORY LOCATION 'ZERDLY'
ZMODE	(-132)	[$FF7C]	MONITOR & MINIASSEMBLER MEMORY LOCATION 'ZMODE'
ZMODE	(-57)	[$FFC7]	MONITOR MEMORY LOCATION 'ZMODE'
ZPGBM3 ZPGFCB	(67~67)	[$0043~$0043]	DOS - USED AS GENERAL PURPOSE POINTER BY SECOND-LEVEL DOS ROUTINES

Appendix A
The Apple //e -- A New Edition
Memory Pages 192-207 and 248-255
($C000-$CFFF and $F800-$FFFF)

A.1
Overview

The latest Apple II, called the "//e" for "enhanced", has several features added that make it more standard and versatile. The keyboard has been improved and will now generate all 128 ASCII key codes, including screen display of lower case. The RESET key now requires pressing the CONTROL key simultaneously and rebooting can be accomplished by pressing CTRL-OPEN APPLE-RESET, saving wear and tear on the on/off switch, always a weak point. A CTRL-CLOSED APPLE RESET initiates a built-in self-test. The screen display has been improved to allow either 40 or 80 column display under software control. There is also a full cursor control in all four directions. The 16K language card has been made a built-in feature and slot 0 has been eliminated. International versions are available for European and Asian buyers with switchable character sets.

Despite all these additional features, compatability was kept with most of the previous software. All of the standard monitor entry points were preserved so that, unless software uses undocumented monitor entries, it should run on the //e. The only other problem that might arise is the utilization of one formerly unused page zero location. A program that used that location will probably not function properly on the new Apple.

Another new feature is the addition of a 64K expansion available as an enhanced 80 column card, which will make additional memory available to sophisticated programs such as Visicalc.

A.2
A Third Apple Monitor

There is now a third major version of the Apple monitor to go along with the Auto-Start and (old) System monitors. While all of the documented entry points remain the same, most of the routines jump to the new ROM in the $C100-$CFFF range. These new routines check on the availability and status of 80 column and extended 80 column cards, and use this additional hardware for enhanced displays and cursor control, when available.

The major differences between the II+ and the //e are as follows:

a) RESET, OPEN APPLE and CLOSED APPLE keys: The Control key must now be pressed to initiate the RESET cycle. This will eliminate accidental RESETs as the keys are on opposite sides of the keyboard. The APPLE keys are paddle button extensions to the keyboard and can be used in conjunction with the RESET cycle to initiate the self diagnostic tests (CLOSED) or power-on reboot (OPEN).

b) EDITING: In addition to the I, J, K, and L diamond cursor control pattern, there are four arrow keys that can also be used to move the cursor on the screen. Pressing ESC to enter the editing mode changes the cursor to an inverse "+" to indicate editing mode. Additional commands are also available. ESC-R enters upper-case restrict mode, which allows only upper-case letters during keyboard entry except after typing a "'", when both upper and lower case are allowed for PRINT statement. Typing another "'" returns to upper-case only. ESC-T exits this mode. ESC-4 displays a 40 column screen similar to the II+, while ESC-8 shifts to the new 80 column screen display. ESC CTRL-Q exits the new made entirely, returning to the old 40 column display, and turning off the 80 column card.

A.3
The New Display

In order to maintain compatability with the old II and II+, it was necessary to design a screen display that utilized the old screen memory ($400-$7FF). This was insufficient for 80 column display, so Apple designed an 80 column card with its own memory mapped into the same addresses. The hardware alternates its scans from one set of memory to the other when in 80 column mode. Characters are stored alternating from one address to the next, with all the odd screen locations in main memory and all the even ones on the auxiliary card.

There are routines in the new monitor areas that can convert an 80 column screen to 40 by moving the alternate characters to the main board and throwing away the last 40 characters in each column. The opposite switch is accomplished by a similar move to the auxiliary card, using only the leftmost 40 columns for the characters previously on the screen.

A.4
Hardware Locations

On the older Apples, the addresses $C000-$C00F were equivalent addresses and were only partly decoded by the hardware. This meant that reading any of those would yield the same result (reading the keyboard), which was also true of $C010-$C01F (clearing the keyboard strobe). These addresses are now fully decoded and provide a set of soft switches/status indicators for the new 80 column card and extended 80 column card (with 64K memory expansion).

The switches include options to read and/or write either the main board locations or the auxiliary card locations, to set the standard zero page and system stack (main board) or the alternate zero page and system stack (auxiliary card), to turn on or off the $CX00 ROMs, to enable or disable the 80 column display, and to turn on the normal or alternate character sets (normal has upper case flash instead of lower case inverse).

Additionally, there are a group of locations that can be read to determine the current switch settings so that any program changing the switches can save the current settings and restore them at the end. States that can be determined include READ/WRITE status, language card bank status, 80 column status, page status, and text mode.

A.5
Software Status

Apple has always reserved some unused locations in the text page RAM as scratch memory for the 7 hardware slots (1-7). Several of these locations are now permanently assigned to the new 80 column cards, when they are in use, and are used to store the current cursor location, I/O status, and BASL/BASH in Pascal.

One particular location ($4FB) is the software MODE status. Each bit is indicative of the current state of operations: BASIC/Pascal, interrupts set/cleared, Pascal 1.0/1.1, normal/inverse video, GOTOXY in progress/not in progress, upper case restrict/literal mode, BASIC input/print, and ESC-R active/inactive.

These locations enable a program to determine the current state of the machine more easily than before, and make it simpler to utilize the new hardware configurations in programming.

A.6
Programming Considerations

The standard Applesoft GET and INPUT (and associated monitor routine KEYIN) were not designed to work with an 80 column display and using them while in 80 column mode can cause loss of data or erasure of program in memory, but this can be overcome by a routine explained in Appendix E of the new Applesoft Tutorial. Reading the keyboard directly ($C000) functions the same as before.

Do not assume an Apple //e or 80 column card when writing programs; one of the first routines should check for the type of machine being used. Apple supplies a program that will do this on "The Applesoft Sampler"; and Call A.P.P.L.E. has also published a routine for this purpose. HTAB will not function beyond the 40th column. While POKE 36,POS works most of the time, Apple recommends POKE 1403,POS (0-79) for the //e. This routine will not work at all for an old Apple.

It is the programmer's responsibilty to turn off the 80 column card at the end of a program. Do not quit the card with the cursor beyond the 39th column, as this can cause unpredictable results including program erasure. In case of accidently executing this command, pressing RETURN immediately will usually recover the cursor to the left margin. It is also necessary to turn the 80 column card off before sending output to printers, modems, etc.

VTAB no longer works when a window is set (by POKing 32,33 etc.). The solution is to VTAB to the location -1, and then do a PRINT prior to PRINTing the actual data. This causes the firmware to recognise the new VTAB location.

These cautions are a small price to pay for the increased versatility and flexability of the new Apple //e.

There is 1 page 0 location that was not formerly used which is now used.

$1F (31) [YSAV1] \P1\ Temporary storage for the Y register

There are several locations in the text page that are storage for permanent data in these unused screen locations. Any routine which sets page 2 must restore page 1 so that these data may be accessed.

$478 (1144) [TEMP1] \P1\ A temporary storage location
$47B (1147) [OLDCH] \P1\ Old CH set for user
$4FB (1275) [MODE] \P1\ Current operating mode acording to the following bits:

Bit 0 Off Normal mode (Pascal)
Bit 0 On Transparent mode (Pascal)
Bit 0 Off Caller set interrupts (BASIC)
Bit 0 On Caller cleared interrupts (BASIC)
Bit 1 Off Pascal 1.1 F/W active
Bit 1 On Pascal 1.0 Interface
Bit 2 Off Normal video (Pascal)
Bit 2 On Inverse video (Pascal)
Bit 3 Off GOTOXY not in progress
Bit 3 On GOTOXY in progress
Bit 4 Off Upper case restrict mode
Bit 4 On Literal upper/lower case mode
Bit 5 Off Current language is BASIC
Bit 5 On Current language is Pascal
Bit 6 Off BASIC PRINT
Bit 6 On BASIC INPUT
Bit 7 Off ESC-R inactive
Bit 7 On ESC-R active

$57B (1403) [OURCH] \P1\ 80 column CH
$5FB (1531) [OURCV] \P1\ Cursor vertical
$67B (1659) [CHAR] \P1\ In/Out character
$6FB (1787) [XCOORD] \P1\ X coordinate in GOTOXY routine
$77B (1915) [OLDBASL] \P1\ Pascal saved BASL
$7FB (2043) [OLDBASH] \P1\ Pascal saved BASH

```
$C000-$C01F (49152-49183) \H\          Hardware locations/switches
$C000 (49152) [CLR80COL] \H1\          Disable 80 column store
$C001 (49153) [SET80COL] \H1\          Enable 80 column store
$C002 (49154) [RDMAINRAM] \H1\         Read RAM on mainboard
$C003 (49155) [RDCARDRAM] \H1\         Read RAM on card
$C004 (49156) [WRMAINRAM] \H1\         Write RAM on mainboard
$C005 (49157) [WRCARDRAM] \H1\         Write RAM on card
$C007 (49159) [SETINTCXROM] \H1\       Set internal CX00 ROM
$C008 (49160) [SETSTDZP] \H1\          Set standard zero page/stack
$C009 (49161) [SETALTZP] \H1\          Set alternate zero page/stack
$C00B (49163) [SETSLOTC3ROM] \H1\      Enable C300 slot ROM
$C00C (49164) [CLR80VID] \H1\          Disable 80 column video
$C00D (49165) [SET80VID] \H1\          Enable 80 column video
$C00E (49166) [CLRALTCHAR] \H1\        Normal lower case, flash upper case
$C00F (49167) [SETALTCHAR] \H1\        Normal/inverse lower case, no flash
$C011 (49169) [RDLCBNK2] \H1\          Reads language card bank 2
$C012 (49170) [RDLCRAM] \H1\           Reads language card RAM enable
$C013 (49171) [RDRAMRD] \H1\           Reads RAMREAD state
$C014 (49172) [RDRAMWRT] \H1\          Reads BANKWRT state
$C018 (49176) [RD80COL] \H1\           Reads SET80COL
$C019 (49177) [RDVBLBAR] \H1\          Reads VBL signal
$C01A (49178) [RDTEXT] \H1\            Reads Text mode
$C01C (49180) [RDPAGE2] \H1\           Reads page 1/2 status
$C01F (49183) [RD80VID] \H1\           Reads SET80VID
$C100-$CFFF (49408-53247) [CX00ROM] \SB\ A new set of subroutines to handle the 80 column card and auxilliary
                                       memory in slot 3. It is entered from the GOTOCX subroutine in the F800 ROM which
                                       sets interrupts, turns on the CX00 ROMs, and JMPs to C100. Function code is in
                                       Y reg. Note: "B." routines are the new way. "F." routines are the old way.
                                       Stack has status of bank and IRQ. Uses A,Y registers.
                                       Function Codes:
                                       0      CLREOP
                                       1      HOME
                                       2      SCROLL
                                       3      CLREOL
                                       4      CLEOLZ
                                       5      INIT & RESET
                                       6      KEYIN
                                       7      Fix ESCape Character
                                       8      SETWND
                                       If there is a card in the slot then the new video routines are used, since the
                                       screen hole locations belong to the card. Otherwise the F8 ROM routines are
                                       duplicated to avoid slot 3 interference with another type of interface.
$C100 (49408) [B.FUNC] \SE\            Entry point for all routines with code in Y.Check first for KEYIN Y=6
$C107 (49415) [B.FUNCNK] \SE\          Check for ESCape-fix Y=7
$C10E (49422) [B.FUNCNE] \SE\          Test for card. If present, use the new routines, if not, old routines
```

$C11F (49439) [B.OLDFUNC] \SE\	Pushes $C1 on stack, and low byte address of the function -1 by looking up in F.TABLE indexed by Y. Then does fake RTS to routine.
$C129 (49449) [F.CLREOP] \SE\	Monitor S/R to clear from the cursor to the end of page.
$C143 (49475) [F.HOME] \SE\	Clear scroll window to blanks. Set cursor to top left corner.
$C14D (49485) [F.SCROLL] \SE\	Monitor S/R to scroll up one line.
$C17D (49533) [F.CLREOL] \SE\	Monitor S/R to clear to end of line.
$C18A (49546) [F.SETWND] \SE\	Monitor S/R to set normal low-resolution graphics window, cursor bottom left.
$C19C (49564) [F.CLEOLZ] \SE\	Monitor S/R to clear entire line.
$C1A1 (49569) [F.GORET] \L\	Exit routine to F.RETURN
$C1A4 (49572) [B.FUNCO] \SE\	Entry point to new routines. Sets the IRQ mode and screen holes, Y reg.
$C1CD (49613) [B.SCROLL] \SE\	Entry point for monitor routine to scroll up one line
$C1D3 (49619) [B.CLREOL] \SE\	Entry point for monitor routine to clear to end of line
$C1D9 (49625) [B.CLEOLZ] \SE\	Entry point for monitor routine to clear entire line
$C1E1 (49633) [B.CLREOP] \SE\	Entry point for monitor routine to clear to end of page
$C1E7 (49639) [B.SETWND] \SE\	Entry point for monitor routine to set text window
$C1EA (49642) [B.RESET] \SE\	Entry point for monitor routine to reset entire system
$C1ED (49645) [B.HOME] \SE\	Monitor S/R to clear the text page and put cursor in upper left corner
$C1FF (49663) [B.VECTOR] \SE\	Monitor S/R to check on 80 col use and get current Cursor Horizontal position (CH)
$C20E (49678) [B.GETCH] \SE\	Save CH in screenhole
$C211 (49681) [B.FUNC1] \SE\	Pushes $C1 on stack, and low byte address of the function -1 by looking up in B.TABLE indexed by Y. Then does fake RTS to routine.
$C219 (49689) [B.SETWNDX] \SE\	Monitor S/R to set normal text window 40/80 columns
$C234 (49716) [B.RESETX] \SE\	Monitor routine to reset system, checks for "Apple" keys for cold start, else does warm restart without diagnostics, blasts memory from BFXX down to stack, checks 80 col board to see if CX ROM needs resetting, and returns
$C261 (49761) [DIAGS] \SE\	Entry point for monitor S/R diagnostics
$C26E (49774) [B.ESCFIX] \SE\	Monitor S/R to map i,j,k,m and <-,^,-> , and V into I,J,K,M for cursor movement Returns with old form of character in A.
$C280 (49792) [ESCIN] \P4\	Table of arrow keys
$C284 (49796) [ESCOUT] \P4\	"J,K,M,I" translations for arrows
$C288 (49800) [B.KEYIN] \SE\	Monitor routine to read a key with new additions to save CX bank status, check interrupt status, put new cursor ASC"$FF" on screen, JSR to KEYDLY (old RDKEY), restore the original screen character, put the new character in A reg., clear the keyboard strobe and return to caller.
$C2C6 (49862) [KEYDLY] \SE\	Monitor routine to get a key from KBD, also checking interrupts, and still incrementing RNDL and RNDH, the random locations
$C2EB (49899) [F.RETURN] \SE\	Monitor routine to exit from CX ROM routines either leaving I/O disabled or enabling it if it was on entry
$C300 (49920) [BASICINT] \SE\	Sets INIT Flag (V) and branches to BASIC I/O entry point
$C307 (49927) [BASICOUT] \SE\	Clears INIT Flag (V) and branches to BASIC I/O entry point
$C30B (49931) [PASFPT] \P6\	Pascal 1.1 firmware protocol table
$C311 (49937) [128KJMP] \P6\	Jump table for 128K support routines
$C317 (49943) [BASICENT] \SE\	BASIC I/O entry point, saves CHAR, A, Y, X, and P, pulls P from stack, checks IRQ status, and sets appropriately.

HEX LOCN (DEC LOCN) [NAME] \USE-TYPE\ - DESCRIPTION

$C336 (49974) [BASICENT2] \SE\	Turns off any slots using C8 area, sets C8SLOT to $C3, checks INIT flag, and jumps to warm or cold BASIC in C8 ROM
$C34B-$C362 (49995-50018) [PJUMPS]	Pascal jump table
$C34B (49995) [JPINIT] \SE\	Pascal INIT
$C351 (50001) [JPREAD] \SE\	Pascal READ
$C357 (50007) [JPWRITE] \SE\	Pascal WRITE
$C35D (50013) [JPSTAT] \SE\	Pascal STATUS
$C363 (50019) [MOVE] \SE\	Monitor S/R to move memory across memory banks. Call with A1 = Source start, A2 = Source end, A4 = Destination start, Carry set for Main to Card, Carry clear for Card to Main.
$C3B0 (50096) [XFER] \SE\	Transfer program control from main board to card or vice versa. $3ED-$3EE is address to be executed upon transfer, carry set means transfer to card, carry clear means transfer to main board, V flag clear means use standard zero page/stack, V flag set means use alternate zero page/stack. Also uses $3ED-$3EE in destination bank. Enter via JMP not JSR.
$C3EB (50155) [SETC8] \SE\	Setup IRQ C800 protocol. Stores $C3 in C8SLOT.
$C800 (51200) [PINIT1] \SE\	Pascal 1.0 init
$C803 (51203) [BASICINIT] \SE\	Checks the F8 ROM version, if not //e, copies ROM to RAM Card, and checks again, if still not good, hangs the system.
$C816 (51222) [BINIT1] \SE\	Set up BASIC I/O in CSW and KSW to point to BASICENT in the C3 ROM and set text or graphics windows
$C84B (51272) [PREAD1.0]	Pascal 1.0 input hook
$C850 (51280) [BINIT2] \L\	Check for 80 column mode and enable, if so
$C85D (51293) [CLEARIT] \L\	Monitor routine to set lower case mode, clear screen and clears carry
$C866 (51302) [C8BASIC] \L\	Monitor routine to check mode and set 80 column store in case Integer BASIC cleared Also rounds WNDWDTH to next lower even, if odd in 80 column mode.
$C874 (51316) [C8B2] \L\	Monitor routine to check current CH and store it if different from OLDCH
$C87E (51326) [C8B3] \L\	Monitor routine to check RAM card for correct version and, if not, recopy the F8ROM to RAM card , check again and hang if not correct.
$C890 (51344) [C8B4] \L\	Monitor routine to check carry, on clear-print a character, set-input a character
$C896 (51350) [BOUT] \SE\	Monitor S/R to set MODE to BASIC printing, falls through to BPRINT
$C8A1 (51361) [BPRINT] \SE\	Monitor S/R to output character in CHAR, checks for CTRL-S, clears high bit, checks for CTRL chars, if it is, process and return, if not, fall through to BPNCTL.
$C8CC (51404) [BPNCTL] \SE\	Monitor S/R to reload CHAR (to get 8th bit, and print the char on the screen, Increments cursor horizontal and scrolls, if necessary
$C8E2 (51426) [BIORET] \L\	Monitor routine to store cursor position, restore X, Y, and A and return to BASIC
$C8F6 (51446) [BINPUT] \SE\	Monitor routine to set MODE to BASIC input, get the cursor position, and CHAR
$C905 (51461) [B.INPUT] \SE\	Monitor routine to inverse char at current position, get a char from the keyboard, remove cursor, and process char, including ESCapes. If not ESC then JMP to NOESC.

$C918 (51480) [ESCAPING] \SE\
Monitor routine to process ESCape command sequences. The commands are:
@ - Home and Clear screen
E - Clear to end of line
F - Clear to end of page
A,K,-> - Cursor right
B,J,<- - Cursor left
C,M,V - Cursor down
D,I,^ - Cursor up
R - Restrict to uppercase
T - Turn off Esc-R
4 - Go to 40 column mode
8 - Go to 80 column mode
CTRL-Q - Quit new routines. (PR#0/IN#0)

Places ESCape cursor on screen, GETs a command key, puts lower case into upper, checks the ESCTAB for a valid character. If the char is there, load A with the Y index into ESCCHAR, and "print" the control character, if its not, check for "T", "R" and "CTRL-Q" special functions and process, if its not, return to caller. If the ESCCHAR entry has the high bit set, return to ECSAPING, otherwise return to caller.

$C972 (51570) [ESCTAB] \P17\
Table of ESCape codes

$C983 (51587) [ESCCHAR] \P17\
Table of corresponding control codes-high bit set for "remain in ESCape mode"

$C994 (51604) [PSTATUS] \SE\
pascal check if ready for input or output, return 3 in X if not ready (ILLEGAL OPERATION)

$C9A6 (51622) [PHOOK] \SE\
Pascal 1.0 output hook

$C9B7 (51639) [NOESC] \SE\
Monitor routine to process normal characters. Checks for copy char (right arrow), literal input, double quotes to turn literal input off/on, and restricted case input before storing in CHAR and returning to caller

$C9DF (51679) [B.CHKCAN] \L\
Monitor routine to check for cancelling literal mode

$C9F7 (51703) [B.FLIP] \L\
Monitor routine to switch the literal mode

$CA02 (51714) [B.CANLIT] \L\
Monitor routine to cancel literal mode

$CA0A (51722) [B.FIXCHAR] \L\
Monitor routine to up/shift the character in non-literal or restrict mode

$CA24 (51748) [B.INRET] \L\
Monitor routine to return to caller from input

$CA27 (51751) [GETPRIOR] \SE\
Monitor S/R to get the character before the cursor. Uses OURCH, OURCV; destroys A, TEMP1; outputs BEQ if character is double quote, BNE if not. Used for changing literal mode if backspacing over a double quote.

$CA4A (51786) [PINIT1.0] \SE\
Pascal initialization 1.0

$CA4F (51791) [PINIT] \SE\
Pascal initialization 1.1

$CA51 (51793) [PINIT2] \L\
Set up for running Pascal, set mode, set window, zero page, check for card, return X=9 (NO DEVICE) if missing, turn on card, set normal lower case mode, home and clear screen, put cursor on screen and return.

$CA74 (51828) [PREAD] \SE\
Pascal input-Get a character, remove high bit, store in CHAR, if 1.1 return "$C3" in X, 1.0 return CHAR in A

$CA8E (51854) [PWRITE] \SE\
Pascal output-Set zero page, turn cursor off, check GOTOXY Mode and process if necessary, check if GOTOXY and start if true, else store it on screen, increment cursor horizontal, check if transparent mode and do carriage return/line feed if necessary, replace the cursor and return.

$CB15 (51989) [GETKEY] \SE\	Monitor S/R to read the keyboard, incrementing the random locations while waiting, load the char into A, clear the keyboard strobe and return
$CB24 (52004) [TESTCARD] \SE\	Monitor S/R to test for presence of 80 column card, destroys A,Y; returns BEQ if card is there, BNE if not.
$CB51 (52049) [BASCALC] \SE\	Monitor S/R to calculate base address for screen line using OURCV. Stores result in BASL/BASH.
$CB54 (52052) [BASCALCZ] \SE\	Monitor S/R to calculate base address for screen line using CV. Checks for 40/80 column mode and if IRQ is enabled and not in Pascal, uses SNIFFIRQ to check for interrupts.
$CB99 (52121) [CTLCHAR] \SE\	Monitor S/R to process command control characters. Char in A to process, returns BCC if executed, BCS if not control command.
$CBB6 (52150) [CTLXFER] \L\	Monitor routine to push CTLADH and CTLADL onto stack for control routine address and execute a fake RTS.
$CBBC (52156) [X.BELL] \SE\	Monitor S/R to beep speaker, same as F8: BELL1
$CBCF (52175) [WAIT] \SE\	Monitor S/R to wait depending on A. Same as F8: WAIT
$CBDB (52187) [X.BS] \SE\	Monitor S/R to execute a backspace
$CBEC (52204) [X.CR] \SE\	Monitor S/R to execute a carriage return
$CC0D (52237) [X.EM] \SE\	Monitor S/R to execute HOME
$CC1A (52250) [X.SUB] \SE\	Monitor S/R to execute clear line
$CC26 (52262) [X.FS] \SE\	Monitor S/R to execute a forward space
$CC34 (52276) [X.US] \SE\	Monitor S/R to execute a reverse linefeed
$CC49 (52297) [X.SO] \SE\	Monitor S/R to execute "normal video"
$CC52 (52306) [X.SI] \SE\	Monitor S/R to execute "inverse video"
$CC5F (52319) [CTLADL] \P24\	Table of low byte addresses for control characters subroutines: 0 = Invalid
$CC78 (52344) [CTLADH] \P24\	Table of high byte addresses for control character subroutines: 0 = Invalid
$CC91 (52369) [X.LF] \SE\	Monitor S/R to execute linefeed
$CCA4 (52388) [SCROLLUP] \SE\	Monitor S/R to scroll the screen up one line
$CCAA (52394) [SCROLLDN] \SE\	Monitor S/R to scroll the screen down one line
$CCAE (52398) [SCROLL1] \L\	Monitor routine to check for 40/80 columns
$CCB8 (52408) [SCROLL2] \L\	Monitor routine to scroll 40 columns
$CCC0 (52416) [SCROLL80] \L\	Monitor routine to scroll the other 40 columns
$CCD1 (52433) [SCRLSUB] \SE\	Monitor S/R to scroll only 40 column active window
$CD11 (52497) [X.SCRLRET] \L\	Monitor rotuine to clear top or bottom line (depending on scroll up or down) Return to user via BASCALC.
$CD23 (52515) [X.VT] \SE\	Monitor S/R to clear to end of page
$CD42 (52546) [X.FF] \SE\	Monitor S/R to home the cursor. Returns via X.VT to clear screen.
$CD48 (52552) [X.GS] \SE\	Monitor S/R to clear to end of line
$CD4E (52558) [X.GSEOLZ] \SE\	Monitor S/R to clear entire line
$CD59 (52569) [X.DC1] \SE\	Monitor S/R to set 40 column mode
$CD77 (52599) [X.DC2] \SE\	Monitor S/R to set 80 column mode
$CD90 (52624) [X.NAK] \SE\	Monitor S/R/ to quit 80 column card
$CD9B (52635) [FULL80] \SE\	Monitor S/R to set full 80 column window parameters
$CDAA (52650) [QUIT] \SE\	Monitor S/R to restore 40 column window,convert 80 to 40 if needed, set cursor at bottom left corner, reset video and keyboard to old mode

$CDDB (52699) [SCRN84] \SE\	Monitor S/R to convert 80 column screen to 40 column screen. Moves leftmost 40 characters to TXTPAGE1
$CE0A (52746) [ATEFOR] \SE\	Monitor S/R to convert one line from 80 to 40 columns
$CE22 (52770) [GET84] \SE\	Monitor S/R to move one character from 80 window to 40 window
$CE32 (52786) [SCRN48] \SE\	Monitor S/R to convert 40 column screen to 80 column screen. Moves whole 40 character screen to left most 40 positions on 80 column screen
$CE63 (52835) [FORATE] \SE\	Monitor S/R/ to convert one line from 80 to 40 columns
$CE91 (52881) [CLRHALF] \SE\	Monitor S/R to clear right half of both screen pages
$CEA3 (52899) [DO48] \L\	Monitor S/R to move one character from 80 to 40 columns
$CEAF (52911) [SETCH] \SE\	Monitor S/R to set OURCH and CH. In 40 column mode sets to A value. In 80 column mode, sets to 0 unless less than 8 from end of line, in which case moves up near right
$CEDD (52957) [INVERT] \SE\	Monitor S/R to invert the character at the current screen location: CH,CV
$CEF2 (52978) [STORCHAR] \SE\	Monitor S/R to store character in A at screen horizontal position Y.
$CF01 (52993) [PICK] \SE\	Monitor S/R to read the character at screen position Y = horizontal, returns with character in A
$CF06 (52998) [SCREENIT] \SE\	Monitor S/R/ to either store character on screen or read character from screen. V clear for pick, V set for store, character in A for store, Y = CH position. Saves Y and checks for mode. 40 branches to SCREEN40, 80 falls through to SCREEN80
$CF0E (53006) [SCREEN80] \L\	Monitor routine to calculate which page, and if V set, branch to STOR80, otherwise read the character from the screen and return.
$CF2A (53034) [STOR80] \L\	Monitor routine to store the character on the screen.
$CF37 (53047) [SCREEN40] \L\	Monitor routine to get cursor position, and if V set, branch to STOR40, otherwise read the character from the screen and return.
$CF4A (53066) [STOR40] \L\	Monitor routine to store the character on the screen.
$CF52 (53074) [ESCON] \SE\	Monitor S/R to save current character in CHAR and put inverse "+" on screen. Returns via ESCRET.
$CF65 (53093) [ESCOFF] \SE\	Monitor S/R to replace original character back on the screen that was saved in CHAR. Falls through to ESCRET.
$CF6E (53102) [ESCRET] \L\	Monitor routine to put character on screen and return.
$CF78 (53112) [COPYROM] \SE\	Monitor S/R to copy the F8 ROM to the language card. Destroys X and Y. Uses CSWL/CSWH (which it saves) as hook for transfer. Sets ROM/RAM banks for transfer, moves the bytes, and resets the language card to it's previous state before returning.
$CFC8 (53192) [PSETUP] \SE\	Monitor S/R to set up zero page for Pascal operation. Checks 40-80 columns, sets INVFLG, and updates BASL/BASH before returning.
$CFEA (53226) [F.TABLE] \P9\	Table of addresses for ESCape functions in 40 column mode. Entries at $CFF0-1 are used by SCROLL (Label = PLUSMINUS1).
$CFF3 (53235) [B.TABLE] \P9\	Table of addresses for ESCape functions in 80 column mode. Entries at $CFF9-A are used by SCROLL (Label = WNDTAB).

Changes in the F800 ROM

$F7FF (63487) [?] was $D7, is now $78, appears to be unused
$FA75-$FA7A (64117-64122) [RESET] A change in the RESET code to allow for the presence of an 80 column card. Does a JSR to GOTOCX Y=5
$FB0A-$FB0D (64226-64269) [TITLE] APPLE][-> Apple][
$FB51-$FB54 (64337-64340) [SETWND] A change in the SETWND code to allow for the presence of an 80 column card. Does a branch to GOTOCX Y=8
$FBA3 (64419) [ESCNOW] A change in the ESCNOW code to allow for i,j,k,m and arrow keys. Does JSR to RSDEC which is the old KEYIN2
$FBB3 (64435) [VERSION] ID code for check on which kind of Apple it is //e=$06][+=$EA][=$38
$FBB4-$FBC0 (64436-64448) [GOTOCX] Formerly NOPs, now code to save current ROM states, set interrupts, turn on CX00 ROMS and JMP to C100:new code for 80 cols. Requires function code to be in Y Reg.
$FC42-$FC45 (64578-64581) [CLREOP] Changed to branch to GOTOCX Y=0
$FC46-$FC57 (64582-64599) [COPYRT] Notice of copyright "(C) 1981-82, APPLE"
$FC58-$FC5B (64600-64603) [HOME] Changed to branch to GOTOCX Y=1
$FC5C-$FC61 (64604-64609) [AUTHOR1] "RICK A" for Rick Auricchio
$FC70-$FC71 (64624-64625) [SCROLL] Changed to jump to GOTOCX Y=2
$FC72-$FC74 (64626-64628) [XGOTOCX] A JMP to GOTOCX for long branching purposes
$FC75-$FC9B (64629-64667) [SNIFFIRQ] IRQ Sniffer for Video Code: A new routine to check the current video mode, CXROM usage, and check for interrupts
$FC9C-$FC9D (64668-64669) [CLREOL] Changed to branch to GOTOCX Y=3
$FC9E-$FCA7 (64670-64679) [CLREOLZ] Changed to branch to GOTOCX Y=4
$FD1B-$FD20 (64795-64800) [KEYIN] Changed to jump to GOTOCX Y=6 KEYIN no longer falls through to KEYIN2.
$FD21-$FD28 (64801-64808) [RDESC] Formerly KEYIN2, changed to jump to GOTOCX Y=7
$FD29-$FD2D (64809-64813) [FUNCEXIT] Return from GOTOCX here: A new routine that restores the CXROM bank and the IRQ before an RTS to the calling routine.
$FD30 (64816) [ESC] A change to JSR to RDESC instead of RDKEY
$FD42-$FD43 (64834-64835) [NOTCR] A change to NOPs of the cursor inverse mode. No longer needed now that the cursor is a standard character.
$FD83 (64899) [CAPTST] \P1\ A change in the input AND mask that used to convert lower case input to upper case
$FEAF (65199) [CKSUMFIX] \P1\ Correct CKSUM at create time.
$FEC5-$FEC9 (65221-65225) [AUTHOR2] "Bryan" for Bryan Stearns.

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION

? (63487) [$F7FF] was $D7, is now $78, appears to be unused

ATEFOR (52746) [$CE0A] \SE\ Monitor S/R to convert one line from 80 to 40 columns

AUTHOR1 (64604-64609) [$FC5C-$FC61] "RICK A" for Rick Auricchio

AUTHOR2 (65221-65225) [$FEC5-$FEC9] "Bryan" for Bryan Stearns.

B.CANLIT (51714) [$CA02] \L\ Monitor routine to cancel literal mode

B.CHKCAN (51679) [$C9DF] \L\ Monitor routine to check for cancelling literal mode

B.CLREOL (49619) [$C1D3] \SE\ Entry point for monitor routine to clear to end of line

B.CLEOLZ (49625) [$C1D9] \SE\ Entry point for monitor routine to clear entire line

B.CLREOP (49633) [$C1E1] \SE\ Entry point for monitor routine to clear to end of page

B.ESCFIX (49774) [$C26E] \SE\ Monitor S/R to map i,j,k,m and <-,^,->, and V into I,J,K,M for cursor movement

B.INPUT (51461) [$C905] \SE\ Monitor routine to inverse char at current position, get a char from the keyboard, remove cursor, and process char, including ESCapes. If not ESC then JMP to NOESC.

B.FIXCHAR (51722) [$CA0A] \L\ Monitor routine to up/shift the character in non-literal or restrict mode

B.FLIP (51703) [$C9F7] \L\ Monitor routine to switch the literal mode

B.FUNC (49408) [$C100] \SE\ Entry point for all routines with code in Y.Check first for KEYIN Y=6

B.FUNC1 (49681) [$C211] \SE\ Pushes $C1 on stack, and low byte address of the function -1 by looking up in B.TABLE indexed by Y. Then does fake RTS to routine.

B.FUNCNE (49422) [$C10E] \SE\ Test for card. If present, use the new routines, if not, old routines

B.FUNCNK (49415) [$C107] \SE\ Check for ESCape-fix Y=7

B.FUNCO (49572) [$C1A4] \SE\ Entry point to new routines. Sets the IRQ mode and screen holes, Y reg.

B.GETCH (49678) [$C20E] \SE\ Save CH in screenhole

B.INRET (51748) [$CA24] \L\ Monitor routine to return to caller from input

B.KEYIN (49800) [$C288] \SE\ Monitor routine to read a key with new additions to save CX bank status, check interrupt status, put new cursor ASC"$FF" on screen, JSR to KEYDLY (old RDKEY)

B.OLDFUNC (49439) [$C11F] \SE\ Pushes $C1 on stack, and low byte address of the function -1 by looking up in F.TABLE indexed by Y. Then does fake RTS to routine.

B.RESETX (49716) [$C234] \SE\ Monitor routine to reset system, checks for "Apple" keys for cold start, else does warm restart without diagnostics, blasts memory from BFXX down to stack, checks 80 col board to see if CX ROM needs resetting, and returns

B.SCROLL (49613) [$C1CD] \SE\ Entry point for monitor routine to scroll up one line

B.SETWND (49639) [$C1E7] \SE\ Entry point for monitor routine to set text window

B.SETWNDX (49689) [$C219] \SE\ Monitor S/R to set normal text window 40/80 columns

B.TABLE (53235) [$CFF3] \P9\ Table of addresses for ESCape functions in 80 column mode. Entries at $CFF9-A are used by SCROLL (Label = WNDTAB).

B.VECTOR (49663) [$C1FF] \SE\ Monitor S/R to check on 80 col use and get current Cursor Horizontal position (CH)

BASCALC (52049) [$CB51] \SE\ Monitor S/R to calculate base address for screen line using OURCV. Stores result in BASL/BASH.

BASCALCZ (52052) [$CB54] \SE\ Monitor S/R to calculate base address for screen line using CV. Checks for 40/80 column mode and if IRQ is enabled and not in Pascal, uses SNIFFIRQ to check for interrupts.

BASICENT (49943) [$C317] \SE\ BASIC I/O entry point, saves CHAR, A, Y, X, and P, pulls P from stack, checks IRQ status, and sets appropriately.

BASICENT2 (49974) [$C336] \SE\ Turns off any slots using C8 area, sets C8SLOT to $C3, checks INIT flag, and jumps to warm or cold BASIC in C8 ROM

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION

BASICINIT (51203) [$C803] \SE\ Checks the F8 ROM version, if not //e, copies ROM to RAM Card, and checks again, if still not good, hangs the system.

BASICINT (49920) [$C300] \SE\ Sets INIT Flag (V) and branches to BASIC I/O entry point

BASICOUT (49927) [$C307] \SE\ Clears INIT Flag (V) and branches to BASIC I/O entry point

BINIT1 (51222) [$C816] \SE\ Set up BASIC I/O in CSW and KSW to point to BASICENT in the C3 ROM and set text or graphics windows

BINIT2 (51280) [$C850] \L\ Check for 80 column mode and enable, if true

BINPUT (51446) [$C8F6] \SE\ Monitor routine to set MODE to BASIC input, get the cursor position, and CHAR

BIORET (51426) [$C8E2] \L\ Monitor routine to store cursor position, restore X, Y, and A and return to BASIC

BOUT (51350) [$C896] \SE\ Monitor S/R to set MODE to BASIC printing, falls through to BPRINT

BPNCTL (51404) [$C8CC] \SE\ Monitor S/R to reload CHAR (to get 8th bit, and print the char on the screen, Increments cursor horizontal and scrolls, if necessary

BPRINT (51361) [$C8A1] \SE\ Monitor S/R to output character in CHAR, checks for CTRL-S, clears high bit, checks for CTRL chars, if it is, process and return, if not, fall through to BPNCTL.

C8B2 (51316) [$C874] \L\ Monitor routine to check current CH and store it if different from OLDCH

C8B3 (51326) [$C87E] \L\ Monitor routine to check RAM card for correct version and, if not, recopy the F8ROM to RAM card , check again and hang if not correct.

C8B4 (51344) [$C890] \L\ Monitor routine to check carry, on clear-print a character, set-input a character

C8BASIC (51302) [$C866] \L\ Monitor routine to check mode and set 80 column store in case Integer BASIC cleared Also rounds WNDWDTH to next lower even, if odd in 80 column mode.

CAPTST (64899) [$FD83] \P1\ A change in the input AND mask that used to convert lower case input to upper case

CHAR (1659) [$67B] \P1\ In/Out character

CKSUMFIX (65199) [$FEAF] \P1\ Correct CKSUM at create time.

CLEARIT (51293) [$C85D] \L\ Monitor routine to set lower case mode, clear screen and clears carry

CLR80COL (49152) [$C000] \H1\ Disable 80 column store

CLR80VID (49164) [$C00C] \H1\ Disable 80 column video

CLRALTCHAR (49166) [$C00E] \H1\ Normal lower case, flash upper case

CLREOL (64668-64669) [$FC9C-$FC9D] Changed to branch to GOTOCX Y=3

CLREOLZ (64670-64679) [$FC9E-$FCA7] Changed to branch to GOTOCX Y=4

CLREOP (64578-64581) [$FC42-$FC45] Changed to branch to GOTOCX Y=0

CLRHALF (52881) [$CE91] \SE\ Monitor S/R to clear right half of both screen pages

COPYROM (53112) [$CF78] \SE\ Monitor S/R to copy the F8 ROM to the language card. Destroys X and Y. Uses CSWL/CSWH (which it saves) as hook for transfer. Sets ROM/RAM banks for transfer, moves the bytes, and resets the language card to it's previous state before returning.

COPYRT (64582-64599) [$FC46-$FC57] Notice of copyright "(C) 1981-82, APPLE"

CTLADH (52344) [$CC78] \P24\ Table of high byte addresses for control character subroutines: 0 = Invalid

CTLADL (52319) [$CC5F] \P24\ Table of low byte addresses for control characters subroutines: 0 = Invalid

CTLCHAR (52121) [$CB99] \SE\ Monitor S/R to process command control characters. Char in A to process, returns BCC if executed, BCS if not control command.

CTLXFER (52150) [$CBB6] \L\ Monitor routine to push CTLADH and CTLADL onto stack for control routine address and execute a fake RTS.

CX00ROM (49408-53247) [$C100-$CFFF] \SB\ A new set of subroutines to handle the 80 column card and auxilliary memory in slot 3. It is entered from the GOTOCX subroutine in the F800 ROM which sets interrupts, turns on the CX00 ROMs, and JMPs to C100. Function code is in Y reg. Note: "B." routines are the new way. "F." routines are the old way. Stack has status of bank and IRQ. Uses A,Y registers.

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION

DIAGS (49761) [$C261] \SE\ Entry point for monitor S/R diagnostics

DO48 (52899) [$CEA3] \L\ Monitor routine to move one character from 80 to 40 columns

ESC (64816) [$FD30] A change to JSR to RDESC instead of RDKEY

ESCAPING (51480) [$C918] \SE\ Monitor routine to process ESCape command sequences. Places ESCape cursor on screen, GETs a command key, puts lower case into upper, checks the ESCTAB for a valid character. If the char is there, load A with the Y index into ESCCHAR, and "print" the control character, if its not, check for "T", "R" and "CTRL-Q" special functions and process, if its not, return to caller. If the ESCCHAR entry has the high bit set, return to ECSAPING, otherwise return to caller.

ESCCHAR (51587) [$C983] \P17\ Table of corresponding control codes-high bit set for "remain in ESCape mode"

ESCIN (49792) [$C280] \P4\ Table of arrow keys

ESCNOW (64419) [$FBA3] A change in the ESCNOW code to allow for i,j,k,m and arrow keys. Does JSR to RSDEC which is the old KEYIN2

ESCOFF (53093) [$CF65] \SE\ Monitor S/R to replace original character back on the screen that was saved in CHAR. Falls through to ESCRET.

ESCON (53074) [$CF52] \SE\ Monitor S/R to save current character in CHAR and put inverse "+" on screen. Returns via ESCRET.

ESCOUT (49796) [$C284] \P4\ "J,K,M,I" translations for arrows

ESCRET (53102) [$CF6E] \L\ Monitor routine to put character on screen and return.

ESCTAB (51570) [$C972] \P17\ Table of ESCape codes

F.CLREOL (49533) [$C17D] \SE\ Monitor S/R to clear to end of line.

F.CLEOLZ (49564) [$C19C] \SE\ Monitor S/R to clear entire line.

F.CLREOP (49449) [$C129] \SE\ Monitor S/R to clear from the cursor to the end of page.

F.GORET (49569) [$C1A1] \L\ Exit routine to F.RETURN

F.HOME (49475) [$C143] \SE\ Clear scroll window to blanks. Set cursor to top left corner.

F.RETURN (49899) [$C2EB] \SE\ Monitor routine to exit from CX ROM routines either leaving I/O disabled or enabling it if it was on entry

F.SCROLL (49485) [$C14D] \SE\ Monitor S/R to scroll up one line.

F.SETWND (49546) [$C18A] \SE\ Monitor S/R to set normal low-resolution graphics window, cursor bottom left.

F.TABLE (53226) [$CFEA] \P9\ Table of addresses for ESCape functions in 40 column mode. Entries at $CFF0-1 are used by SCROLL (Label = PLUSMINUS1).

FORATE (52835) [$CE63] \SE\ Monitor S/R/ to convert one line from 80 to 40 columns

FULL80 (52635) [$CD9B] \SE\ Monitor S/R to set full 80 column window parameters

FUNCEXIT (64809-64813) [$FD29-$FD2D] Return from GOTOCX here: A new routine that restores the CXROM bank and the IRQ before an RTS to the calling routine.

GET84 (52770) [$CE22] \SE\ Monitor S/R to move one character from 80 window to 40 window

GETKEY (51989) [$CB15] \SE\ Monitor S/R to read the keyboard, incrementing the random locations while waiting, load the char into A, clear the keyboard strobe and return

GETPRIOR (51751) [$CA27] \SE\ Monitor S/R to get the character before the cursor. Uses OURCH, OURCV; destroys A, TEMP1; outputs BEQ if character is double quote, BNE if not. Used for changing literal mode if backspacing over a double quote.

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION

GOTOCX (64436-64448) [$FBB4-$FBC0] Formerly NOPs, now code to save current ROM states, set interrupts, turn on CX00 ROMS and JMP to C100:new code for 80 cols. Requires function code to be in Y Reg.

HOME (64600-64603) [$FC58-$FC5B] Changed to branch to GOTOCX Y=1

INVERT (52957) [$CEDD] \SE\ Monitor S/R to invert the character at the current screen location: CH,CV

JPINIT (49995) [$C34B] \SE\ Pascal INIT

JPREAD (50001) [$C351] \SE\ Pascal READ

JPSTAT (50013) [$C35D] \SE\ Pascal STATUS

JPWRITE (50007) [$C357] \SE\ Pascal WRITE

KEYDLY (49862) [$C2C6] \SE\ Monitor routine to get a key from KBD, also checking interrupts, and still incrementing RNDL and RNDH, the random locations

KEYIN (64795-64800) [$FD1B-$FD20] Changed to jump to GOTOCX Y=6 KEYIN no longer falls through to KEYIN2.

MODE (1275) [$4FB] \P1\ Current operating mode acording bits set.

MOVE (50019) [$C363] \SE\ Monitor S/R to move memory across memory banks. Call with A1 = Source start, A2 = Source end, A4 = Destination start, Carry set for Main to Card, Carry clear for Card to Main.

NOESC (51639) [$C9B7] \SE\ Monitor routine to process normal characters. Checks for copy char (right arrow), literal input, double quotes to turn literal input off/on, and restricted case input before storing in CHAR and returning to caller

NOTCR (64834-64835) [$FD42-$FD43] A change to NOPs of the cursor inverse mode. No longer needed now that the cursor is a standard character.

OLDBASH (2043) [$7FB] \P1\ Pascal saved BASH

OLDBASL (1915) [$77B] \P1\ Pascal saved BASL

OLDCH (1147) [$47B] \P1\ Old CH set for user

128KJMP (49937) [$C311] \P6\ Jump table for 128K support routines

OURCH (1403) [$57B] \P1\ 80 column CH

OURCV (1531) [$5FB] \P1\ Cursor vertical

PASFPT (49931) [$C30B] \P6\ Pascal 1.1 firmware protocol table

PHOOK (51622) [$C9A6] \SE\ Pascal 1.0 output hook

PICK (52993) [$CF01] \SE\ Monitor S/R to read the character at screen position Y = horizontal, returns with character in A

PINIT (51791) [$CA4F] \SE\ Pascal initialization 1.1

PINIT1 (51200) [$C800] \SE\ Pascal 1.0 init

PINIT1.0 (51786) [$CA4A] \SE\ Pascal initialization 1.0

PINIT2 (51793) [$CA51] \L\ Set up for running Pascal, set mode, set window, zero page, check for card, return X=9 (NO DEVICE) if missing, turn on card, set normal lower case mode, home and clear screen, put cursor on screen and return.

PJUMPS (49995-50018) [$C34B-$C362] Pascal jump table

PREAD (51828) [$CA74] \SE\ Pascal input-Get a character, remove high bit, store in CHAR, if 1.1 return "$C3" in X, 1.0 return CHAR in A

PREAD1.0 (51272) [$C84B] Pascal 1.0 input hook

PSETUP (53192) [$CFC8] \SE\ Monitor S/R to set up zero page for Pascal operation. Checks 40-80 columns, sets INVFLG, and updates BASL/BASH before returning.

PSTATUS (51604) [$C994] \SE\ pascal check if ready for input or output, return 3 in X if not ready (ILLEGAL OPERATION)

PWRITE (51854) [$CA8E] \SE\	Pascal output-Set zero page, turn cursor off, check GOTOXY Mode and process if necessary, check if GOTOXY and start if true, else store it on screen, increment cursor horizontal, check if transparent mode and do carriage return/line feed if necessary, replace the cursor and return.
QUIT (52650) [$CDAA] \SE\	Monitor S/R to restore 40 column window,convert 80 to 40 if needed, set cursor at bottom left corner, reset video and keyboard to old mode
RD80COL (49176) [$C018] \H1\	Reads SET80COL
RD80VID (49183) [$C01F] \H1\	Reads SET80VID
RDCARDRAM (49155) [$C003] \H1\	Read RAM on card
RDESC (64801-64808) [$FD21-$FD28]	Formerly KEYIN2, changed to jump to GOTOCX Y=7
RDLCBNK2 (49169) [$C011] \H1\	Reads language card bank 2
RDLCRAM (49170) [$C012] \H1\	Reads language card RAM enable
RDMAINRAM (49154) [$C002] \H1\	Read RAM on mainboard
RDPAGE2 (49180) [$C01C] \H1\	Reads page 1/2 status
RDRAMRD (49171) [$C013] \H1\	Reads RAMREAD state
RDRAMWRT (49172) [$C014] \H1\	Reads BANKWRT state
RDTEXT (49178) [$C01A] \H1\	Reads Text mode
RDVBLBAR (49177) [$C019] \H1\	Reads VBL signal
RESET (64117-64122) [$FA75-$FA7A]	A change in the RESET code to allow for the presence of an 80 column card. Does a JSR to GOTOCX Y=5
SCREEN40 (53047) [$CF37] \L\	Monitor routine to get cursor position, and if V set, branch to STOR40, otherwise read the character from the screen and return.
SCREEN80 (53006) [$CF0E] \L\	Monitor routine to calculate which page, and if V set, branch to STOR80, otherwise read the character from the screen and return.
SCREENIT (52998) [$CF06] \SE\	Monitor S/R/ to either store character on screen or read character from screen. V clear for pick, V set for store, character in A for store, Y = CH position. Saves Y and checks for mode. 40 branches to SCREEN40, 80 falls through to SCREEN80
SCRLSUB (52433) [$CCD1] \SE\	Monitor S/R to scroll only 40 column active window
SCRN48 (52786) [$CE32] \SE\	Monitor S/R to convert 40 column screen to 80 column screen. Moves whole 40 character screen to left most 40 positions on 80 column screen
SCRN84 (52699) [$CDDB] \SE\	Monitor S/R to convert 80 column screen to 40 column screen. Moves leftmost 40 characters to TXTPAGE1
SCROLL (64624-64625) [$FC70-$FC71]	Changed to jump to GOTOCX Y=2
SCROLL1 (52398) [$CCAE] \L\	Monitor routine to check for 40/80 columns
SCROLL2 (52408) [$CCB8] \L\	Monitor routine to scroll 40 columns
SCROLL80 (52416) [$CCC0] \L\	Monitor routine to scroll the other 40 columns
SCROLLDN (52394) [$CCAA] \SE\	Monitor S/R to scroll the screen down one line
SCROLLUP (52388) [$CCA4] \SE\	Monitor S/R to scroll the screen up one line
SET80COL (49153) [$C001] \H1\	Enable 80 column store
SET80VID (49165) [$C00D] \H1\	Enable 80 column video
SETALTCHAR (49167) [$C00F] \H1\	Normal/inverse lower case, no flash
SETALTZP (49161) [$C009] \H1\	Set alternate zero page/stack
SETC8 (50155) [$C3EB] \SE\	Setup IRQ C800 protocol. Stores $C3 in C8SLOT.
SETCH (52911) [$CEAF] \SE\	Monitor S/R to set OURCH and CH. In 40 column mode sets to A value. In 80 column mode, sets to 0 unless less than 8 from end of line, in which case moves up near right

NAME (DEC LOCN) [HEX LOCN] \USE-TYPE\ - DESCRIPTION

SETINTCXROM (49159) [$C007] \H1\ Set internal CX00 ROM
SETSLOTC3ROM (49163) [$C00B] \H1\ Enable C300 slot ROM
SETSTDZP (49160) [$C008] \H1\ Set standard zero page/stack
SETWND (64337-64340) [$FB51-$FB54] A change in the SETWND code to allow for the presence of an 80 column card. Does a branch to GOTOCX Y=8
SNIFFIRQ (64629-64667) [$FC75-$FC9B] IRQ Sniffer for Video Code: A new routine to check the current video mode1 CXROM usage and interrupt status
STOR40 (53066) [$CF4A] \L\ Monitor routine to store the character on the screen.
STOR80 (53034) [$CF2A] \L\ Monitor routine to store the character on the screen.
STORCHAR (52978) [$CEF2] \SE\ Monitor S/R to store character in A at screen horizontal position Y.
TEMP1 (1144) [$478] \P1\ A temporary storage location
TESTCARD (52004) [$CB24] \SE\ Monitor S/R to test for presence of 80 column card, destroys A,Y; returns BEQ if card is there, BNE if not.
TITLE (64226-64269) [$FB0A-$FB0D] APPLE -> Apple
VERSION (64435) [$FBB3] ID code for check on which kind of Apple it is //e=$06 +=$EA =$38
WAIT (52175) [$CBCF] \SE\ Monitor S/R to wait depending on A. Same as F8: WAIT
WRCARDRAM (49157) [$C005] \H1\ Write RAM on card
WRMAINRAM (49156) [$C004] \H1\ Write RAM on mainboard
X.BELL (52156) [$CBBC] \SE\ Monitor S/R to beep speaker, same as F8: BELL1
X.BS (52187) [$CBDB] \SE\ Monitor S/R to execute a backspace
X.CR (52204) [$CBEC] \SE\ Monitor S/R to execute a carriage return
X.DC1 (52569) [$CD59] \SE\ Monitor S/R to set 40 column mode
X.DC2 (52599) [$CD77] \SE\ Monitor S/R to set 80 column mode
X.EM (52237) [$CC0D] \SE\ Monitor S/R to execute HOME
X.FF (52546) [$CD42] \SE\ Monitor S/R to home the cursor. Returns via X.VT to clear screen.
X.FS (52262) [$CC26] \SE\ Monitor S/R to execute a forward space
X.GS (52552) [$CD48] \SE\ Monitor S/R to clear to end of line
X.GSEOLZ (52558) [$CD4E] \SE\ Monitor S/R to clear entire line
X.LF (52369) [$CC91] \SE\ Monitor S/R to execute linefeed
X.NAK (52624) [$CD90] \SE\ Monitor S/R/ to quit 80 column card
X.SCRLRET (52497) [$CD11] \L\ Monitor rotuine to clear top or bottom line (depending on scroll up or down) Return to user via BASCALC.
X.SI (52306) [$CC52] \SE\ Monitor S/R to execute "inverse video"
X.SO (52297) [$CC49] \SE\ Monitor S/R to execute "normal video"
X.SUB (52250) [$CC1A] \SE\ Monitor S/R to execute clear line
X.US (52276) [$CC34] \SE\ Monitor S/R to execute a reverse linefeed
X.VT (52515) [$CD23] \SE\ Monitor S/R to clear to end of page
XCOORD (1787) [$6FB] \P1\ X coordinate in GOTOXY routine
XFER (50096) [$C3B0] \SE\ Transfer program control from main board to card or vice versa. $3ED-$3EE is address to be executed upon transfer, carry set means transfer to card, carry clear means transfer to main board, V flag clear means use standard zero page/stack, V flag set means use alternate zero page/stack. Also uses $3ED-$3EE in destination bank. Enter via JMP not JSR.
XGOTOCX (64626-64628) [$FC72-$FC74] A JMP to GOTOCX for long branching purposes
YSAV1 (31) [$1F] \P1\ Temporary storage for the Y register

Announcing Other Works on the Apple Computer from

MICRO INK

MICRO on the Apple

A series of books for Apple users

Each volume in this series presents the best Apple articles from MICRO magazine in an integrated collection, plus additional material that has never before been published. Articles and programs have been updated by the original authors or MICRO's staff. All programs have been tested and entered on a diskette which comes with the book. Each volume in the series is 6 × 9 inches, with approximately 224 pages, and includes a pocket for storing the diskette. A Wire-O binding allows the book to lie flat when open.

[Since these programs were all written for the Apple II, some of them may not work on the Apple //e.]

Volume 1 is currently out-of-print.

Volume 2 contains:

Breaker: An Apple II Debugging Aid
Step and Trace for the Apple II Plus
Tracer: A Debugging Tool for the Apple II
Integer BASIC Subroutine Pack and Load
MEAN 14: A Pseudo-Machine Floating Point Processor for the Apple II
Screen Write/File Routine
Bi-Directional Scrolling
Ingeger BASIC Program List by Page
Paged Printer Output
Hexadecimal Printer
Common Variables
PRINT USING for Applesoft
Searching String Arrays
Applesoft and Matrices
AMPER-SORT
Trace List Utility
Versatile Hi-Res Function Plotter
Hi-Res Picture Compression
An Apple Flavored Lifesaver
... plus 12 additional articles

(Diskette in 13 sector DOS 3.2 format)
(May be 'muffined' to 16 sector format)

Volume 3 contains:

Applesoft Line Finder Routine
Amper-Search
Applesoft Variable Lister
Double Barrelled Disassembler
Cross Referencing 6502 Programs
A Fast Fractional Math Package for the 6502
Applesoft Error Messages from Machine Language
Serial Line Editor
Trick DOS
LACRAB: Formatted Program Listings
Apple Color Filter
True 3-D Images
Apple Bits: Low Resolution Graphics
Apple Byte Table
How Microsoft BASIC Works
Simple Securities Manager
Solar System Simulation
Reversi Game
Musical Duets

(Diskette in 16 sector DOS 3.3 format)

The price of each volume (including the diskette) is **$24.95**.

For shipping and handling on mail orders, add $2.00 for surface shipping.
Massachusetts residents add 5% for sales tax.

MICRO INK
P.O. Box 6502
Chelmsford, MA 01824
Telephone: (617) 256-3649

VISA and Mastercard accepted.

What's Where in the Apple

A Complete Guide to the Apple Computer

*** Apple II * Apple II Plus * Apple //e ***

Every Apple user needs this book! The original *What's Where in the Apple?* provided more information on the Apple's memory than was available anywhere else. Now this REVISED EDITION shows you how to use this valuable data.

Providing both a numerical Atlas and an alphabetical Gazetteer, *What's Where in the Apple...Plus...* guides the user to over 2,000 memory locations of PEEKs, POKEs, and CALLs.

The names and locations of various Monitor, DOS, Integer BASIC, and Applesoft routines are listed, and information is provided on their use.

The easy-to-read format includes:

- The address in hexadecimal (useful for assembly programming) $FC58
- The address in signed decimal (useful for BASIC programming) ..... (-936)
- The common name of the address or routine ..... [HOME]
- Information on the use and type of routine ..... \SE\
- A description of the routine ..... CLEAR SCROLL WINDOW TO BLANKS.
  SET CURSOR TO TOP LEFT CORNER
- Related register information ..... {A- Y-REGS ALTERED}

Applesoft and Integer BASIC users will find information which will speed up and streamline programs. Assembly language users will gain access to routines which will simplify coding and interfacing. Both BASIC and assembly language users will find this book helpful in understanding the Apple II, and essential for mastering it! (ISBN: 0-938222-09-0)

$19.95 in U.S.

About the Author

William F. Luebbert is adjunct Professor of Engineering at Thayer School of Engineering, Dartmouth College, Hanover, New Hampshire. He is also president of the Computer Literacy Institute, an organization founded in 1980 to train educators in the uses and applications of computers in education.

Professor Luebbert, now a U.S. Army retired Colonel, served on the faculty of the U.S. Military Academy, West Point, New York, from 1960 to 1978, where he taught Electrical Engineering and headed the Academic Computer Center.

He has received the Automation Educator of the Year Award from *Business Automation Magazine*, the Certified Data Processor Award from the Data Processing Management Association, and the American Society for Engineering Education Award and Prize for excellence in teaching engineering students.

MICRO INK
P.O. Box 6502
Chelmsford, MA 01824