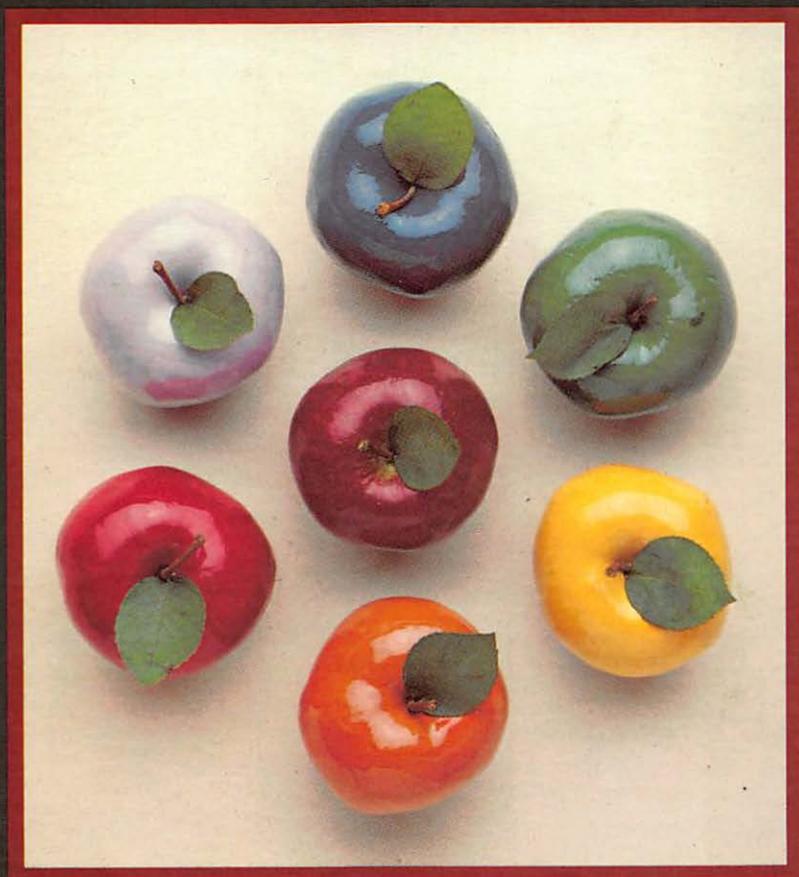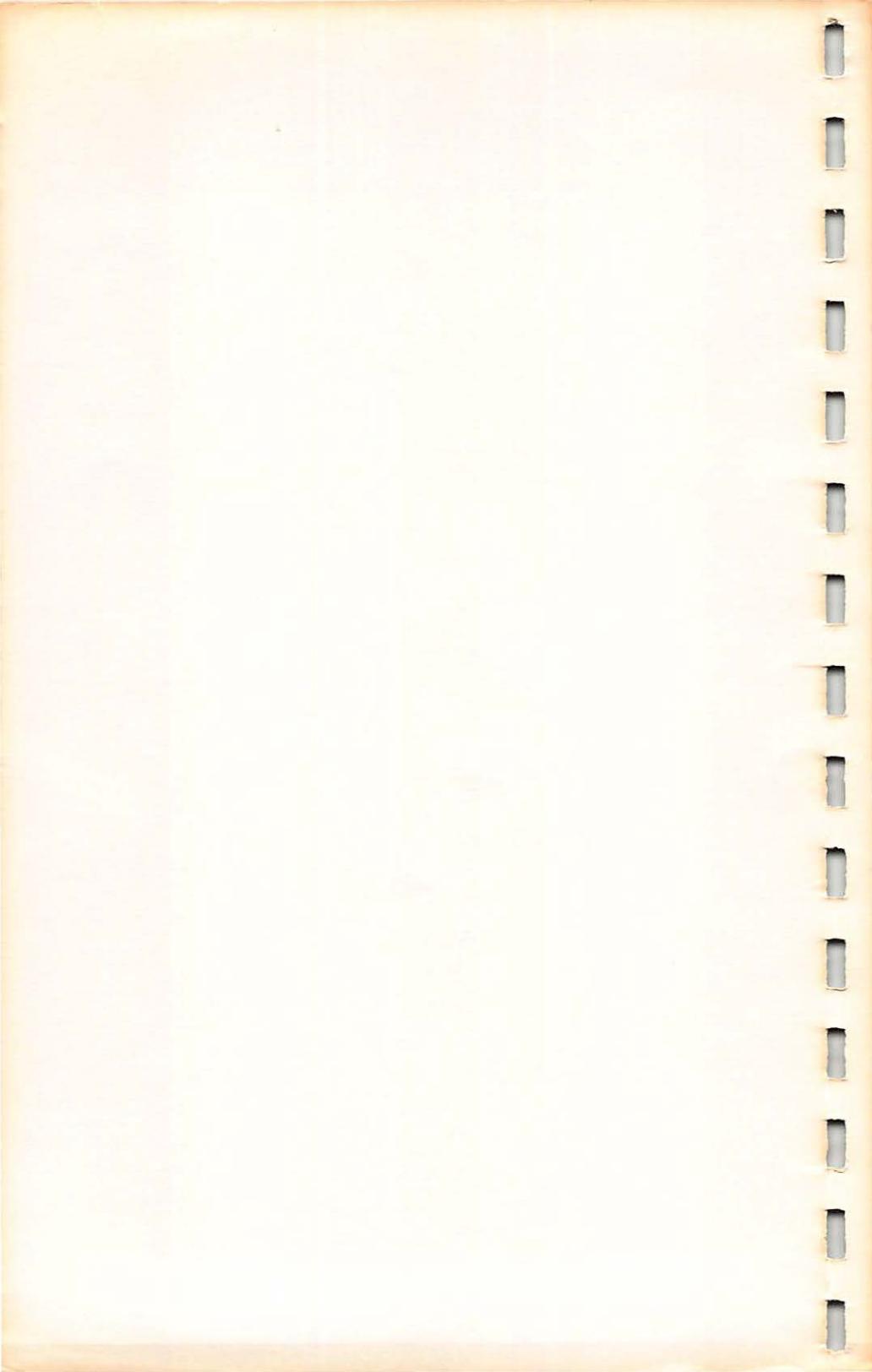# Apple®
Programmer's
Handbook

Paul Irwin

Apple® Programmer's Handbook

Paul Irwin is a computerist who writes, teaches, and consults in Ottawa, Canada. His computer experience ranges from large mainframes used in government administration to small microcomputers used in scientific research. He holds a B.Sc. in Mathematics and Physics from Laurentian University of Sudbury and a certificate in Computer Programming from Algonquin College (Ottawa). For recreation, Mr. Irwin enjoys skiing, swimming, and sailing.

Apple® Programmer's Handbook

by
Paul Irwin

Preface

In 1977, the Apple II computer was launched on a dealership basis across the United States. Prior to that time, Apple was a two-man garage operation making single-board computers for the hobbyists on an order-by-order basis. This hobby market saw the Apple II as cheap, flexible, and somewhat funky. It consisted of a lot of hardware like color graphics, built-in I/O, on-board bus sockets for peripherals and so forth; but, there were few software features.

A manual came with the Apple II. This became known as the *red-book* and it was as funky as the Apple itself. With startup hints, Integer BASIC description, game instructions and assorted listings, commands and schematics, it gave the dedicated computer hacker a beginning with a wonderful new toy.

But Apple found money — lots of money. So, like Cinderella, the Apple II became beautiful. A disk with an operating system was developed. Microsoft BASIC became Applesoft BASIC in a special Apple version. The Monitor that works between most software and the hardware was modified to start the disk automatically at powerup. Called Autostart, this Monitor was fitted to Apples with Applesoft in ROM instead of Integer BASIC. The new version, different only in this firmware, became the Apple II Plus.

Meanwhile, the cost of RAM dropped. Many of the early Apples were sold with only 4K of RAM; today, few of these remain with less than 48K RAM in them. An extension card with an additional 16K of RAM became popular and gave the Apples a RAM complement of 64K. With this additional RAM, users ran other languages like Pascal or simply enjoyed the ability to switch from Applesoft to Integer BASIC at will.

The disk system changed during this period as well. The hardware stayed almost the same — shielding on the ribbon cable to reduce radio interference was the main modification. The disk firmware and software changed to provide sixteen sectors per track instead of the original thirteen. This new version called DOS 3.3 gives almost double density capacity to the 5¼-inch diskettes.

In 1983, the Apple IIe replaced the Apple II Plus. It is compatible with the earlier models and adds several of the features found in common Apple customizations: full ASCII keyboard, lower-case display, and the ability to switch to an 80-column screen display. The Monitor modifications needed to do this were Herculean. It works and it works well. The few programs that can't coexist with the IIe model are being superseded by others — the IIe model proved to be a success.

However great the changes to the Apple II itself, the most powerful changes came from outside Apple Computer Inc.

The most powerful feature of the Apple II is the built-in peripheral bus. Unlike much of the competition of the day, Apple published Monitor source listings and schematics. With the listing and bus pinouts, hundreds of peripheral boards were designed and built. The Apple II became an *open system* in the truest sense. With an Apple II anyone could configure his own custom computer. If a remote terminal, a data logger, a word processor, a video game, or a super calculator was wanted the answer was the same — get an Apple. This is still true today. You can have all the computers you want by adding reasonably priced peripheral boards to your Apple.

This book is for the people who have these Apples. Students, hobbyists, accountants, engineers, scientists, and artists who need specific information can use it to look up just what they want to know. The organization is top down; each topic is treated with specific examples presented in increasing order of depth. For instance, the Applesoft statements you need to address the screen cursor are given before the Assembler statements to do the same thing. All these routines have been tried and tested true; many developed over a period of Apple II programming of two and three years.

This book can also be used for self-study by the Apple II user. The top down organization allows the development of concepts from known to unknown. It will be useful in any related course: computer programming, computer science, systems analysis, digital electronics, etc.

Chapter One introduces the Apple II with details not emphasized in the manuals. An overview is kept to help you envisage your system in terms of your requirements. Programming is restricted to BASIC and stresses data management needs.

The words Apple, Apple II, Apple IIe, Apple II Plus, and Applesoft are registered trademarks of Apple Computer Inc.

Chapter Two is a condensed atlas, emphasizing the maps and locations most often needed by the Assembler programmer.

Chapter Three can be used alone or with a tutorial text to learn Assembler programming. Those with experience will find useful routines and methods they may add to their repertoires.

Chapter Four to Seven need some Assembler programming experience to understand and use. The BASIC programmer can find some command definitions and specific usages, however.

Chapter Eight is for the hardware freaks. Some hardware background is needed to build the projects given and suggested there. Hints and cautions to the novice are included to encourage the beginner.

Use this book to make your Apple II into the many custom computers you want it to be.

PAUL IRWIN

Acknowledgments

To all those who gave so freely of their time and talent to make the first Apples, especially to Stephen Wozinak, this book is dedicated. Thanks, Woz.

Contents

CHAPTER ONE

Getting It Together

1.1 REQUIREMENTS

1.1.1 Hardware

The Apple II is supplied with a case, a power supply, a keyboard, and a motherboard. You add a video monitor or a tv set with an rf modulator, a disk drive with a controller card, and possibly a tape recorder. If you wish, you can get a motherboard separately, without the Apple firmware. But, however you acquire your Apple, you will probably end up with a disk system operating from version DOS 3.3 or later. The BASIC language supplied is Applesoft except in the older Apples. Before delving too deeply into your Apple's internals, you should be able to program with Applesoft BASIC and have a feel for some of the computer's abilities. To begin with, look at the major parts of your system first to see what to expect from them.

First, the motherboard. Fig. 1-1 shows the motherboard and connector locations. This is the Apple II proper, a complete microcomputer. You can buy it separately without case, power supply, or keyboard for special installations. Apple Computer Inc., supplies *the real thing*, but some *equivalent* boards offered by several other suppliers come without Applesoft or Apple's other firmware. These so-called clones give you a motherboard that will run Visicalc™ and other software that is independent of the host firmware, but you must still acquire Applesoft and a Monitor before you have an equivalent

TOP VIEW

PERIPHERALS

CASSETTE DATA IN
CASSETTE DATA OUT
VIDEO OUTPUT

POWER
CONNECTOR

0   1   2   3   4   5   6   7

K12 K13 K14

AUXILIARY
VIDEO OUTPUT
CONNECTOR

J14B

K1

J

GAME I/O
CONNECTOR

J2 J4 J5 J6 J8 J9 J11 J12

J14

B

SPEAKER
CONNECTOR

A7

B14A

KEYBOARD
CONNECTOR

A

1 2 3 4 5 6 7 8 9 10 11 12 13 14

FRONT OF PC BOARD
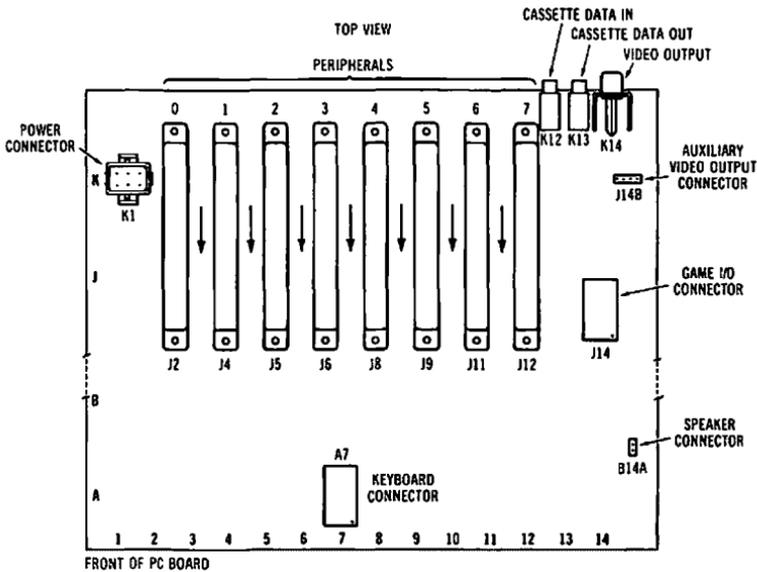
Fig. 1-1. Connector locations on Apple II motherboard (except for Model IIe).

computer. Even then, hardware differences may give you differences that some Apple programs won't know about. Like missing colors or special terminal defaults. At the time of writing this book, several motherboards offered in the underground market have poor quality with circuit traces that are lifting from the board either during home assembly or afterwards in the case of preassembled boards. Caveat emptor!

Clones that are assembled and supplied in cases are the safest. You can run them in the dealer's store and see if it handles the features you need. And, by asking around at computer club meetings, you can find the dealers with happy customers. With full 64K of RAM, you can purchase DOS 3.3 from Apple and have both Applesoft and Integer BASICS on disk to load in place of the normal Applesoft/Monitor firmware. Or you can program custom systems if you are good at machine language and can make your own PROMs to plug in. Most clones provide 2716 or 2732 sockets for this purpose. Be careful of BASIC and Monitor PROMs on the black market; selling or buying copyright material is illegal without permission from the owners (Apple Computer Inc., and Microsoft).

The power supply from Apple is a high quality switching type. It supplies 5.0 volts at 3 amperes, and this is the minimum size you should accept for most systems. There are 5-ampere supplies available for applications that you may have with a lot of peripheral cards. Be wary of *lightweight bargains* that won't put out 3 amperes; better pay a bit more for the 5-ampere supply if you are unsure of your future needs.

Keyboards also vary in quality. Apple has always supplied good keys with a quiet, yet distinct sound and a pleasant feel. They are supplied by Cherry and should be the standard by which you compare any other keyboard offered. The first Apple IIs had a *live* RESET key that was often struck in error because it is close to the RETURN key. Later Apples interlock the RESET with the CTRL key so both must be pressed to reset the processor. If you have one of the older Apple IIs, Chapter Eight contains a simple method to interlock the CTRL key.

For a video display, connect a video monitor to the VIDEO OUTPUT jack (K14, see Fig. 1-1). If you use a tv set as the monitor, you must install an rf modulator inside the Apple II case. Fig. 1-2 shows this installation.

Until the Apple IIe model, lowercase was tricky to implement. The keyboard itself wouldn't generate lowercase characters and the normal Monitor routine that gets character lines from the keyboard compounded the problem by converting all characters to uppercase. Despite these handicaps, several lowercase schemes are available for the Apple II, usually with a replacement character display ROM.

A great deal of elaborate software is available for the Apple II on disks. The 5¼-inch floppy disk is a great improvement over the cassette tape and is now almost universally used. The Apple II uses a stripped-down version of the *Shugart 440* drive with a controller card that plugs into a peripheral slot (usually Slot Six). With the controller card you need a disk copy of the Disk Operating System, or DOS. Version 3.3 is current at the time of this writing; earlier versions used different controller card firmware and had less disk storage capacity. A controller card will handle two drives so you can add a second drive without getting another card.

WARNING

To avoid damage to the disk drive see Fig. 1-3 for proper cable connections.
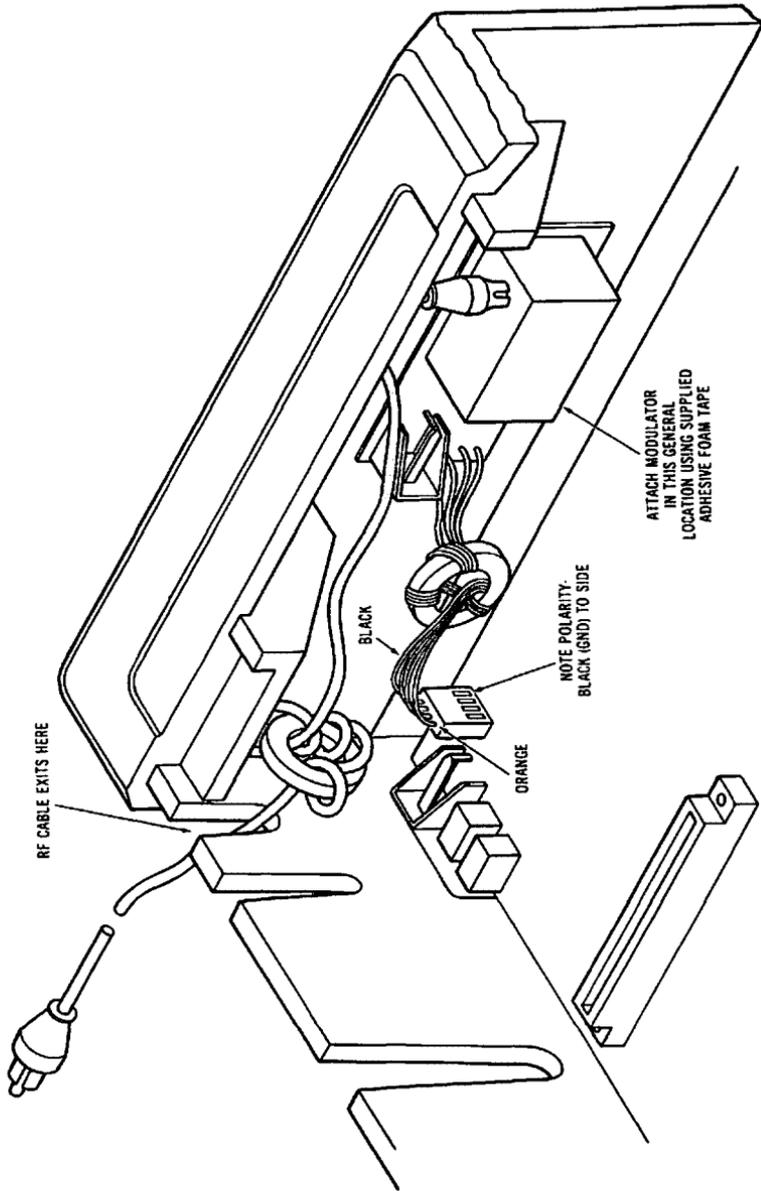
Fig. 1-2. RF modulator installation inside the Apple II case. (Courtesy M & R Enterprises)

ARROW SHOULD BE POINTING UP
WHEN CONNECTOR IS INSTALLED

You can permanently damage the disk drive if you attach
the cable incorrectly. Here is how to attach it correctly.
1. The side cable leaves the connector on the
   side away from the card.
2. Make sure all pins go into their matching holes.
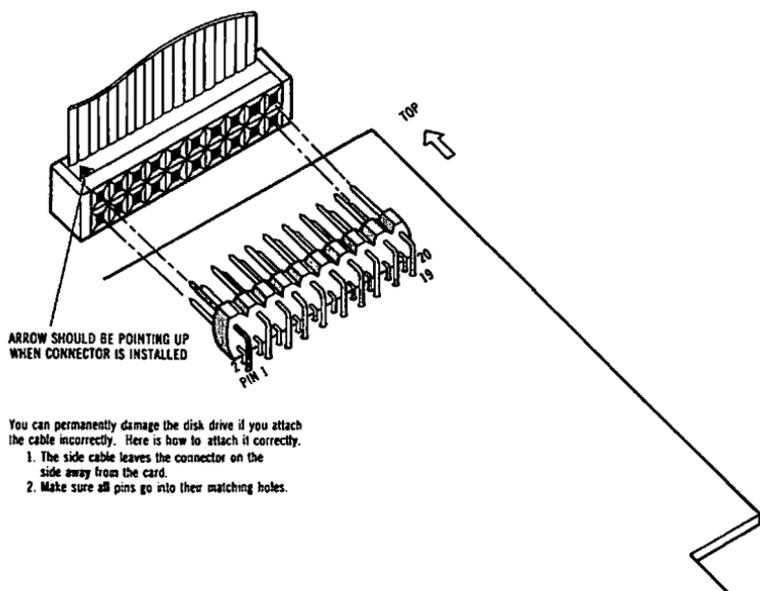
Fig. 1-3. Correct method for connecting the disk cable.

You may also want a tape cassette recorder. Only a few disks are
usually needed at a time; the rest of your software can be stored on
tape. A *C-60* tape will hold the files from four disks, making tape
storage quite a bit cheaper. In addition to archival storage, tape can be
used to back up your working disks. Tape is slow, but it is cheaper.
See Chapter Eight for more details on using tape.

To make your Apple do things besides talk to the tv set, you will
need various peripherals: printer, modem, etc. The hardware and
firmware needed to make the interface with each peripheral comes on
a card that you plug into one of the eight sockets or *slots* on the
motherboard. So, a printer may use a parallel interface card and a
modem may use a serial interface card; each card is designed specif-
ically for the Apple. Some peripherals are designed especially for the
Apple and come with their own cards. You must decide which card to
put into which slot.

According to Apple, you can use any card from any slot with the ex-
ception of Slot Zero, the leftmost slot. The Apple is called *slot inde-
pendent* in its input-output system for this reason.

The leftmost slot — Slot Zero — is special. Cards must be specifi-
cally designed for that slot and usually are memory expansion cards.

The 16K-RAM card is common, but ROM cards and even $R-G-B$ video display cards have been used there. The Apple IIe model does not have a Slot Zero. The IIe acts like it had a 16K-RAM card instead; the actual 16K of RAM is already on the motherboard. If you have an Apple other than the IIe model, you can put a 16K-RAM card in Slot Zero, if it is otherwise unoccupied. Table 1-1 lists the recommended card uses for the remaining seven slots.

Table 1-1. Recommended Slots

Slot	Description
0	16K RAM card (not on IIe)
1	printer
2	modem/communications
3	80-column terminal (use Aux Slot on IIe)
4	
5	second disk controller
6	disk controller
7	

The disk controller card usually goes in Slot Six. It will work in any slot, but most software assumes Slot Six has the disk card. This slot is your choice for the disk card by convention.

Another convention grew around Slot Three. Earlier users began plugging serial cards into Slot Three to connect 80-column terminals to RS-232 interface cables. This convention became the standard for the Pascal Language System. Later when the IIe model came out Slot Three had special hardware as well. In the IIe model, Slot Three can be used like any other slot, just like earlier models. But if you use the built-in 80-column display instead of the normal 40-column display, then it changes. You can't use Slot Three when using the built-in 80-columns because the extended display uses Slot Three from the motherboard. A special slot called the Auxiliary Slot accepts the extra memory the 80-columns need; the Auxiliary Slot is a second Slot Three with special connections. So, use Slot Three for 80-column display cards, or use the Auxiliary slot on the IIe, or use Slot Three for anything else in 40-column applications. You should choose only one of these three cases for your own Apple.

Slot One is usually used for a printer. If you are connecting a printer, a parallel or serial interface card is plugged into Slot One. Some software assumes your printer is connected to Slot One.

If you plan on doing any hardware work on your Apple, you'll need some special tools. You may have most or all that you need already, but if you are starting from scratch, the lists in Table 1-2 will help.

Table 1-2. Hardware Tools

Start with connections:
VOM multimeter Soldering iron and holder Desoldering braid and sucker 4-inch diagonal pliers 4-inch long-nosed pliers Set of jeweler's screwdrivers 3/16-inch x 4-inch slot screwdriver #1 x 4-inch Phillips screwdriver Alligator clips for heat sinks Old toothbrush for cleaning Small stuff: solder, heatshrink, hookup wire, etc. Silicone Seal™ and Epoxy cements
Add circuit board tools:
Wire-wrap tool and #30 wire (OK WK-2-B) Modified wire stripper (for #30 wire) LED logic probe, homemade Carbide tip scriber X-acto knife 16-pin IC clip Debounced TLL pushbuttons, homemade Files, 6-inch bastard: flat, round, triangular
Enlarge chassis and cabinet ability:

Shears, 10-inch compound aircraft type
Nibbling tool
Hacksaw with blades
Machinists' square with 12-inch rule
Files, 10-inch bastard: round and half-round
File, 10-inch flat mill
File handle for 10-inch files
Electric drill, ⅜-inch with variable speed
High-speed twist drills, as required.
¼-inch x 6-inch slot screwdriver
8-oz ball-peen hammer
Large vise or Work-Mate™
Assortment of materials: sheet aluminum, self-tapping screws,
    styrene plastic, wood, machine screws and nuts, PCB stand-
    offs, etc.

First thing you need is the ability to work with cables and con-nectors. You will need a soldering iron, pliers, small screwdrivers, and maybe a couple of small files. Get a supply of small-diameter heat-shrink tubing to dress solder connections on terminal pins. To test the circuits you should have a multimeter or a continuity tester.

If you want to make peripheral cards using the methods given in Chapter Eight, then consider wire-wrap. It is easy to use and you can make modifications without the problems of desoldering. Wire-wrap tools for AWG 30 wire are available from several sources. The OK Wire-Wrap kit (WK-2-B) comes with a wrapping tool and an assort-ment of prestripped wire. Wrapping tools are available separately from Radio Shack and others. If you do much wrapping, get a better wrapping tool that will *daisy chain* by stripping and wrapping a con-tinuous wire to several posts. Also, the simpler hand wrapping tool has a built-in stripper that nicks the wire horribly. Get a pair of wire strippers especially for wire-wrap work, file the setting down to make a 30-guage stripper, and then cement the setting wheel in place.

Boards for wire-wrapping are available from Vector, Apple, and other manufacturers. They have + 5.0 volt and ground buses on them; just add wire-wrap sockets using small dabs of black Silicone Seal™ or a hot glue gun. Then solder 0.1 μF decoupling capacitors near each socket and perhaps a 1.0 μF tantalum capacitor near the pins across the 5 volts.

You will need some test equipment, even if it is only a multimeter. A common LED in series with 220 ohms can be mounted in a discarded ballpoint pen case to make a logic state probe. If you work with TTL, a couple of debounced pushbuttons and a 16-pin IC clip will make life easier. You must check the continuity of all connections before plug-ging in the ICs to avoid possible disasters. Remember that if you can't test something, then you can't get it to work.

Most tests won't require an oscilloscope. For those that do, you may not need a large bandwidth. Unless you do a lot of hardware work, you probably won't have to purchase one.

Large hardware projects need chassis and cabinet work. At this stage, you can add a drill set, large files, a nibbling tool, and a supply of sheet aluminum, screws, plastic, and decals. A vise or Work-Mate™ is a must.

There are enough goodies on the market so that there is little need to roll your own hardware. But if you have the tools on hand you can save money over several small projects to pay for the tools. Some large

projects are available in kits so you can save assembly and testing costs. Aside from these reasons, tools let you make those rare items you may want in specialized applications, such as scientific research. All it takes is a few hand tools and a little creative talent to extend your Apple's hardware abilities.

1.1.2 Firmware

When an Apple II is powered on without a disk operating system to bootstrap, a programmed machine routine is used. You can program in BASIC because an *interpreter* program remains in the Apple, even during power off. In addition to a BASIC interpreter, the Apple II has a special program called the Monitor that allows you and the BASIC interpreter to work with the hardware. These special programs that reside in ROM (read-only memory) are collectively called the Apple II's *firmware*. These programs can't be lost even during power off.

You can work directly with the Apple II's machine language without going through BASIC by interacting directly with the firmware Monitor. The Monitor has a set of routines that allows you to access the machine language programs, so you can change the contents of memory locations, copy data around in RAM, and perform other functions that can seem like magic to the uninitiated. From BASIC you reach the Monitor with a CALL − 151 command; to go back to BASIC you type ctrl/C. A summary of the Monitor commands appears on the Apple II Reference Card included with this book.

The Monitor may be one of three versions. The old Standard Monitor is the easiest to use. On power up, it displays an asterisk − * − as the prompt, unlike the later Autostart and IIe Monitors that run BASIC at power up. The asterisk is the prompt for the Monitor commands which you get whenever you CALL, 151 from BASIC. If a disk controller card is in any slot, the Autostart and IIe Monitors will attempt to bootstrap a disk, requiring you to type a RESET to override that feature if you don't have a disk mounted. When the Autostart features were added: the old Monitor S (step) and T (trace) commands were deleted, the RESET function expanded to force a disk bootstrap at power on, and the ESC cursor control keys I, J, K, and M were made available. The greatest difference in the three versions is in the way each handles RESETs. The Standard Monitor treats them all the same; the Autostart lets programs use the keypress RESET and handles it differently than a power on RESET. The IIe model has a

Monitor similar to Autostart, but with more RESET modes, resulting in the addition of OPEN-APPLE and CLOSED-APPLE keys, and the automatic RAM program destruction on a keypress RESET.

Practice in using the Monitor makes learning machine-language Assembler programming easier. A summary like the Reference Card in Appendix B will help. Play with it; just don't leave any disks mounted and you won't do any harm. Power off or disconnect any peripherals that may accidentally be activated. You should be able to examine any block of memory, move the contents of a block of memory to any block of RAM, and to disassemble programs in machine language. The range $1000.1FFF is a nice "safe" RAM destination for practice. The Monitor begins at $F800 and you can disassemble there to see how the L (list) command works. Specific activities with the Monitor appear beginning in Chapter Three.

In addition to commands the Monitor has routines that handle the keyboard, the video display, the cassette tape recorder, and the interface with other inputs and outputs. These routines are used by BASIC and directly by software written in Assembler. You can use them in your own programs with PEEKs, POKEs, and CALLs or with Assembler routines.

Just as there are three versions of the Monitor, there are two kinds of BASIC for the Apple. Applesoft BASIC is the most common today, but older Apples have Integer BASIC as the firmware BASIC. Many of these older Apples have Standard Monitors as well, but not all. When fitted with Applesoft instead of Integer, the Apple II is called an Apple II Plus. From the factory, a special label appears on the lid, but with a used machine, you can't go by that. The Apple II Plus with Autostart Monitor is the most common Applesoft BASIC arrangement before the IIe model. The IIe acts much like a Plus model; it has Applesoft and Autostart Monitor features.

With regard to the two BASICs, Applesoft is more comprehensive, with built-in functions and floating-point math, and has more similarities to other Microsoft BASICs. On the other hand, Integer BASIC is faster and allows longer variable names. Many functions can be found in the Programmer's Aid #1 ROM chip that can be used with PEEKs, POKEs, and CALLs. Of importance to machine language programmers is the miniassembler program that is in the Integer BASIC ROM. Early Apple II programs were written in Integer BASIC, but most of the later BASIC programs are in Applesoft. This book assumes Applesoft as the resident BASIC, but you can find

some Integer usage and examples, especially in Chapter Five. A description of how Applesoft works is in Chapter Four. Programming techniques are in Section 1.2 of both chapters. Your normal choice is Applesoft.

Normally, there is only one chunk of memory for the firmware. However, it is possible to relocate the resident firmware in the ROMs with "soft" firmware from a disk. To do this, an Apple II or an Apple II Plus must have a 16K RAM card plugged into Slot Zero. The Apple IIe already has this extra memory on its motherboard. This memory is called the *bank-switched memory* because the bank of ROMs can be replaced by a bank of RAM with a memory address switching circuit. By loading a binary file containing the Monitor and an alternate BASIC into the RAM, the personality of the Apple changes from an Applesoft to an Integer BASIC machine. Or, an old Apple can switch from its resident Integer BASIC to a disk-based Applesoft. After the second BASIC has been loaded, the DOS commands FP and INT will switch from one type of BASIC to the other on command.

The System Master disk that comes with DOS 3.3 has the binary files, INTBASIC and FPBASIC. When the HELLO program runs, one or the other BASIC is BLOADed into the bank-switched RAM.

Other systems can also be loaded into the bank-switched RAM instead. The *Apple Pascal Language System* is an example. A Pascal bootstrap loads the P-code interpreter and the kernel of the Pascal Operating System into the bank-switched RAM. Alternately, you can choose from among languages like Logo, Fortran, Lisp, Forth, and Visicalc that are available from Apple or from other vendors.

Once booted the language is usually write-protected. This provides the benefits of firmware while retaining the features of software on disks.

1.1.3 Software

While much of the software available for the Apple II is specific to a single task or application, many programs can be called utilities. A utility has no specific application, but it lets you use the Apple in a certain way.

The utility you use often is an editor. The Monitor supports an editor to handle Monitor commands and BASIC statements. And you can have word/text processors that allow you to write letters, contracts, manuals, research papers, articles, books, flyers, or whatever.

For programming, you use a type of editor called a *line editor*.

The Apple Monitor has a line input that is the heart of the built-in programming editors. It lets you enter a line of text before turning it over to BASIC or to its own interpreter. You type in what you wish, use the forward and backward arrows for corrections, and then press RETURN to enter the line. You can quit the line at any time with a ctrl/X. As a minimum editor it works well but there are better ways to program.

For instance, when you change previously entered Applesoft lines, the extra spaces at the right of the screen mess up any strings that are within quotes. The old trick of typing POKE 33,33 helps, but before long a more sophisticated line editor is often needed.

One solution is an extension to the built-in editor called *Program Line Editor*. It gives you cursor search and edit features one line at a time, as well as a listing control for Standard Monitors without ctrl/S. You can add your own commands as well. See Appendix B.

The best solution is to get a text file editor like the ones that come with an Assembler. This way you get an Assembler, with an editor as a bonus, that you can use right away, and you don't have to know Assembler programming to make good use of the editor for BASIC programming. Then when you get into Assembler, you have a familiar editor to use.

Such a text file line editor will give you the features you need. It will let you enter lines and will number them for you. You can search for line numbers or for text strings; then the search lets you replace or list what it finds. Separate instructions are included to run the Assembler. In this book, all examples of Assembler routines are written for the *DOS Toolkit Assembler* from Apple.

To load a BASIC program written as a TEXT file, you use the EXEC command instead of the LOAD command. This reads your text in just as if you were typing it at the time. To make corrections to the disk copy, however, you will have to go back to your line editor and make the correction to the TEXT file.

If you have a program already in a BASIC file, but want to convert it to a TEXT file for editing, use the CAPTURE routine. Make one by creating a one-liner with your text line editor:

```
0   PRINTCHR$(4)"OPENXXXX":PRINTCHR$(4)"WRITEXXXX":
LIST1,32767:PRINTCHR$(4)"CLOSEXXXX":END
```

Save it as CAPTURE.

To use CAPTURE, *search* and *replace* XXXX to your program's text file name. For example, your program in BASIC called PHASORZAP could have a text file name of PHASORZAP.TEXT. Save this version of CAPTURE as, say, CAPTURE.PHASORZAP. Quit the editor and load the BASIC program, PHASORZAP. Then EXEC CAPTURE.PHASORZAP to add the capture line to the program in memory. Typing the RUN command will cause the text file called PHASORZAP.TEXT to be created on disk. Notice that you can't use Line Zero in any of your BASIC programs for any other purpose or this trick will wipe it out.

In addition to editors, several other utilities are useful. You will want them to help in program development and maintenance of your files.

You can use various *disk zaps*. FID is the most common one. The COPY program on the System Master disk will back up or copy entire disks, not just files like FID. And MUFFIN will copy files from earlier DOS disks in thirteen-sector format to the DOS 3.3 sixteen-sector format disks. For advanced use, a disk-zap utility for working with disk sectors is given in Chapter Seven.

Such sector-type disk zaps are quite powerful. They have different names; the one in Chapter Seven is just called DISK ZAP. They let you read and write from the disk by *track* and *sector numbers* and examine the bytes of data within each 256-byte sector. You can change the bytes and replace the altered sector on disk. You use this utility to recover crashed disks, find *hidden* files, see special characters on disk, and learn how DOS works. You can create special disks and customize DOS. See Chapter Seven for details.

If you do a lot of Assembler programming, get a *debugger,* sometimes called DDT. This will give you step and trace capability, let you set breakpoints, examine both registers and memory, and other debugging routines. It may come with your Assembler, but there are separately supplied debuggers available. They can make your program debugging much easier.

COPY programs are available besides the one on the DOS 3.3 System Master disks. Use it or another COPY utility to keep your daily work backed up. Copy program disks when you get them and use the copy as the working disk; keep the original archived in case

you lose your working copy. A more powerful COPY routine may work with an *uncopyable* disk supplied in a nonstandard disk format. Many manufacturers use different schemes to protect their interests against software piracy. Unfortunately, this makes it difficult for you to have backups.

One solution is not to buy *uncopyable* disks. They are fragile and often expensive. Another answer might be to find the scheme used to alter the DOS and write a special COPY routine. This takes time and skill. A third solution is to write your own version of the program you want. These programs are not always as complicated as they appear; much time and effort in commercial packages goes into the *bells and whistles* to make them look slick.

If you simply must have the program to use and don't have the time to develop an alternative, then buy it. Often you can obtain a replacement disk during a warranty period by sending in the original. Some suppliers will replace out-of-warranty disks for a fee.

1.2 PROGRAMMING

1.2.1 BASICs

Applesoft has an extensive command set. Of these, a few are learned very quickly, some are learned only by a few people, and some aren't learned at all. If you have practiced with Applesoft or taken a course, you are familiar with the common ones. Specialized commands are simple to understand and use because of their specific nature. For example, the SPC(function inserts spaces when used with a PRINT statement. The few remaining difficult ones are rarely used, but are quite useful. See Table 1-3.

One pair of commands that you should play with is the TRACE and NOTRACE. They are BASIC line number tracing features that let you see where your program is going as it runs. You can use them easily when debugging.

A little-used command is the WAIT instruction. It does bit testing that lets you examine hardware locations and then waits until the device being tested does something. Such a device may be the keyboard. For example, you WAIT for a keypress, or you could WAIT for a pushbutton. Some uses for the WAIT instruction appear in this book, but there are others.

The ONERR GOTO . . . statement is used, but not often enough. All programs that access disks should use this error-trapping feature

Table 1-3. Applesoft Command Set

Program Flow	Input/Output
&	GET...
CALL...	IN#...
DEF FN...	INPUT...
END...	LOAD
FOR...=...TO...STEP...	PDL (
GOSUB...	PEEK (
GOTO...	POKE...
IF...GOTO...	PR#...
IF...THEN...ELSE...	PRINT...or
	?...
NEXT...	RECALL...
ONERR GOTO...	SAVE
POP	SHLOAD
REM...	SPC (*
RESUME	STORE...
RETURN	TAB (*
SPEED...	WAIT (
STOP	
USR (	*-used only in PRINT

Screens (text,LORES,HIRES)			Variables Control
FLASH	COLOR =...	DRAW...	CLEAR
HOME	GR	HCOLOR...	DATA...
HTAB...	HLIN...	HGR	DIM...
INVERSE	PLOT...	HGR2	FRE (
NORMAL	SCRN (	HPLOT...	READ...
POS (	VLIN...	ROT =...	RESTORE
TEXT		SCALE =...	
VTAB..		XDRAW...	

Math and String Functions				Edit and Debug
ABS (	EXP (	MID$ (	SQR (	ctrlC, ctrlX, and reset
ASC (	INT (	RIGHT (	STR (	CONT
ATN (	LEFT$ (	RND (	TAN (	DEL...
COS (	LEN (	SGN (	VAL (	HIMEM:...
CHR$ (	LOG (	SIN (		LIST...
				LOMEM:...
				NEW
				NOTRACE
				TRACE
				RUN...

Table 1-3 – cont. Applesoft Command Set

Assignment Symbols
LET ( ) = + –
AND * / ∧
OR
NOT

of Applesoft. And don't forget about arithmetic and syntactic errors as well; the ONERR grabs those too.

The DEF FN feature of Applesoft is rarely used, although it is one of its most powerful programming tools. It is used to express simple functions without having to restate them over and over again. Make single-argument functions such as: converting radians to degrees, getting address bytes for POKEing, scaling graphics displays, encoding characters and making special math functions like MOD.

When writing *FOR loops,* remember the STEP option. A STEP-1 makes the loop count backwards. Often you can use a regular variable instead of creating a new one for the loop index. For instance, if you want a table of temperature conversion, you might write:

```
FOR F = 28 TO 36 STEP .1
PRINT F,(F – 32)*5/9
NEXT
```

Keep the keywords TRACE, WAIT, ONERR, DEF FN, and STEP in mind as you program with Applesoft. They can make programming much easier.

Instead of Applesoft, you may program with Integer BASIC. See Table 1-4. It has a much shorter instruction set, but it *parses* your commands faster. When an Integer line is entered, it parses the statement more completely than an Applesoft statement would be parsed. This results in a *predigested* line stored in the program so that it executes faster. Applesoft must parse most of the statement at execution time. So, for fast execution, especially with paddle games, Integer BASIC is better.

For reference, here are the Integer BASIC commands:

AUTO – gives you an automatic line numbering mode for entering programs. AUTO 100, for instance, will start at 100 and give you new line numbers incrementing by 10. Other increments are

Table 1-4. Integer Command Set

Program Flow	Input/Output
CALL. . . END FOR. . . = . . .TO. . .STEP. . . GOTO. . . GOSUB. . . IF. . .GOTO. . . IF. . .THEN. . .ELSE. . . NEXT. . . POP REM. . . RETURN	IN#. . . INPUT. . . LOAD PDL (PEEK (POKE. . . PR#. . . PRINT. . . SAVE

Screens (text and LORES)		Variables Control

TAB. . .   COLOR = . . .
TEXT   GR
VTAB. . . HLIN. . .
          PLOT. . .
          SCRN (
          VLIN. . .
		CLR
		DIM. . .

Math and String Functions		Edit and Debug
ABS (RND (ASC (SGN (LEN (		ctrlC, ctrlX, and reset AUTO. . . CON DEL. . . DSP. . . HIMEM:. . . LIST. . . LOMEM:. . . MAN NEW NODSP. . . NOTRACE RUN. . . TRACE

Assignment Symbols	
LET ( ) = + − AND * / ∧ # OR NOT	

possible: AUTO 300,4 will start at 300 and increment by 4; AUTO 500,25 will start at 500 and increment by 25. To exit AUTO mode, use *ctrl/X* followed by the MAN command.

CLR — clears all variables and undimensions all arrays.

CON — continues program execution after a STOP or *ctrl/C*. All variables are normally left intact.

DSP — turns on a debug display feature that displays a given variable each time the executing program references it. For instance, DSP COUNT will display the contents of COUNT whenever a statement containing COUNT is executed. You use CON or GOTO to run because the RUN command cancels the DSP feature. The DSP is an attribute of the variable itself, so you can DSP any number of variables at the same time as you want.

HIMEM: — sets the highest memory location available to any programs. It will *destroy* the current program.

LIST — works just like the Applesoft LIST.

LOAD — is a tape command. Two *beeps* and a ">" signals the successful LOAD of an Integer BASIC program.

LOMEM: — sets the lowest memory location available to any programs. It will *destroy* current variables, so it must be used before any variables are declared.

MAN — turns off the AUTO line numbering.

NEW — clears out any current program in memory.

NO DSP — turns off the display attribute of a variable. For instance, a DSP COUNT can be canceled by a NO DSP COUNT statement.

RUN — works like Applesoft. All variables are cleared, the dimensions are removed, DSPs cleared, and program execution is begun at the lowest line number. If a line number is given, like RUN 1000, execution begins there.

SAVE — is the tape command to save the current Integer BASIC program to cassette tape.

TRACE and **NOTRACE** — work like their Applesoft counterparts. They display line numbers of executing statements.

For further reference, here are the Integer BASIC statements that can be used in programs. Several statements can be included on a single line, separated by line numbers, just like Applesoft. These statements are the programmable commands, then, of Integer BASIC:

CALL — works like Applesoft's CALL except for the restriction of numbers from −32768 to 32767. The negative addresses are used

for values above 32767; for example, CALL − 936 is the Monitor call for HOME.

COLOR − will set the LORES color. Give a number from zero to fifteen.

DIM − dimensions a variable differently from Applesoft. For integers, give *one* number for the array size; sorry, no higher orders. For strings, give the maximum length for *one* string, from one to 255. String variables default to single byte characters if not DIMensioned.

DSP − can be used within statements as well. Each statement must DSP only one variable.

END − halts program execution.

FOR − works like it does in Applesoft. You can use STEP for increments other than + 1.

GOSUB − works with a line number or an expression to calculate a line number.

GOTO − has the same syntax as GOSUB.

GR − sets the LORES graphics display mode and blacks the screen. You get *scrolling* text at the bottom of the screen.

HLIN, VLIN, and **PLOT** − work in LORES graphics like Applesoft.

IF . . THEN . . − tests an expression. If true, it executes a statement. An ELSE may be used for an alternate statement. In Integer BASIC, any statements within the same line after the ELSE statement will *always* be executed, regardless of the IF. This is different from Applesoft BASIC where following statements on the same line are treated as part of the ELSE condition. Watch this one; it can be deadly.

INPUT − works the same as Applesoft.

IN# − sets the current input device to the slot number.

LET − is optional on assignment statements.

LIST − can be used in statements. Use it to capture Integer BASIC programs to TEXT files as described.

NEXT − must have the variable name of the FOR statement.

NODSP − turns off the DSP attribute of a variable.

NOTRACE − turns off the TRACE feature.

POKE and **PEEK,** − of course.

POP − acts like a dummy RETURN. It *pops* the GOSUB stack by one without actually doing a RETURN.

PRINT − must be typed; you can't use the trick of typing "?" like you can with Applesoft. Commas tabulate; semicolons suppress the car-

riage return at the end of the statement.

PR# — sets the current output device to the slot number.

REMs — are allowed.

RETURN — returns from GOSUB in subroutines.

TAB — is Integer's equivalent of HTAB in Applesoft.

TEXT — acts the same as Applesoft.

TRACE — displays executing line numbers.

VTAB — is the same as Applesoft.

MOD — is unique to Integer BASIC. This function gives you the remainder from a division. For example, 23 MOD 7 gives 2, and 36 MOD 9 gives zero. The quotient comes from the DIV; like 23 DIV 7 that gives 3 or 36 DIV 9 that gives 4. Other functions appear in the summary of Table 1-4.

More information on Integer BASIC is given in Chapter Five.

1.2.2 Strings

Using Applesoft, it's easy to make strings and join them together. Just by entering

$$A\$ = A\$ + B\$$$

you can join the contents of A$ and B$ with the result as A$. If you want to do this in Integer BASIC, it's a little trickier; enter

$$A\$(LEN(A\$)+1) = B\$$$

instead. In either case it can be done. This joining operation is called *concatenation*.

You concatenate strings all the time when programming. One reason is to make a natural-language display like

```
INPUT"HI, WHAT'S YOUR NAME? ";N$
PRINT"PLEASED TO MEET YOU, "+N$+"."
```

that concatenate a name in N$ with the screen message. Or, you may join strings to write to a disk file. This is done usually by a statement like

```
PRINT A$,B$,C$,D$
```

that concatenate four strings with three-comma characters to make a single record. Another statement to read these four strings from a single record is

INPUT A$,B$,C$,D$

where there are exactly four variables corresponding to four strings separated by commas in the record.

Each string in the list of a PRINT or INPUT statement is called a *field*. A record, therefore, consists of one or more fields separated by special characters called *delimiters*. The comma is a delimiter of fields in Apple records. The statement

PRINT A$,B$,C$,D$

and the statement

PRINT A$ + "," + B$ + "," + C$ + "," + D$

both result in the same record being output.

The corresponding input statement

INPUT A$,B$,C$,D$

expects one record of four fields separated by commas. If there are fewer fields, the missing ones will be taken from the next input record; that is why keyboard INPUTs reprompt for missing fields. If there are too many fields in the record, the last ones are ignored and a message to that effect is output — EXTRA IGNORED.

When using the DOS manual with records and fields, be careful. The manual is excellent in many respects, but unfortunately it often refers to records as fields. To set the record straight, so to speak, records are defined as all characters in those file substrings terminated by CR characters (negative ASCII code $8D). Within each record is one or more fields separated by delimiters, usually commas. All fields are character strings, and numeric variables are read from string fields by the INPUT routine that uses a string to number conversion subroutine. Just remember that a file consists of records, which in turn consists of fields.

Strings can have any length from zero (the *null* string) to 255 characters each. On screen they are best displayed with scrolling, on disk

they are best managed in sequential TEXT files. Such free-form strings are the simplest and easiest to program with and should be your first choice.

Instead of variable-length strings, you sometimes need fixed-length strings to do a job. A screen layout with a lot of information that can't be allowed to scroll itself off screen is one example. Random access for *query* and updating very active files on disk is another. While fancy screens and random access have nice features, make sure you really need them as fixed-length strings take more programming to get working properly than do variable-length strings.

You can go one of two ways to get a fixed screen layout. One way is to simply prompt for each entry at its final display position on the screen. You must set the window to the field area each time you prompt for the field; otherwise, the user can enter outside the area. Then you have to clear the window, input the user's entry, and re-display the field if justification is needed. The second way to go is to set up a prompt line, usually two or three scrolling lines at the bottom of the screen. This lets you prompt and give error messages by scrolling in the old way. When you get a good field, you display it on the screen in its proper position. The second method is easier, but uses more screen space. Use the second method unless your screen must be especially intense.

When you use fixed-length strings, you must maintain them by *truncating* any that are too long and *filling* any that are too short. For example, to make a string A$ fit a length, L, write

$$A\$ = LEFT\$(A\$ + BL\$, L)$$

where BL$ is a long string containing all blanks.

Fixed-length strings are often changed by replacing a substring. You may want to handle the fields-within-a-record logic yourself with this method, or just change part of a display before PRINTing it. You need two different statements to insert a substring. The general one is

$$A\$ = LEFT\$(A\$, P) + B\$ + MID\$(A\$, P + LEN(B\$) + 1)$$

where P is the position of the substring in A$ to be replaced by B$. However, if B$ must go at the beginning of A$, you need

$$A\$ = B\$ + MID\$(A\$, LEN(B\$) + 1)$$

instead. This is because the LEFT$ function can't use a position, P, of zero.

Extracting substrings is a bit easier than inserting them. You need only one statement:

$$B\$ = MID\$(A\$,P+1,LEN(B\$))$$

The position, P, is the same one; its range is

$$0, 1, 2, \ldots, LEN(A\$) - 1$$

Sometimes you want to position a substring into a variable-length string. Placing any length string into the left, center, or right of another string is called *justification*. Report titles, for instance, are center justified to look proper. Numbers are often right justified to *line up* the columns. Labels can be left justified. Here's how to do justification. To left justify:

$$A\$ = B\$ + MID\$(A\$,LEN(A\$) - LEN(B\$))$$

to right justify:

$$A\$ = LEFT\$(A\$,LEN(A\$) - LEN(B\$)) + B\$$$

to center justify:

$$A\$ = LEFT\$(A\$,(LEN(A\$) - LEN(B\$))/2) + B\$$$
$$+ RIGHT\$(A\$,(LEN(A\$) - LEN(B\$))/2$$

for any pair of strings where LEN(A$) > LEN(B$).

Extracting variable-length substrings requires a search. A BASIC loop will work slowly, so if you want to use it often, you might get an Assembler routine to do it for you instead. Here's the Applesoft version:

```
P = 256
FOR I = 1 TO LEN(A$) - LEN(B$)
IF MID$(A$,I,LEN(B$)) = B$ THEN P = I - 1
NEXT I
```

A large value, 256, returned in P signifies a miss. A smaller value is the position of B$ in A$.

1.2.3 Terminal

The normal INPUT and PRINT statements input lines from the built-in keyboard and output them to the video display. The GET inputs single characters. Using PR# and IN# will change these to the device of the slot you specify. When you want to work with the built-in keyboard and video displays, there are several tricks and shortcuts you can use.

For instance, the GET. If you don't want the cursor display and character echo that the GET command features, you can use your own Applesoft subroutine instead:

```
WAIT  – 16384,128
A$ = CHR$(PEEK( – 16384) – 128)
POKE  – 16368,0
RETURN
```

This subroutine waits for a keypress, fetches the character into A$, then clears the keyboard for the next keystroke.

Another GET routine can just look for a keystroke in response to a prompt. In this case you don't care to know which key was pressed; you just want to wait until a message has been read before continuing:

```
VTAB 23: HTAB 12: PRINT"PRESS-A-KEY";
WAIT  – 16384,128
POKE  – 16368,0
VTAB 23: HTAB 12: PRINT"
RETURN
```

And you can use another character GET routine to prompt with a special cursor

```
PRINT C$;:REM Your special cursor character
POKE 36,PEEK(36) – 1: REM Backup
WAIT  – 16384,128
A$ = CHR$(PEEK( – 16384) – 128)
POKE  – 16368,0
PRINT A$;: REM Echo character
RETURN
```

This one uses a PRINT to display your special cursor character in C$. A second PRINT character *echoes* the keyboard character at the cursor position.

A favorite trick with programmers is the *on-the-fly* GET. You use this whenever you don't want to wait for a keypress; instead you want the keypress to interrupt whatever you are doing. In Applesoft, you write your task with a loop that tests the keyboard. If you get the keypress you want, you can change your routine or RETURN, as you wish. The trick is in not stopping the routines while you wait for a keypress:

```
2000  GOSUB your task
2010  IF PEEK(– 16384) < 128 THEN 2000
2020  POKE – 16368,0
2040  RETURN
```

The task you perform at line 2000 is repeated until any key is pressed. At that time, the keyboard is cleared and the routine RETURNS.

Whenever you use the HOME or CALL-936 statement to clear the video display, only the display *window* is cleared. This window is the area of the screen that scrolls; it is the full screen if the TEXT statement is used. The HOME positions the cursor to the upper left corner of the window.

From the home position, the text cursor can be moved by the tab statements and by PRINT actions. You can easily lose track of the cursor in a program, especially if the program structure is weak. Often this doesn't matter much, as in the case of simple scrolling. In other cases, when you may have a busy form on the screen, it matters a lot. Then, you need methods of finding the cursor location and forcing the cursor to follow a specific screen layout.

To locate the cursor, you need three cursor parameters: the horizontal cursor, the vertical cursor, and the left window. The expressions to calculate the absolute row and column of the current cursor are

$$\text{row} = \text{PEEK}(37) + 1$$
$$\text{col} = \text{PEEK}(32) + \text{PEEK}(36) + 1$$

The row is found from the vertical cursor. The column is found from the left window plus the horizontal cursor. The top window parameter

is not needed because the vertical cursor counts from the very top of the screen. See Chapter Six for details on the window parameters.

You can always find the cursor with the above method. It is useful whenever you want to change the cursor in a relative way, or you want to test the cursor for a boundary condition. The other need you have of cursor control — forcing the cursor to follow a form — requires planning if an oversized, hard-to-maintain result is to be avoided.

The first thing you must do is draw layouts of your screens. You can get forms for this purpose with rows and columns numbered along the borders. Or you can use that old standby — quad paper. Keep each screen simple and for a single task. Use one of the standard kinds of screen whenever possible: a menu, a box form, an open form, or an information message. A menu has a list of choices and prompts for a single key response. A box form displays a label and shows the size for each field to be entered. An open form divides the screen into a display area and a scrolling prompt-and-answer area to dialog with the user. An information message screen just displays message text and waits for a keypress to give the reader time to read it.

With screen layouts in hand, you are ready to program them into BASIC.

For each screen list the row number, the column number, and the string you want to display. Program these lists using DATA statements. In the initialization subroutine of your program, write loops to READ them into DIMensioned variables. Write display routines by using loops with HTAB, VTAB, and PRINT statements. The actual routine to display a screen is then quite simple; something like

```
3000 FOR F = 0 TO NF
3010 HTAB FH(F): VTAB FV(F)
3020 PRINT FL$(F)
3030 NEXT F
```

where NF is the number of the last field, FH is a list of rows of the fields, FV is a list of columns of the fields, and FL is a list of the label strings of each field. If you DIMensioned another set of strings for the contents of the fields as FC$(NF) then

```
3100 HTAB FH(F) + LEN(FL$(F)) + 1
3110 VTAB FV(F)
3120 PRINT FC$(F)
```

would display the F-th field contents one space after its label on the screen.

You can use the same kind of programming to design and code reports to your printer. The best advantage of this approach is the ease of changing field information in the program. You just alter a DATA statement in most cases without concerning yourself with how the program actually works.

You can keep track of the cursor in graphics mode — HIRES or LORES — easier than in TEXT mode. The PLOT or HPLOT statements use screen coordinates. You can use a function in LORES called SCRN that returns the COLOR value at the cursor location you specify. If you have several colored objects on a LORES screen, you can tell which one, if any, is at the X-Y location you plan to use next by testing the location first with a SCRN comparison. So, with graphics you always write with an absolute cursor position, and in the case of LORES, you can always read the color value at any cursor position.

You can use HIRES graphics to draw complex, delineated shapes. Unfortunately, it is slower and has fewer colors than LORES. Also, HIRES is more difficult to program. Generally, if you want HIRES use Applesoft or Assembler; if you want LORES use Integer BASIC. The Assembler can give faster execution times than Applesoft, but Integer BASIC with LORES graphics is quite fast.

Here's how to use HIRES with Applesoft. Select HIRES graphics mode with the statements

HGR : HCOLOR = 3

where 3 is the so-called *white1* drawing color. Even though you will assign another color later set this one first.

When using Applesoft to draw objects, you can build them from simpler objects called *primitives*. Those primitives you will use most often are rectangles, circles, and polygons. They may be *filled* with a color or *unfilled* as an outline. Unfilled primitives are the easiest to program. Then use primitives of different sizes and shapes to draw the objects you want.

The rectangle is easy. You can draw an unfilled rectangle with the routine

```
HPLOT X,Y TO X+DX,Y TO X+DX,Y+DY TO
X,Y+DY TO X,Y
```

where X,Y are the coordinates of the starting corner and DX,DY are the dimensions — width and height — of the rectangle. A filled rectangle with the same parameters is drawn by

```
FOR IY = Y TO Y+DY
HPLOT X,IY TO X+DX,IY
NEXT
```

Filled rectangles are great for backgrounds and making large objects. Avoid filled primitives on small objects unless you try it out first.

Circles are another class of primitives; you use them to get curved lines. They can give softness to an object's shape. Each circle has three parameters — two center coordinates and a radius. It's a little trickier than the rectangle, but you can use Applesoft's trig functions to make it easy to write.

```
P2 = ATN(1)*8: REM 2pi radians
DT = ATN(1.0/R)
FOR TH = 0 TO P2 STEP DT
HPLOT R*COS(TH), R*SIN(TH)
NEXT
```

This draws a circle, unfilled. The filled circle must use a loop to draw lines from side to side. The routine given here starts at the top and works to the bottom:

```
HP = ATN(1)*2: REM half pi
DT = ATN(1.0/R)
FOR TH = -HP TO +HP STEP DT
HPLOT X-R*COS(TH), Y+R*SIN(TH) TO
    X+R*COS(TH), Y+R*SIN(TH)
NEXT
```

Be careful using circles. If you draw beyond the screen area, the figure will wrap around to the opposite edge of the screen. You must range test your parameters first to keep this from happening.

Polygons are the most powerful primitives. They permit you to draw any shape you wish by the *join the dots* method of children's coloring books. Deceptively simple, the TO option of the HPLOT routine lets you draw any polygon with an explicit statement. If you stuff all your polygons in vector tables, you can write a single routine to scan a table and draw its polygon. For example, if your polygon was in vectors X and Y with the end of the polygon marked by large coordinate values, the routine to draw it is

```
410 I = 1
420 HPLOT X(I − 1),Y(I − 1) TO X(I),Y(I)
430 I = I + 1
440 IF X(I) < 280AND Y(I) < 192 THEN 420
450 HPLOT X(I − 1),Y(I − 1) TO X(0),Y(0)
460 RETURN
```

The first point is X(0),Y(0) and the last HPLOT closes the polygon to the first point. This gives an outline of the polygon; the filled polygon is difficult to program and takes a lot of testing to get debugged. Execution of such a routine would take a long time in Applesoft.

For most drawing, the limited resolution of the Apple II can be exploited quite well with these few primitives. Use fills only on backgrounds and other large areas; use outlines on the detailed objects. Keep drawings simple like posters and cartoons.

1.2.4 Program Design

The first thing a program must do is initialize. This includes such tasks as setting program memory usage, loading Assembler routines, defining constants, setting initial values, setting up screens, DIMensioning vectors, READing lists, and DEFining functions. All of these things are only done once in the execution of the entire program, so the best place for initialization is at the beginning of its mainline.

When writing the initialization of an Applesoft program, keep the following statements in the sequence shown:

```
NOTRACE
POKE 49236,0 :REM set Screen One switch
TEXT: SPEED= 255
NORMAL: HOME
IN#0: PR#0: CALL 1002: REM reset DOS hooks
MAXFILES 3
HIMEM: 38400:REM $9600
CLEAR: RESTORE
```

By using all this stuff, you can make sure your program will rerun after an error stop or after another program leaves you a dirty system. Then, with the system tidied up, you can proceed with initialization and load Assembler routines, DIMension vectors, READ and assign initial values and constants, and call your main procedures with GOSUBs.

When you declare constants, do so all in the same chunk of line numbers in the initialization, before variables are initialized. This lets you use constants in your variables setup. Some constants usually needed are

```
D$ = CHR$(4) : REM ctrl/D for DOS
BL$ = ''
             '': REM make a blank string
PI = 4*ATN(1) : REM pi = 3.14159...
HP = 2*ATN(1) : REM half of pi
P2 = 8*ATN(1) : REM twice pi
Z$ = "0000000000000" :REM zeros for formatting
```

Allow additional line numbers for adding more constants; you'll need them as you write your routines.

Don't forget your functions. Most programs can be written easier with a few extra functions available. Here are a few you may want to choose from:

```
DEF FN LO(X) = X - 256*INT(X/256)
         low-order byte of an address, for POKEing
DEF FN HI(X) = INT(X/256)
         high-order byte of an address, for POKEing
DEF FN AD(X) = PEEK(X) + 256*PEEK(X + 1)
         fetches an address pointer from memory
```

DEF FN DE(X) = 180*X/PI converts radians to degrees
DEF FN RA(X) = PI*X/180 converts degrees to radians

Do you need others for scaling, rounding, translating, or other repetitive use? Create your own.

While you are writing the mainline of an Applesoft BASIC program, code the error handler routine. Without using an ONERR GOTO . . . statement, the program will stop executing and you get an error message displayed.

Error handling may be simple or complex, depending on your needs. Here is a simple error handler that uses various options:

```
30900 ER = PEEK(222): EL = FN AD(218)
30910 POKE 216,0
30920 PR#0: IN#0: CALL 1002
30930 TEXT: POKE 49236,0: PRINT CHR$(7)
30940 PRINT"ERR";ER;" AT LINE ";EL;
30950 PRINT" RESUME/QUIT? ";
30960 GET A$
30970 IF A$ < > "R" AND A$ < > "Q" THEN 30960
30980 IF A$ = "Q" THEN 30990
30982 ONERR GOTO 30900: RESUME
30990 PRINT D$"CLOSE"
30992 CALL 62248: GOTO 32767
```

Include the ones you need in your error handler. ER is the *error code*. See Table 1-5. EL is the line number at which the *error occurred*. At memory location, the ONERR flag is cleared by the POKE to disable further error traps. Then, DOS is reset without any other device. The screen is returned to normal at line 30930 and the beep is sounded. The error prompt occurs on the bottom line of the screen, thanks to the TEXT statement, and a "Q" for *quit* or an "R" for *resume* is accepted from the user. To RESUME, the ONERR GOTO is restated. To quit, an attempt to CLOSE all files is made. If successful, a polite END is made with a GOTO 32767. The program END statement is there. The CALL 62248 should be made to clean up the outstanding error whenever a RESUME won't be stated.

After a program has been written and debugged, the most common error is a DOS *drive error* when someone forgets to close the drive door. The RESUME should take care of that.

Table 1-5. The ONERR GOTO Codes

Code	Source	Message
0	Asoft	NEXT without FOR
1	DOS	LANGUAGE NOT AVAILABLE
2	DOS	RANGE ERROR
3	DOS	RANGE ERROR
4	DOS	WRITE PROTECTED
5	DOS	END OF DATA ·
6	DOS	FILE NOT FOUND
7	DOS	VOLUME MISMATCH
8	DOS	I/O ERROR or <u>drive error</u>
9	DOS	DISK FULL
10	DOS	FILE LOCKED
11	DOS	SYNTAX ERROR
12	DOS	NO BUFFERS AVAILABLE
13	DOS	FILE TYPE MISMATCH
14	DOS	PROGRAM TOO LARGE
15	DOS	NOT DIRECT COMMAND
16	Asoft	SYNTAX ERROR
22	Asoft	RETURN without GOSUB
42	Asoft	OUT OF DATA
53	Asoft	ILLEGAL QUANTITY
69	Asoft	OVERFLOW
77	Asoft	OUT OF MEMORY
90	Asoft	UNDEFINED STATEMENT
107	Asoft	BAD SUBSCRIPT
120	Asoft	REDIMENSIONED ARRAY
133	Asoft	DIVISION BY ZERO
163	Asoft	TYPE MISMATCH
176	Asoft	STRING TOO LONG
191	Asoft	FORMULA TOO COMPLEX
224	Asoft	UNDEFINED FUNCTION
254	Asoft	Bad INPUT response
255	Asoft	ctrl/C input

The trickiest problem in programming in BASIC is managing line numbers. This is due in part to the use of line numbers as statement labels for GOSUBs and GOTOs. The most annoying problem is running out of line numbers. Then, if you want to put your most often

used routines at the beginning of the program to speed up GOSUB references, you may not have enough line numbers available.

Many programs are written by starting with a low line number like 1000 or even 100 and adding lines by increments of 10 or so. An increment of ten is usually enough *insurance* against running out of lines added when debugging a routine, but it won't protect you from trouble when you have to write an entire routine at a low line number.

There are RENUMBER utilities available. Smart ones resolve GOTO and GOSUB references and are quite good. In a large program already written, it could be the best solution. In general, though, they have problems. You have to get used to a new set of routine addresses when working on the renumbered program. And, you can create conflicts if you want to keep a library of subroutines in TEXT files to EXEC into new programs. For any new program design, you should assign line numbers for the exclusive use of the various parts of the program before doing any coding. By *blocking* line numbers this way, conflicts and the need to renumber can be eliminated entirely.

Here's how you can block line numbers like the professional BASIC programmer does. You break up all the line numbers from zero to 32767 into blocks of numbers. Each block is then assigned to a different level of the program. Within each level, you block the numbers further for all routines at that level. The rule to follow is: the higher the program level a routine has, the higher the line number you assign.

Look at some parts of a program and see how they get their line numbers.

The mainline is not referenced often and it has the highest level. For these reasons, the best place for the mainline is at the *end* of the program. You can use line numbers from 30000 to 32767 for the mainline. These are the highest line numbers for the highest level of the program. To reach the mainline when a program is RUN, let line number 10 be the lowest line number of the program and make it:

```
10 GOTO 30000
```

This leaves lines zero to nine clear for CAPTURE routines and special tests during debugging. The mainline proper begins at 30000 with the initialization statements.

The block from 31000 to 31999 within the main block is ideal for DATA statements. They are accessed usually only once, during ini-

tialization, so they can have high numbers. The remaining block from 32000 to 32767 can be used for your program exit routine. By convention, many programmers place the END statement alone at line 32767.

The mainline routine after initialization should be brief. A few GOSUBs, perhaps with an IF statement to detect the end of the program, are common. So is the computed GOSUB. All the work should be done by subroutines; only very simple, high level logic should be done by the mainline. The mainline calls the major functions of the program.

Each of the major functions is blocked. Since they are at the next lowest level, you can use the line numbers from 20000 to 29999 for them. Each one can have its own block there: put one at 20000 to 20999; put another at 21000 to 21999; put yet another at 22000 to 22999; and so on, up to ten major functions. Examples of major functions include menus, disk file access, a sort, a drawing routine, etc. Each one works by performing a major task by calling on minor tasks and utility routines in turn.

Minor tasks are tasks that more than one major task may perform. Examples include accessing the disk with READs and WRITEs using special formats, setting up screens and table-driven displays, and report printing logic.

Utilities on the other hand are short, fast, and may be called by any routine in the program. Your special GET routine, graphics scaling, cursor control, formatting, lookups and searches all qualify as utilities. By having the lowest line numbers, 100 to 999, they are executed as quickly as possible.

Table 1-6 is an example of a line number blocking scheme. The assignment of blocks over the middle levels of a program varies with that program's call structure. Remember the principle of high level, high line numbers as you design your BASIC program. See Example 7-1 for an Integer BASIC program that has block structuring.

1.3 FILE HANDLING

1.3.1 Sequential Files

The simplest method of file creation is the sequential file method. Sequential files have records of varying lengths, just like character strings from the keyboard. Unless you have to make random access to

Table 1-6. Blocking Line Numbers

Line Numbers	Description
0 to 9	Unused: Reserved for capture routine
10	GOTO 30000
100 to 999	Fast, common utility routines
1000 to 9999	Specific service routines
10000 to 29999	Main routines of primary menu selection
30000 to 30999	Main line: initialization and menu. An error handler included.
31000 to 31999	The DATA statements
32000 to 32767	Termination routines. Line 32767 should contain only an END statement.

any record in the file, the sequential file access method is the one to use.

You create sequential files according to how you intend to update them. The simple method given in the DOS manual assumes you know exactly how many records are in any file at any given time. If this is the case, then the file can be read sequentially using the number of records as the last record number. After the last record has been read, an error (code = 5) occurs when the next READ attempts to read beyond the end of the file.

If you don't know how many records the file contains when you write the file reading routine, you need a better method. The most obvious is perhaps one that traps the end-of-file error by testing the error code in the ONERR routine:

```
30912  IF ER = 5 THEN 30982
```

This forces a RESUME at line 30982 if there is an END OF FILE error:

```
30982 EF = 1 : ONERR GOTO 30900 : RESUME
```

The line that contains the INPUT where the end of file was found must detect EF, the end-of-file flag:

```
1350  EF = 0 : make FALSE
1360  IF NOT EF THEN INPUT F1,F2,F3
```

where 1360 can be any line that INPUTs from the file.

Perhaps a more secure method is to use one record of the file itself to keep count of the total number of records. Such a *header record* can be created at the same time as the file:

```
22000 PRINT D$"OPEN"F$
22010 PRINT D$"CLOSE"F$
22020 PRINT D$"DELETE"F$
22030 PRINT D$"OPEN"F$
22040 PRINT D$"WRITE"F$
22050 PRINT "    0"
22060 PRINT D$"CLOSE"F$
22070 RETURN
```

This creates a new file, deleting any previous version, by using F$ as the file name. The file contains exactly one record having the fixed length of five characters — four spaces and a zero. There are no data records yet; hence the zero.

To add records to the file, you must get the header record to update, add records starting at the end of the file, and finally change the header record to the new number of records. Here's how to open the file and set its position to the end of file:

```
22100 PRINT D$"OPEN"F$
22110 PRINT D$"READ"F$
22120 INPUT NR : REM number of records
22130 PRINT D$"CLOSE"F$
22140 PRINT D$"APPEND"F$
22150 RETURN
```

Upon RETURN, the record counter is in NR and the file position is at the end of the file, ready to append further records.

After records have been added, you need a special close routine that will update the header record with the record counter:

```
29000 PRINT D$"CLOSE"F$
29010 PRINT D$"OPEN"F$
29020 PRINT D$"WRITE"F$
29030 PRINT RIGHT$("    "+STR$(NR),5)
29040 PRINT D$"CLOSE"F$
29050 RETURN
```

All you have to do to make sure that the APPEND method works properly, then, is to add one to NR each time you output a new record:

```
23000 PRINT D$"WRITE"F$
23010 PRINT R$ : REM output record
23020 PRINT D$ : REM cancels WRITE command
23030 NR = NR + 1 : bump count
23040 RETURN
```

To read a sequential file with a header like this, you just make a simple OPEN statement followed by a READ:

```
21000 PRINT D$"OPEN"F$
21010 PRINT D$"READ"F$
21030 INPUT NR
21040 RN = 0
21050 RETURN
```

Set a record counter, RN, to keep track of which record is current. The open routine here does that; the current record is zero. By comparing RN and NR, you can detect the end of file before an error (code 5) occurs. Just increment RN at each record read.

Besides the APPEND method, you can choose another called the sentinel method. It uses simpler routines and is intended for files that are made with all their records entered at one run. You cannot add more records later on to the same file. However, sentinel files are compatible with sorts and merges while header files are not.

The idea behind sentinel files is to make your own endfile marker. Whenever a new file of records is created, you write an extra record just before CLOSEing the file. This record must be unique to all files so your programs can detect it when READing and know that the file is ended.

Declare a special character, called *high value,* to use as a sentinel: HI$ = CHR$(127).

Create your new file and put data records into it at the same time. Then you close the new file with an extra record made with the high value:

```
29000 PRINT D$"WRITE"F$
29010 PRINT HI$,HI$,HI$,HI$,HI$
29020 PRINT D$"CLOSE"F$
29030 RETURN
```

This is for a five-field file; you use as many high values as fields in your file.

To summarize, there are four ways to make sequential files. First, always use the exact same number of records so that your READ routine knows how many records are there, by implication. Second, trap the endfile with the ONERR. Third, keep count of the number of records in a fixed-length header record as you APPEND new records. And fourth, use a high value sentinel on fixed files to be merged and sorted.

Sorting puts records in order. With a printout of unsorted records, you have to search through the listing one record at a time to find the one you want. But when they have been sorted, you can quickly search out the one you want.

To sort a file, you decide first which field you will use to search. For example, a mailing list of friends and relatives could have either the last name or the first name chosen. Then, when printing labels, you may want to re-sort the mailing list by zip code. Whichever field is used to sort and search, it is called the *key* field.

You can choose any field of a file as the key field. Once chosen, however, it stays as the key field for all searches until sorted again. What you do is first sort the file into sequence according to the sort key field. Then use the sorted file as input to a report program that prints all records on the file. The report will be in sequence so that you can look up any record you want.

One use you can make of a sorted report is to make changes and additions to the sequential file. To change a record in a sequential file is difficult because the length of the replacement record may be greater than the old record. In that case, there won't be enough room for the replacement file. So, to replace records in sequential files, you have to mark them for *deletion* and treat the replacement record as an *addition*. The procedure is to make a second file containing all the additions, sort this new file on the same key field as the original, then merge the two files to create a new file, preferably on another disk. This new file won't have deleted records because the merge routine won't copy them as it builds the new file.

A sort routine and a merge are needed to completely maintain sequential files.

Sorts are usually written in Assembler because BASIC is much slower. Several sort routines have been published in magazines. Many of the routines can be bought at computer stores. The most successful ones are the Shell-Metzner, the Quicksort, and the Bubble Sort. Using one of these algorithms, a sort need only sort strings, because all fields are in string format. A variation called the Tag Sort will give you the sequence of the original keys in a separate numeric array. This lets you read in all the keys without storing the entire record, sort the keys, then use the tag numbers to give you the record numbers of the unsorted file in the sorted sequence. After sorting the keys, you read the file in blocks of as many records as will fit easily into memory. Then write the block out in sorted sequence, according to the tags. The resulting files of sorted blocks are then merged to make the sorted file.*

Remember that you must read the entire file and sort it all at once; then you can write out the sorted version. When the file is too big to stuff into memory all at once, you can use the Tag Sort method just described or break the file into smaller ones for sorting and merging. If possible, design your system so that any sequential files are kept small until sorted.

There are two ways to add records to a sorted file. The simplest way is to APPEND the new records, then re-sort the entire file. If the file is small and records aren't added often, then this is the way to go.

On the other hand, adding even only a few records to a file on a regular basis will make it grow quickly. Very soon, re-sorting will take considerable time, making the job of adding new records tedious. As an alternative, you can create a new file just for the additional records. Then sort the additions file and merge the old master file with the additions to make a new master file. The old master file is not re-placed, so you have a backup provided by the procedure.

The best way to organize your file for merging is to add new records to an additions file *on the same disk*. Then, merge the two files to a new file on a second disk. This way you have all the records of the file on one disk before merging, so you know that the new file will fit on the new disk. For instance, you could update a file on the Drive One disk by creating an additions file, then complete the update by mount-

*Irwin, "Tsort and Amperjump," *Nibble* magazine, V.2, N.6, 1981.

ing a scratch disk in Drive Two and INITing the new disk. Sorting the additions, then merging them with the master file to make a new master on Drive Two, completes the update session.

After merging, the old disk in Drive One is your backup; the new disk in Drive Two is now your master file disk.

Here's how to do a merge. First, you must define a high value constant: HI$ = CHR$(127). Remember, in the sentinel method, this value marks the end of file. The merge routine itself goes like this:

```
21100  GOSUB 11000 : REM read old master
21110  GOSUB 12000 : REM read addition
21120  IF KM$ > =KA$ THEN 21160
21130  GOSUB 13000 : REM write old master
21140  GOSUB 11000 : REM read old master
21150  GOTO 21120
21160  IF KM$ = KA$ THEN 21200
21170  GOSUB 14000 : REM write addition
21180  GOSUB 12000 : REM read addition
21190  GOTO 21120
21200  IF KM$ < > HI$ THEN 21130
21210  GOSUB 15000 : REM write HI$ at endfile
21220  RETURN
```

This is just the merge logic (see Fig. 1-4). The keys must be read as: KM$ for the master key, KA$ for the additions key. For instance, if your records had four fields and the first field was your sort key, then the routine to read the old master would look like:

```
11000  PRINT D$"READ"FM$ : REM FM$ is master file name
11100  INPUT KM$,M2$,M3$,M4$
11200  PRINT D$ : REM ctrl/D kills READ
11300  RETURN
```

Similarly, the routine to write the additions record would be:

```
14000  PRINT D$"WRITE"NF$ : REM NF$ is new master name
14100  PRINT KA$,A2$,A3$,A4$
14200  PRINT D$ : REM ctrl/D kills WRITE
14300  RETURN
```

A AND B ARE MERGED INTO C. WHERE KEYS ARE EQUAL
B RECORDS FOLLOW A RECORDS IN FILE C.
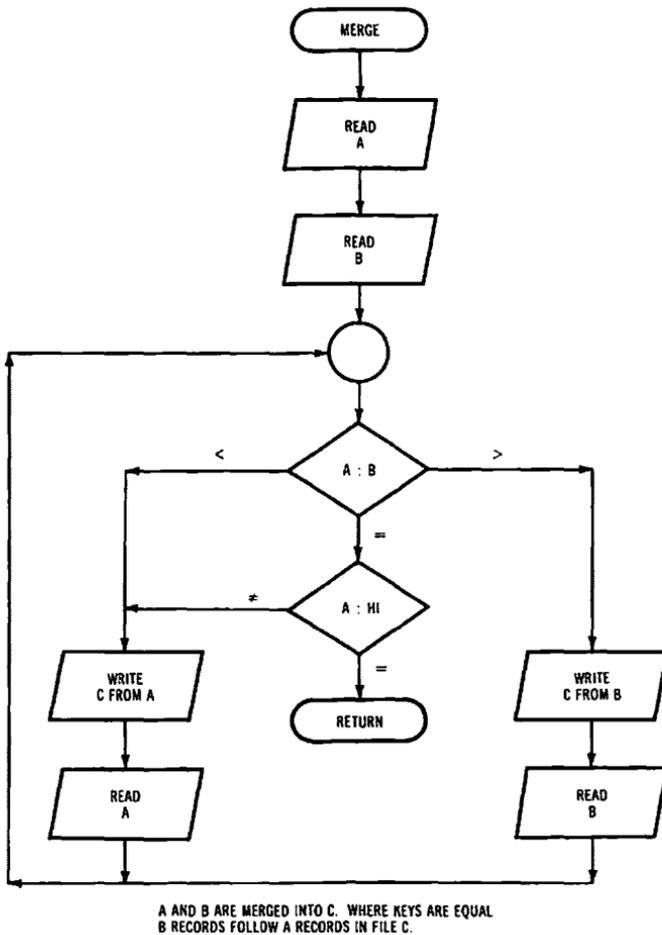
Fig. 1-4. The classic merge.

You can write other routines yourself. When coding the routine to write an old master record — at line 12000 — you can use an IF statement to test for your delete flag. This way, you can ignore the record to be deleted.

To start, write a simple merge to handle a simple file. You can rewrite it later, expand its features, and adapt it to other files.

To delete a record from a sequential file, the merge routine needs a list of records to be deleted. The list should be in sort sequence and

containing the *identifier* field of the records. The identifier is a field that is always different for each record. By keeping a subscript to the list, the merge routine that writes master records can test for deletion by comparing the identifier with the delete list. If a record is to be deleted, increment the delete list subscript instead of writing the record.

Here's how to start a sequential file system using the sentinel method. First, write a short program to create a new file on a blank disk. INIT three disks with successive volume numbers: 1, 2, and 3, say. Use your create program to put a null file — one with just the endfile record — on each disk. Label all three disks with the same file name. Make three labels for their *jackets* with their file name and generation: SON, FATHER, and GRANDFATHER. The younger the generation, the larger the volume number. So, place Volume 1 in the GRANDFATHER jacket, Volume 2 in the FATHER jacket, and Volume 3 in the SON jacket.

With a three-disk system like this, you can update, sort, and merge without danger of losing your file. Two disk drives are needed; each file has a full disk of file capacity. The disks are maintained such that anytime the file gets clobbered, it can be re-generated by a previous generation disk.

The procedure for updating a sequential file is diagrammed in Fig. 1-5. Here are the steps:

1. Mount SON in Drive One. Use UPDATE program to enter new records, creating an ADDITIONS file.
2. Optionally, you can run a report to check the ADDITIONS file for accuracy.
3. Sort the ADDITIONS file in Drive One.
4. Mount the GRANDFATHER disk in Drive Two and run the MERGE. This generates a new file by using the master and additions files in Drive One.
5. Remove the old FATHER disk from his jacket and put him in the GRANDFATHER jacket.
6. Remove the old SON disk from Drive One and put him in the FATHER jacket.
7. Remove the new file disk from Drive Two and put it in the SON jacket.
8. If you want an updated listing of the file, run a report using the new SON. This completes the file update.
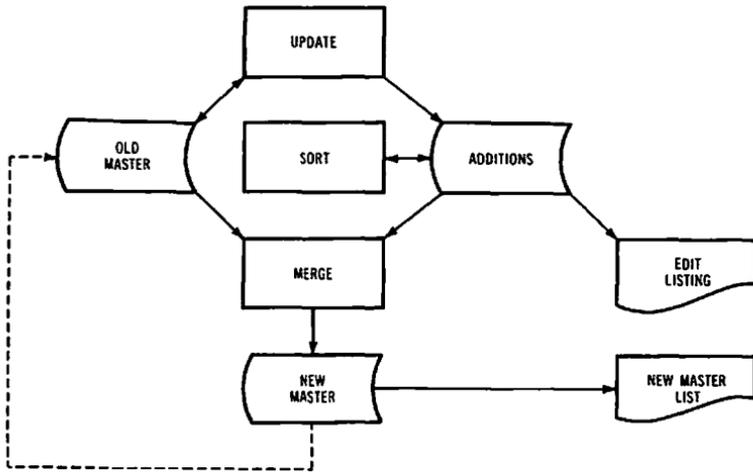
Fig. 1-5. Updating a sequential file.

An example of a Sequential File Manager Program appears in Fig. 1-6. Five routines will be needed altogether.

1.3.2 Random-Access Files

If you need to access a file and change it often, then it needs to be in a *random-access* organization. Such a file is different from a sequential file only by the lengths of its records. In random-access files, record lengths are all equal, while in sequential files, they can have various lengths. Because of the fixed length, DOS can quite easily calculate the position of any record in the file in terms of its actual disk location.

Random-access files are much easier to query and update because the sort, merge, and report functions aren't needed. A record can be replaced simply with a new record of exactly the same size. The trade off is in disk capacity, because random records must be filled out to the length of the longest record anticipated in the file. This filling could give you only half the file capacity of the equivalent sequential file. But, for files that must be queried and updated often, random-access files are your best choice.

One reason random-access files are easier to use is that you can use the record number directly. Each record in the file has its own number, counting from the beginning of the file, that you can specify with
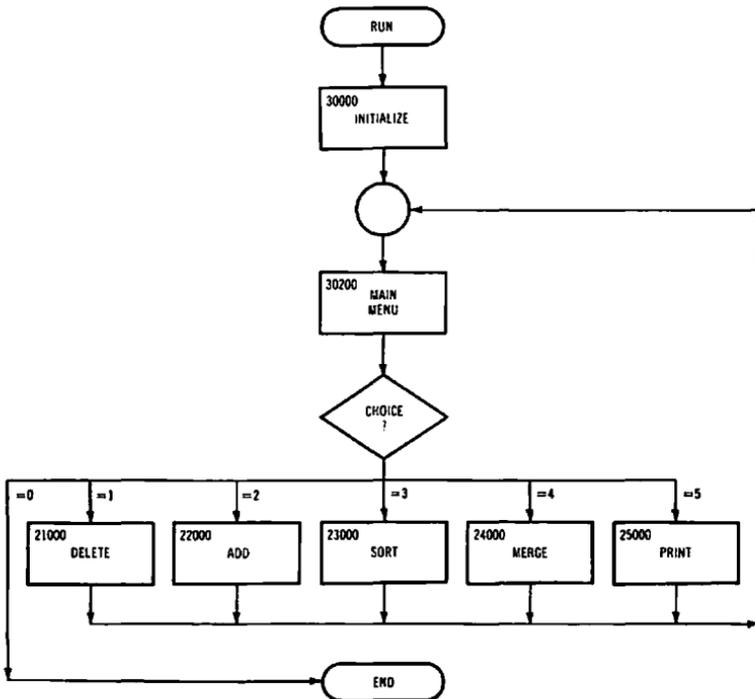
Fig. 1-6. Sequential file manager program.

the R option of the READ and WRITE commands. If you have a sort routine, you can also have *keyed access* by creating an index file to look up the record number for a given key value. To do this, you should have a Tag Sort routine that sorts the keys and gives you a table of their original position numbers. Regardless of the way you choose to go, random-access files will be much easier to manage when you have changes going on all the time.

The secret to making a good random-access file is in having a header. If the header is short, it may go at the beginning of the file as the first record. It is easier to handle if it is in a separate, small sequential file on the same disk.

You can put into the header all the information your UPDATE program will need to access the data. This includes file information like the number of records in the file, the number of fields in each record,

and the length of each record. Then you need information for each field you have defined for the file; at least its position, length, and name. With this information, your OPEN routine can get the header first and load up the file parameter variables your program uses to access the data. This scheme lets you change your file definition just by making a few changes to the header; your access program need never know the difference.

Here's how you might get the header from a separate, sequential file. The header file has the same name as the data file except for a ".HDR" extension:

```
21000 PRINT D$"OPEN"F$".HDR"
21010 PRINT D$"READ"F$".HDR"
21020 INPUT NR,NF,LR : REM  number of records,
                            number of fields, and
                            length of records.
21030 FOR I = 1 TO NF
21040    INPUT FN$(I),FP(I),FL(I)
21050    NEXT : REM field names, positions, and lengths
21060 PRINT D$"CLOSE"F$".HDR"
21070 PRINT D$"OPEN "F$","L"LR
```

With the file open and the header parameters read into their variables, you can update by READing and WRITEing any record you wish.

The length of each field is kept in the header, because of a different method of accessing fields when using random-access. The entire record is now handled as a single string, preferably using an *input anything* routine (see Chapter Six for details). Each field is inserted into the record string as a substring. The unused characters in the record must be made blank; otherwise, they can end up as nulls on disk. And nulls mark the end-of-file for DOS, so a READ will give you an error (code 5). Using substring logic for the fields releases one byte per field for data; commas are no longer needed. And, you can use the input anything routine of Chapter Six to allow commas within fields. So, you can READ and WRITE one string for each record, then use string functions to handle fields using the header information.

Here's how to get fields from a record string. If your fields are to be substringed to vector FD$ from record variable R$:

```
200 FOR I = 1 TO NF
210    FD$(I) = MID$(R$,FP(I),FL(I))
220    NEXT
230 FD$(0) = LEFT$(R$,1)
240 RETURN
```

There is an extra one-byte field at the left because of the inability of MID$ to get it. You might use it as the record status flag.

Here's how to build a record from the field substrings:

```
300 R$ = F$(0)
310 FOR I = 1 TO NF
320    R$ = R$ + LEFT$(FD$(I) + BL$,FL(I))
330    NEXT
340 RETURN
```

You can use the build routine after changing the FD$ strings and before WRITEing the R$ back to disk.

Design your own random-access file. Make a list on paper of all the fields and the length of each. Use a length of one for the zeroth field. Add up the lengths to get the record length; don't forget to add one for the CR character. This total length is your LR. Next, list the name, position, and length of each field. Then, write a routine to create the header file that the OPEN routine will read. By writing a short routine to call your OPEN routine and then CLOSE the data file, you can test your CREATE program.

With the header creation and OPEN routines working correctly, add the data file creation routine to the CREATE program. This should write blank records into the entire data file.

Next, write the record addition routine. Each time a new record is written, the number of records is used as the record number. Then the number of records is incremented by one. Warn the user if this reaches the total capacity of the file — the number of blank records provided at CREATE time. Don't accept any more additions in that case. When a record has been added, you must re-write the header file to update the number of records.

Finally, you can write a *query/change* routine. The header isn't changed by a record change. When the query works, use it to test the additions routine. Getting the changes and making them to disk comes
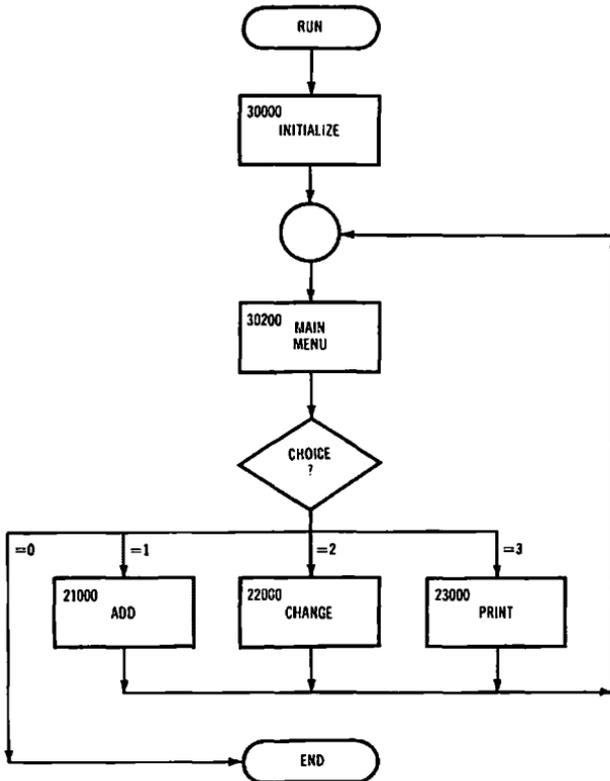
Fig. 1-7. Random-access file manager program.

last; if all other routines have been tested correctly, this won't be diffi-
cult at all. See the mainline in Fig. 1-7.

If you need more details on the DOS file access commands, see
Chapter Seven for a description and summary.

CHAPTER TWO

Atlas of the Apple II

2.1 MEMORY MAPS

The memory maps in this chapter show how the Apple II memories appear to both the display *generator* circuits and the *processor*. Several memory configurations are possible, depending on the model of the Apple II and its hardware and software options.

There may be a few Apple IIs around with less than 48K of RAM; these are not discussed here. These old 32K, 16K, and perhaps 4K Apples won't support much of the disk-based software available today, so most have been upgraded to 48K by adding the appropriate type 4116 RAM chips. If your Apple has a Revision 2 or earlier motherboard, the proper RAM jumper blocks must be installed. You can get them through an Apple dealer; they have 16K written on them.

When programming in Assembler, be careful with your Page Zero usage. With Applesoft and DOS both using Page Zero, space is at a premium. If you need a large chunk, swap it with a block of RAM before and after your routine. This will preserve the current system values:

```
ZSWAP    LDX #SIZE
ZSWAP1   LDA ZERO-1,X
         PHA
         LDA SAVE-1,X
         STA ZERO-1,X
```

```
            PLA
            STA SAVE-1,X
            DEX
            BNE ZSWAP1
            RTS
```

You must declare the save area in your program RAM

```
SAVE                    DS          SIZE
```

and EQUate SIZE as the length of Page Zero you are using and ZERO to its first location.

If you only need a few Page Zero locations, they can be borrowed. Find some locations that won't interfere with Applesoft or DOS by borrowing. Locations $06.09 seem to be unused by everyone. The $50.55 locations are used only for integer calculations and are usually *safe*.

Refer to the following maps to find the configuration that you have and any that you intend to use. Then use the gazetteer in the following section to see specific usage of any one block. Fig. 2-1 shows the methods for accessing the memory for all Apple models.

2.1.1 Apple II Memory Access Methods (for all models)

Each microsecond or so, the processor and the video display take turns accessing the memory. This gives the Apple a faster speed and gives the processor and video displays quite different memory maps.
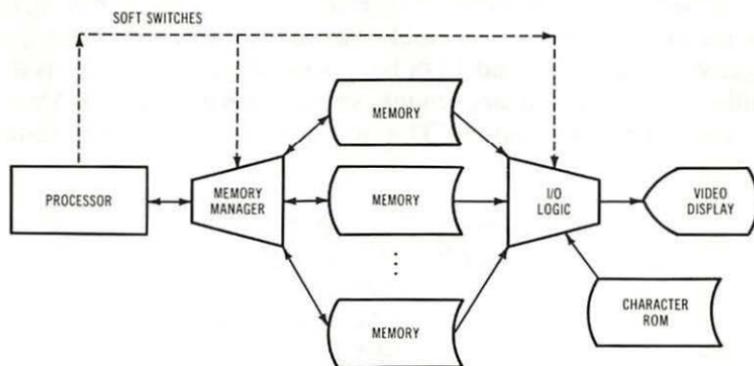
Fig. 2-1. Memory access methods for all Apple models.

To let the Apple handle more memory, a bank switcher called a *Memory Manager* controls the processor's memory access so that there can be one of several possible memory maps for programming. And similarly, the I/O logic that feeds the video display with screen data can select from among several chunks of memory. All these selections by the Memory Manager and the I/O logic depends upon the *soft switches* set by you using the processor.

2.1.2 The BASIC Memory Map

Fig. 2-2 shows the BASIC memory map. Of the many possible memory maps, this is the one most often used. This is what you get upon power up and the bootstrap of a DOS 3.3 program disk. The FIRMWARE area consists of the system Monitor and a BASIC — Applesoft or Integer. The INPUT/OUTPUT has addresses for hardware and peripheral firmware. DOS resides in the highest user RAM


```
FFFF ┌──────────────┐   ┌───────────┐
     │              │   │     ▲     │
     │   FIRMWARE   │   │ 12K      │
     │              │   │ ROM      │
     │              │   │     ▼     │
D000 ├──────────────┤   ├───────────┤
     │              │   │     4K    │
     │ INPUT/OUTPUT │   │ HARDWARE  │
C000 ├──────────────┤   ├───────────┤
     │     DISK     │   │     ▲     │
     │  OPERATING   │   │           │
     │    SYSTEM    │   │           │
9600 ├──────────────┤   │           │
     │              │   │           │
     │              │   │   48K     │
     │              │   │   RAM     │
     │     USER     │   │           │
     │   PROGRAM    │   │           │
     │    SYSTEM    │   │           │
     │              │   │           │
     │              │   │           │
0800 ├──────────────┤   │           │
0000 │  SYSTEM USE  │   │     ▼     │
     └──────────────┘   └───────────┘
```
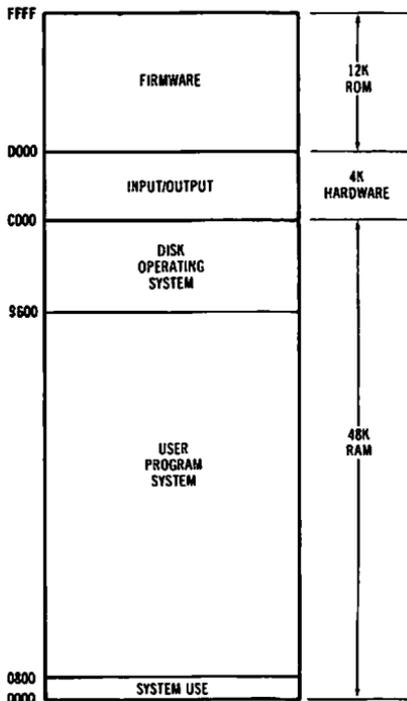
Fig. 2-2. The BASIC memory map.

addresses. The system reserves the lowest addresses for its own use in running the processor and supporting elementary features like screen displaying.

See the following maps (Figs. 2-3 through 2-10) for a breakdown of each area.

2.1.3 BASIC System Use

The 6502 processor requires the use of the lowest two pages of memory — Page Zero and Page One — in special ways. Page Zero is accessed by many instructions as address register locations and Page One is maintained as a special data structure called a *stack* that remembers addresses for machine language routines. By convention, Page Two is the *input buffer,* especially for keyboard inputs. Page Three is how the Monitor and DOS get along with each other; it contains system addresses that either can set. The SCREEN1 area contains all the character codes for each location on the 40 by 24 display.

```
$0800 ┌─────────────────────────┐
      │                         │
      │                         │
      │        SCREEN 1         │
      │         (TEXT)          │
      │                         │
      │                         │
$0400 ├─────────────────────────┤
      │        PAGE 3           │
      │   WORKSPACE & VECTORS   │
$0300 ├─────────────────────────┤
      │        PAGE 2           │
      │    KEYBOARD BUFFER      │
$0200 ├─────────────────────────┤
      │        PAGE 1           │
      │         STACK           │
$0100 ├─────────────────────────┤
      │       PAGE ZERO         │
      │       REGISTERS         │
$0000 └─────────────────────────┘
```
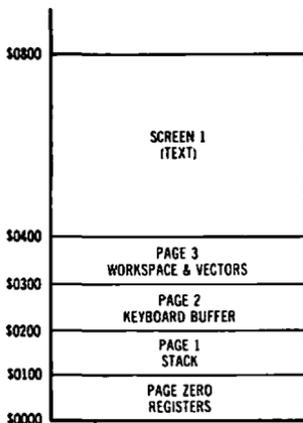
Fig. 2-3. BASIC system use.

2.1.4 The BASIC Disk Operating System

An EXECUTIVE connects with the rest of the system through the COMMAND INTERPRETER and gets commands. The COMMAND INTERPRETER understands DOS commands such as CATALOG and BRUN FID and has routines to execute them. The FILE MANAGER is normally called upon by the COMMAND INTERPRETER to open, close, read, write, etc. The RWTS has all the routines to access the disk and can be used by an Assembly programmer. The three DATA BUFFERS are used by the FILE MANAGER. Changing MAXFILES to any number but three will change the size of this area.

Fig. 2-4. The BASIC disk operating system.

```
C000  ┌─────────────────────┐
      │                     │
      │        RWTS         │
      │                     │
B600  ├─────────────────────┤
      │                     │
      │     FILE MANAGER     │
      │                     │
AAC1  ├─────────────────────┤
      │                     │
      │  COMMAND INTERPRETER │
      │                     │
9FCD  ├─────────────────────┤
      │      EXECUTIVE       │
9D00  ├─────────────────────┤
      │        THREE        │
      │     DATA BUFFERS     │
9600  └─────────────────────┘
```
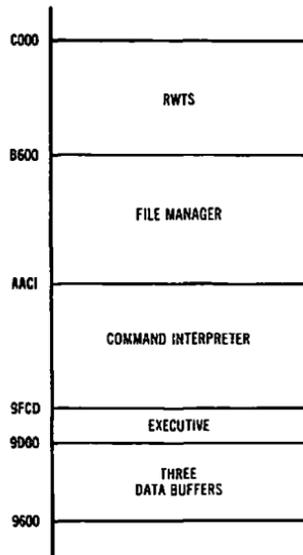
2.1.5 The BASIC Input/Output Map

There are two types of I/O in the Apple — built-in and peripheral. Built-in I/O is confined to the $C000.C07F region while the peripheral I/O has the rest of the space to $CFFF. Built-in I/O includes such things as the games socket, cassette, and speaker. Peripheral I/O is divided into seven slots, each with hardware address space and firmware address space. Each slot *owns* a chunk of sixteen locations in the $C080.C0FF area for its hardware and a chunk of 256 locations in the $C100.C7FF area for its firmware. In addition, each peripheral board may have a 2K block of memory in the $C800.CFFF space that it must share with its neighbors.
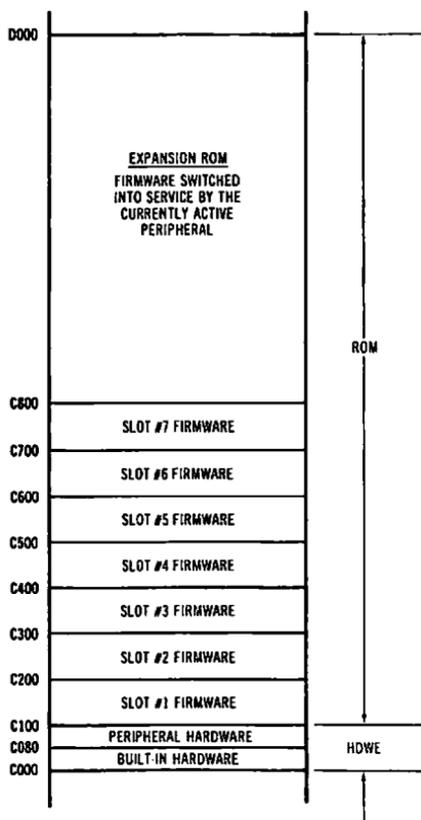
Fig. 2-5. The BASIC Input/Output map.

2.1.6 The BASIC Firmware Memory Map

At power up, the Apple II always has ROM to address at the top of memory. The 6502 needs it for the RESET routine that it first runs, and the address of the routine which it expects at the top of memory. The Monitor may be the old Standard Monitor that comes up with an asterisk at power up, the Autostart Monitor that bootstraps disks, or the IIe Monitor. Below that, BASIC resides. The Apple II came with Integer; the II Plus and IIe come with Applesoft.

If you have a RAM card in Slot Zero or 64K of RAM on the motherboard (like the IIe) then the ROM may be bank switched to RAM. The 16K of RAM is broken up into 8K and two 4K banks that can be selected to make 12K of RAM.

FFFF

(MONITOR)

F800

(BASIC)

8K
RAM

12K
ROM

APPLESOFT
OR
E000   INTEGER
BASIC
MAY BE RESIDENT

4K   4K
RAM   RAM

D000

16K RAM MAY BE
ON CARD IN SLOT ZERO
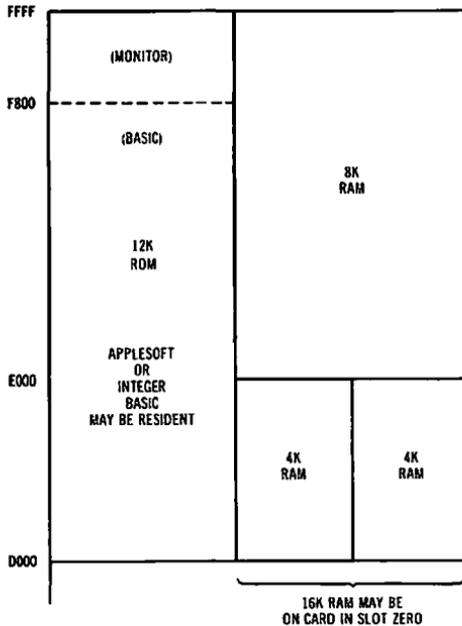
Fig. 2-6. The BASIC firmware memory map.

2.1.7 12K Bank Switching Between RAM and ROM

Bank switching is done by the I/O logic with the soft switches. To use a soft switch, you simply address a location in the $C000.C07F area. Some switches must be read, some written, some don't care. The write enabling switches, $C083 and $C08B, must be read twice for each switch to be effective. All the 12K bank switches can be done with read operations as shown in Table 2-1.

Table 2-1A. Switching Between RAM and ROM from Assembler (12K Bank)

Select RAM	Write Protect	Write Enable
4K Bank One	BIT $C088	BIT $C08B BIT $C08B
4K Bank Two	BIT $C080	BIT $C083 BIT $C083
Select ROM	(with RAM write protected)	
	BIT $C082	

Table 2-1B. Switching Between RAM and ROM from Applesoft (12K Bank)

Select RAM	Write Protect	Write Enable
4K Bank One	X = PEEK(49288)	X = PEEK(49291) X = PEEK(49291)
4K Bank Two	X = PEEK(49280)	X = PEEK(49283) X = PEEK(49283)
Select ROM	(with RAM write protected)	
	X = PEEK(49282)	

2.1.8 The Video Display and I/O Logic Memory Map

The Input/Output logic sees only a portion of the main memory that the processor sees. It accesses one or two areas for video display: HIRES1, HIRES2, SCREEN1, and SCREEN2. It reads the Input/Output area to set the soft switches that tell it and the Memory Manager how to access memory. In addition to the main memory, the I/O logic accesses a character set ROM that is not seen by the processor.
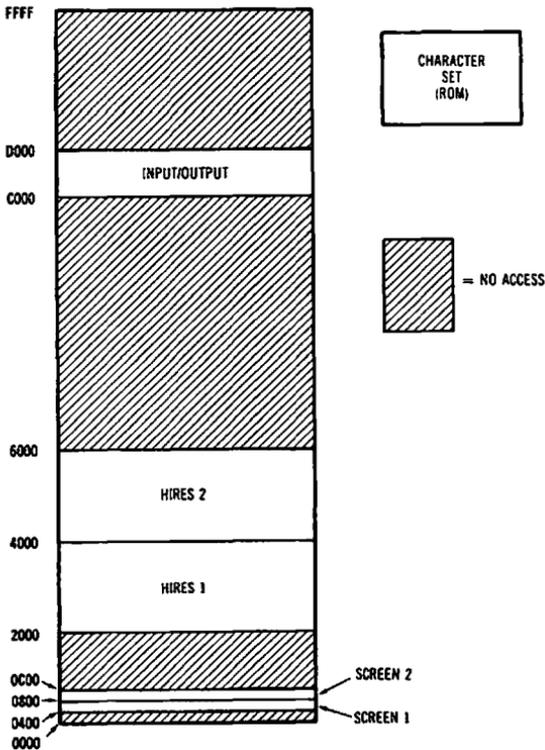
Fig. 2-7. The video display and I/O logic memory map.

2.1.9 The Apple IIe Memory Access Methods

The Apple IIe has several more soft switches than earlier models. This gives it access to more memory and adds more display modes. In addition to the 12K ROM and 64K RAM, the IIe allows up to 64K additional RAM in the auxiliary slot. The SCREEN1 and HIRES1 areas of this second RAM are accessed for video display. The character set ROM is increased to provide an *Alternate Character Set* in addition to the original *Primary Character Set*.
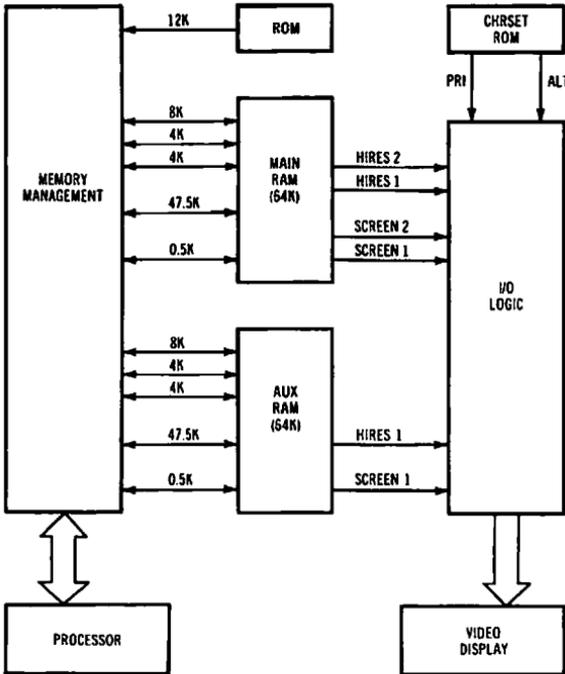
Fig. 2-8. Apple IIe memory access methods.

2.1.10 The Apple IIe Processor Memory Map

Fig. 2-9 shows what you can access with a IIe that has a 64K set of RAM in the Auxiliary slot. There is the normal 12K of ROM, the Input/Output area, and 48K of RAM that the IIe sets on power up. Like the II and the II Plus, you can switch ROM with the 16K RAM at the top of the main memory. In addition, you can switch to auxiliary memory, replacing the main 63.5K of memory with 63.5K of auxiliary memory. The bottommost 512 bytes are switched separately because of the processor's special needs for Pages Zero and One. With auxiliary RAM switched in, the ROM/RAM bank switches remain the same, keeping ROM in service or switching to the corresponding configuration of 12K RAM in auxiliary memory.
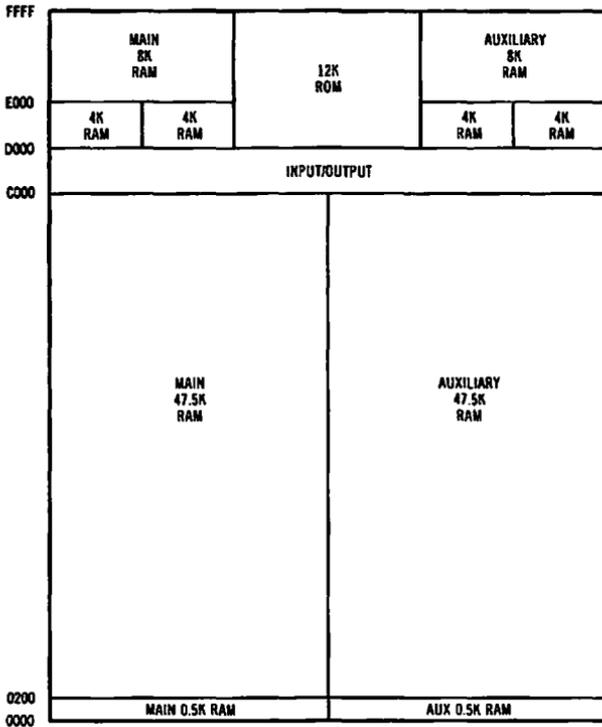
Fig. 2-9. The Apple IIe processor memory map.

2.1.11 The Apple IIe I/O Logic and Video Display Memory Map

Only a portion of the main and auxiliary RAMs are seen by the I/O logic. To have an 80-column display, some of the auxiliary memory can be used to cover SCREEN1 (if you don't need the extra memory for other uses). SCREEN2 is not accessible in auxiliary memory. You get a choice of SCREEN1 in main with SCREEN1 in auxiliary for 80-column work; or SCREEN1 in main with SCREEN2 in main for 40-column screen work. The same scheme works with the HIRES screen areas. In addition the IIe supplies two character sets — primary and alternate — to the I/O logic.
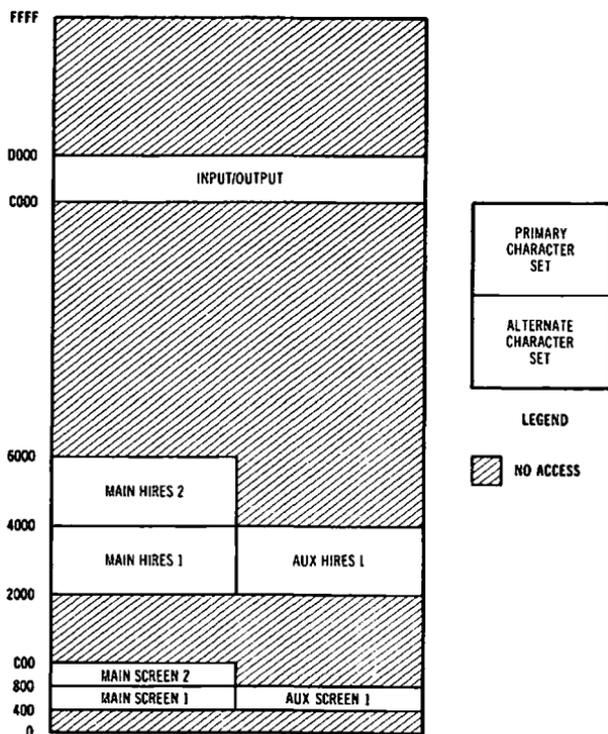
Fig. 2-10. The Apple IIe I/O logic and video display memory map.

2.1.12 Apple IIe Processor Access Soft Switches

The IIe *processor access soft switches* (Fig. 2-11) program the Memory Management to set the memory map of the processor. Switches requiring a write are shown as STA instructions.

The $C054.C055 switch lets you write (but not read) Auxiliary Memory as a convenience when programming for an 80-column display. If you want Auxiliary memory just as memory, then you can forgo the 80-column display and use the three read and write switches.

Always reset the 12K bank switches after switching between Main and Alternate. The same switches point into the corresponding areas in each.

Follow the flowchart to get the map you want.

2.1.13 Apple IIe Video Display Access Soft Switches

The IIe *video display access soft switches* (Fig. 2-12) program the I/O logic to set the memory map of the video display. As with Memory Management switches, those requiring writes are shown as STA instructions.

Be careful of the $C054.C055 switch on the IIe. It is used by the processor access to switch between Main and Alternate memory when writing for an 80-column display. Here, the *same* switch is used to select between SCREEN1 and SCREEN2 (or HIRES1 and HIRES2). Its usage is set by the $C000.C001 switch in Memory Management; this means that SCREEN2 and HIRES2 are not available for display on the IIe in 80-column mode.

In 40-column mode, $C054.C055 works just as it does in the Apple II models.

On the IIe, you must set the mode to 40- or 80-columns at $C00C.C00D consistent with the setting you made at $C000.C001. Plan carefully.

2.2. GAZETTEER

This section lists all memory locations within the BASIC 48K Memory Map referenced throughout this book. The most common configuration of DOS 3.3 and Applesoft BASIC is also presented in this section. The details of Integer BASIC are presented in Chapter Five.
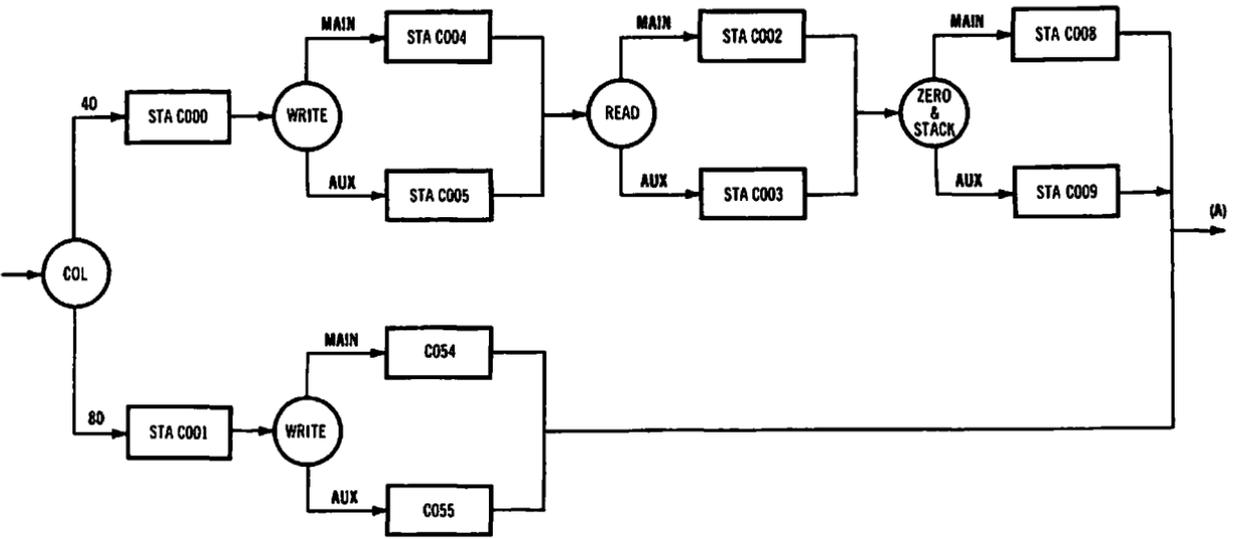
Fig. 2-11. Apple IIe processor access soft switches.
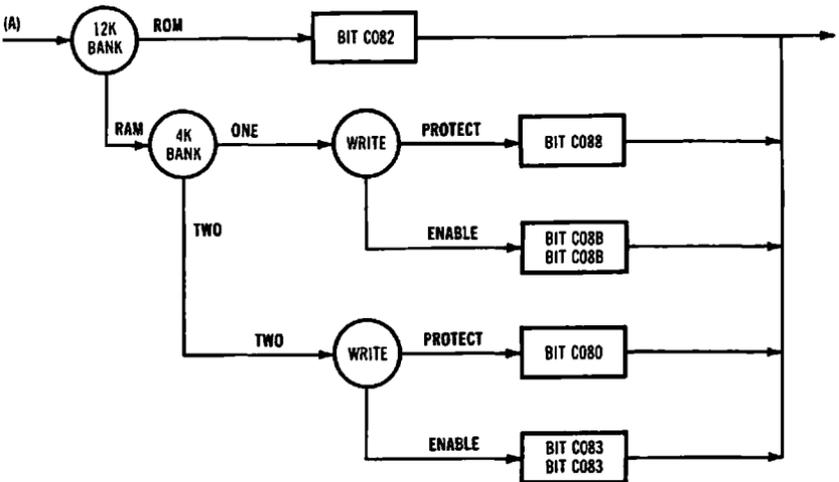
Fig. 2-11-cont. Apple IIe processor access soft switches.
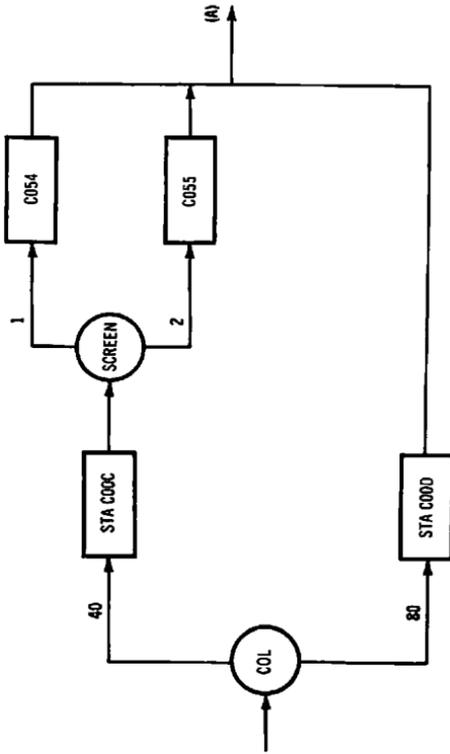
Fig. 2-12. Video display access soft switches.
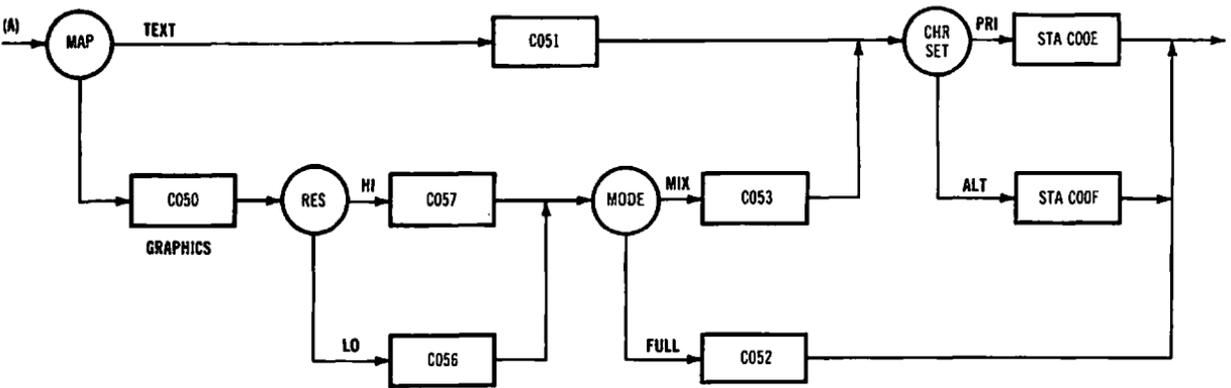
Fig. 2-12 — cont. Video display access soft switches.

Often, two consecutive locations will contain an address pointer, a line number, or other integral value. Unless otherwise noted, all such two-byte integers are in low-byte/high-byte order. This is the native format of the 6502 address operands, so it is used almost without exception throughout the Apple II.

For each entry, the hex address, the decimal address, and the commonly used label — if any — are given. For hardware addresses and Monitor addresses, negative decimal form is given as well for the convenience of Integer BASIC programmers.

This gazetteer is not exhaustive. Don't assume that a location is unused in Page Zero, for example, just because there is no entry. Applesoft, DOS, and the Monitor all use Page Zero in many locations.

2.2.1. Pages Zero and One

These have special meaning to the processor: Page Zero supports indirect addressing for indexing and Page One is the 6502's stack memory.

$0A.0C	(10)	USR

Contains a $4C as the JMP op code, followed by the address of the Applesoft USR function.

$16.	(22)	

The compare parameter used by the routine at $DF6A in Applesoft's floating-point package. It must be set to one of the following codes before calling $DF6A:

CODE	FOR COMPARISON
1	ARG > FAC
2	ARG = FAC
3	ARG < FAC
4	ARG > = FAC
5	ARG < > FAC
6	ARG < = FAC

$20	(32)	WNDLFT

Left margin of the scroll window. The TEXT command sets it to zero; you can set it from zero to $27 (39) for 40-column display.

$21	(33)	WNDWDTH

Width of the scroll window. The TEXT command sets it to $27 (39) for 40-column display. You can set it to any value from zero to $20 (contents of WNDLFT) in 40-column mode. A common trick of Applesoft programmers is to POKE 33,33 to get listings on screen without blanks added within literal quotes.

$22	(34)	WNDTOP

Top of the scroll window. The TEXT command sets it to zero for the topmost line. Range is to $17 (23) for the bottommost line.

$23	(35)	WNDBTM

Bottom of the scroll window. The TEXT command sets it to $17 (23). You can change it to any number less than $17 and greater than or equal to WNDTOP.

$24	(36)	CH

Horizontal text cursor ranges from zero to $27 (39) in 40-column mode. It locates the cursor from the left window. It is maintained by VIDOUT at $FBFD.

$25	(37)	CV

Vertical text cursor ranges from zero to $17 (23). It is always relative to the top of screen, not to WNDTOP. CV is used by VTAB at $FC22 in calculating the screen address at BAS, $28.29.

$28.29	(40)	BAS

Base address of the text cursor. It is calculated within the VIDOUT routines from WNDLFT at $20 and CV at $25 to give the left-most position on the cursor's line.

$2C	(44)	H2

End of line position. Used by HLINE at $F819 in the LORES graphics section of the Monitor.

$2D	(45)	V2

End of line position. Used by VLINE at $F828 in the LORES graphics section of the Monitor.

$2E	(46)	CHKSUM

Used as checksum by tape READ at $FEFD and WRITE at $FECD. Initialized to zero, then EORed to each byte of the data. It is written as the last byte following the data; read and compared to the calculated CHKSUM during the READ.

$30	(48)	COLOR

The current LORES color value, repeated in both nibbles. The sixteen values allowed are set by the COLOR = statement and become one of the following:

$00 black	$88 brown
$11 magenta	$99 orange
$22 dark blue	$AA grey
$33 purple	$BB pink
$44 dark green	$CC light green
$55 grey	$DD yellow
$66 medium blue	$EE aqua
$77 light blue	$FF white

$32	(50)	INVFLG

Mask intended to select inverse, normal, or flash characters display: normal is $FF, inverse is $7F, flash is $3F in value.

$33	(51)	PROMPT

Prompt character code to be displayed by Monitor GETLIN routine at $FD6A. Used by both BASICs and the Monitor's command

interpreter: Applesoft uses $9D for " "; Integer uses $BE for " "; and the Monitor uses $AA for "*".

$34	(52)	YSAV

Used by Monitor command interpreter.

$35	(53)	YSAV1

Used by Monitor command interpreter.

$36.37	(54)	CSW

System output hook address. Used by calling COUT at $FDED as the current output routine. CSW contains the address of the current output device; it defaults to COUT1 at $FDF0. See Chapter Six for a full explanation of the output hook.

$38.39	(56)	KSW

System input hook address. Used by calling RDKEY at $FD0C as the current input routine. KSW contains the address of the current output device; it defaults to KEYIN at $FD1B. See Chapter Six for a full explanation of the input hook.

$3A.3B	(58)	PC

This is where the Monitor command interpreter keeps the 6502 program counter for use by the ctrl/E command. The IRQ interrupt handler also stores the interrupted program address here to be read by the BREAK handler. See Chapter Three for details.

$3C.3D	(60)	A1

Used extensively by the Monitor commands: subtract, move, verify, tape I/O, and any other source address. DOS uses this location; the RWTS routine points to the DCT from here.

$3E.3F	(62)	A2

Used extensively by the Monitor commands: add, subtract, move, verify, tape I/O, and any other second source address.

$40.41   (64)   A3

Used by Monitor command interpreter. DOS uses this for the File Buffer address.

$42.43   (66)   A4

Used by the Monitor commands: move and verify. Can be the destination address for any other command defined. DOS uses it as the buffer address pointer.

$44.45   (68)   A5

Used by the Monitor command interpreter. Used as a file buffer pointer by DOS, see $A792.

$45.49   (69)   ACC, XREG, YREG,
                                                STATUS, SPNT

Registers storage used by Monitor G command and by the BRK routine:

$45 is ACC
$46 is XREG
$47 is YREG
$48 is STATUS, the P-reg
$49 is SPNT, the S-reg

The ctrl/E command reads these and PC at $3A.3B as well. From DOS, the RWTS routine uses $48.49 as the address of the IOB. After using RWTS, you must zero $48 to avoid any Monitor hang-ups.

$4A.4B   (74)   LOMEM

System pointer to lowest available user program location in Integer BASIC; normally $0800.

$4C.4D	(76)	HIMEM

System pointer to highest available user program location in Integer BASIC; normally $9600. The HIMEM: command will be used by DOS to set $4C.4D, regardless of the BASIC.

$4E.4F	(78)	RND

Random number generated by the keyboard input routine. You get a new random number with each keystroke.

$50.51	(80)	LINNUM

Integer value converted from FAC by the GETADR routine at $E752) in Applesoft.

$67.68	(103)	TXTTAB

Applesoft pointer to start of BASIC program text. It normally points to $0801.

$69.6A	(105)	VARTAB

Applesoft pointer to start of BASIC program's variables storage, one byte beyond the end of program text.

$6B.6C	(107)	ARYTAB

Applesoft pointer to the start of BASIC program's array storage; the end of program's variables.

$6D.6E	(109)	STREND

Applesoft pointer to the end of BASIC program storage. The unused free space begins here, one byte beyond the last array.

$6F.70	(111)	FRETOP

Applesoft pointer to the first byte of working string storage, one byte beyond the last free location.

$71.72	(113)	FRESPC

Applesoft pointer at the string storage area, used when a new string is created. See STRINI, at $E3D5.

$73.74	(114)	MEMSIZ

Applesoft pointer to the first byte past the last byte used for string storage. This is the highest RAM address available to Applesoft; it is normally set to $9600.

$75.76	(117)	CURLIN

Applesoft current line number. In direct mode, $FFFF.

$77.78	(119)	OLDLIN

Applesoft last line executed.

$79.7A	(121)	NXTPTR

Applesoft pointer to next BASIC program statement to be executed.

$7B.7C	(123)	DATLIN

Applesoft line number of current DATA statement. Used by READ.

$7D.7E	(125)	DATPTR

Applesoft pointer to DATA to be READ next.

$83.84	(131)	VARPNT

Applesoft pointer to variable as fetched by PTRGET at $DFE3.

$8A.8E	(138)	TEMP3

Applesoft temporary FP register, packed format.

$93.97	(147)	TEMP1

Applesoft temporary FP register, packed format.

$98.9C	(152)	TEMP2

Applesoft temporary FP register, packed format.

$9B.9C	(155)	LOWTR

Applesoft pointer to the address of an entire array, as fetched by GETARYPT at $F7D9.

$9D.9F	(157)	DSCTMP

Applesoft string descriptor: length, addr-lo, addr-hi. Used by STRINI at $E3D5 when creating new string storage.

$9D.A2	(157)	FAC

Applesoft floating-point accumulator. Unpacked format as follows:

$9D exponent in excess-$80
$9E mantissa, MSByte
$9F mantissa
$A0 mantissa
$A1 mantissa, LSByte
$A2 sign in Bit 7

A zero exponent signifies a zero value for the number.

$A5.AA	(165)	ARG

Applesoft argument register. FP number in unpacked format as follows:

$A5 exponent, excess-$80
$A6 mantissa, MSByte
$A7 mantissa
$A8 mantissa
$A9 mantissa, LSByte
$AA sign in Bit 7

A zero exponent signifies a zero value for the number.

$B1	(177)	CHRGET

Applesoft routine to get next BASIC character. Location $B8.B9 — TXTPTR — is incremented then used to read the character into the X-reg. If numeric, $30 to $39, the C-flag is cleared; otherwise, it is set. If it is a delimiter separating statements, like $3A (colon) or $00 (end of line), the Z-flag is set; otherwise, it is cleared.

$B7	(183)	CHRGOT

Applesoft routine to *re-get* a character previously *fetched* by CHRGET. Works like CHRGET except that TXTPTR is not incremented; instead the current character pointed to by TXTPTR is read. Flags returned as per CHRGET.

$B8.B9	(184)	TXTPTR

Applesoft pointer to the current character in BASIC program text. Used by CHRGET and CHRGOT by being embedded in the routine.

$D6	(214)

Normal value is $55. Saved and loaded by the cassette READ and WRITE commands as the third address byte of Applesoft BASIC programs. The Applesoft command interpreter traps this value so that Applesoft won't work in direct mode if it is greater in value than $7F.

$D8	(216)	ERRFLG

Set to $80 by Applesoft's ONERR GOTO statement to flag the error trap to the BASIC routine. This ONERR routine must clear it to inhibit further traps to itself; use a POKE 216,0.

$DA.DB	(218)	ERRLIN

Line number in BASIC at which an error occurred. This can be useful to your ONERR routine.

$E8.E9	(232)	SHADDR

Shape table address; normally set by the SHLOAD statement in Applesoft. Otherwise, you must set it prior to using any shape table commands. See Chapter Six.

$0100.0110   (256)   FBUFFR

String buffer for the FOUT at $ED34, which creates a string representation of the value in FAC ($9D.A2).

$0100.01FF   (256)   STACK

Processor stack address space. The processor builds its stack downwards in memory; the Monitor initializes the S-reg to $FF so as to point to $01FF.

2.2.2 Pages Two and Three

These two pages are set up by the Monitor at RESET for system functions. See Example 2-1.

Example 2.1 Dump of Apple II Vectors in Page Three

```
*3D0.3FF

03D0 — 4C  BF  9D  4C  84  9D  4C  FD
03D8 — AA  4C  B5  B7  AD  0F  9D  AC
03E0 — 0E  9D  60  AD  C2  AA  AC  C1
03E8 — AA  60  4C  51  A8  EA  EA  4C
03F0 — 59  FA  BF  9D  38  4C  58  FF
03F8 — 4C  65  FF  4C  65  FF  65  FF
```

$0200.02FF   (512)   IN

Input line buffer. Used by GETLIN at $FD6A to get input records from the current input device. The record is any length, up to 255 characters, with the last character a CR.

$0300.03CF   (768)

Small memory block, often used for short machine language routines.

$03D0.03D2   (976)

Jump to DOS *warm start* routine at $9DBF. This routine does not return.

$03D3.03D5   (979)

Jump to DOS *cold start* routine at $9D84. This routine does not return.

$03D6   (982)

Jump to DOS File Manager at $AAFD. This routine returns to the caller.

$03D9   (985)

Jump to DOS RWTS at $B7B5. This routine returns to the caller.

$03DC   (988)

Routine to fetch File Manager parameter reference from DOS. Address is returned in Y-reg (low) and A-reg (high).

$03E3   (995)

Routine to fetch RWTS parameter reference (IOB) from DOS. Address is returned in Y-reg (low) and A-reg (high).

$03EA   (1002)

Jump to DOS routine at $A851 that reconnects input and output hooks.

$03EF   (1008)   BRKV

Jump to BRK handler. Not available in Standard Monitor; normally $FA59 in Autostart Monitor. Also available in Apple IIe Monitor.

$03F2   (1010)   SOFTEV

Address of RESET handler; not available in Standard Monitor. Normally set to $9DBF by DOS 3.3 with Autostart Monitor. Also available in Apple IIe.

$03F4	(1012)	PWREDUP

Powered-up byte value indicating warm start to the RESET routine in Autostart or IIe Monitor; it is not available in the Standard Monitor. Set by SETPWRC at $F6BF. to the EOR of #$A5 and $03F5.

$03F5	(1013)	AMPERV

Jump to user's ampersand call from Applesoft BASIC.

$03F8	(1016)	USRADR

Jump to user's ctrl/Y Monitor extension command. RESET to MON at $FF65.

$03FB	(1019)	NMI

Jump to NMI interrupt handler. Rarely used; set to MON at $FF65. Vectored directly from $FFFA.FFFB.

$03FE	(1022)	IRQLOC

Address of IRQ interrupt handler. Control passed from $FA40 routine after saving A-reg at $45 and ensuring the B-flag is clear.

2.2.3 Display Screens

$0400.07FF	(1024)	SCREEN1

One K of RAM mapped by the I/O logic to the video display. Forty bytes map to each of twenty-four rows on the screen. Sixty-four bytes don't display; they are *scratch pad* RAM for the peripherals. See Tables 2-2 and 2-3 for details. Each display byte may represent either two LORES pixels or one character code.

Table 2-2. One K Screens

SCREEN1		SCREEN2	
Address	Use	Address	Use
$400.427	Row 0	$800.827	Row 0
$428.44F	Row 8	$828.84F	Row 8
$450.477	Row 16	$850.877	Row 16
$478.47F	Peripherals	$878.87F	Unused
$480.4A7	Row 1	$880.8A7	Row 1
$4A8.4CF	Row 9	$8A8.8CF	Row 9
$4D0.4F7	Row 17	$8D0.8F7	Row 17
$4F8.4FF	Peripherals	$8F8.8FF	Unused
$500.527	Row 2	$900.927	Row 2
$528.54F	Row 10	$928.94F	Row 10
$550.577	Row 18	$950.977	Row 18
$578.57F	Peripherals	$978.97F	Unused
$580.5A7	Row 3	$980.9A7	Row 3
$5A8.5CF	Row 11	$9A8.9CF	Row 11
$5D0.5F7	Row 19	$9D0.9F7	Row 19
$5F8.5FF	Peripherals	$9F8.9FF	Unused
$600.627	Row 4	$A00.A27	Row 4
$628.64F	Row 12	$A28.A4F	Row 12
$650.677	Row 20	$A50.A77	Row 20
$678.67F	Peripherals	$A78.A7F	Unused
$680.6A7	Row 5	$A80.AA7	Row 5
$6A8.6CF	Row 13	$AA8.ACF	Row 13
$6D0.6F7	Row 21	$AD0.AF7	Row 21
$6F8.6FF	Peripherals	$AF8.AFF	Unused
$700.727	Row 6	$B00.B27	Row 6
$728.74F	Row 14	$B28.B4F	Row 14
$750.777	Row 22	$B50.B77	Row 22
$778.77F	Peripherals	$B78.B7F	Unused
$780.7A7	Row 7	$B80.BA7	Row 7
$7A8.7CF	Row 15	$BA8.BCF	Row 15
$7D0.7F7	Row 23	$BD0.BF7	Row 23
$7F8.7FF	Peripherals	$BF8.BFF	Unused

$0800.0BFF   (2048)   SCREEN2

One K of RAM mapped by the I/O logic to the video display. Works
like SCREEN1 except that the sixty-four undisplayed bytes are un-
used. See Table 2-2.

$0800.95FF   (2048)

Table 2-3. SCREEN1 Peripherals Usage

DOS	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7
$478	$479	$47A	$47B	$47C	$47D	$47E	$47F
$4F8	$4F9	$4FA	$4FB	$4FC	$4FD	$4FE	$4FF
$578	$579	$57A	$57B	$57C	$57D	$57E	$57F
$5F8	$5F9	$5FA	$5FB	$5FC	$5FD	$5FE	$5FF
$678	$679	$67A	$67B	$67C	$67D	$67E	$67F
$6F8	$6F9	$6FA	$6FB	$6FC	$6FD	$6FE	$6FF
$788	$789	$78A	$78B	$78C	$78D	$78E	$78F
$7F8	$7F9	$7FA	$7FB	$7FC	$7FD	$7FE	$7FF

RAM normally available for BASIC program use. Applesoft requires $0800 to be zero; the BASIC program text starts at $0801. This block ends before $9600, the start of DOS buffers.

$2000.3FFF   (8192)   HIRES1

Eight K of RAM where each one K maps to one line in all twenty-four rows on screen. Each line is forty bytes in memory; there are eight lines per row. The sixty-four bytes not displayed in each K are unused. See Tables 2-4 to 2-11 for the significance of each location.

$4000.5FFF   (16384)   HIRES2

Eight K of RAM that maps like HIRES1. To look up the significance of a location, subtract $2000 and use Tables 2-4 to 2-11.

2.2.4 DOS 3.3

$9600.9CF8   (38400, −27136)

File buffers for normal DOS with MAXFILES of three, they break down as follows:

$9600.96FF   File 3:   Data buffer
$9700.97FF   File 3:   TSL buffer
$9800.982C   File 3:   Status
$982D.984A   File 3:   Filename, $00 if free

$94B.984C	File 3:	pointer to Status ($9800)
$984D.984E	File 3:	pointer to TSL buffer ($9700)
$984F.9850	File 3:	pointer to Data buffer ($9600)
$9851.9852	File 3:	link next file ($0000, end)
$9853.9952	File 2:	Data buffer
$9953.9A52	File 2:	TSL buffer
$9A53.9A7F	File 2:	Status
$9A80.9A9D	File 2:	Filename, $00 if free
$9A9E.9A9F	File 2:	pointer to Status ($9A53)
$9AA0.9AA1	File 2:	pointer to TSL buffer ($9953)
$99A2.99A3	File 2:	pointer to Data buffer ($9853)
$99A4.99A5	File 2:	link next file ($982D, File 3)
$9AA6.9BA5	File 1:	Data buffer
$9BA6.9CA5	File 1:	TSL buffer
$9CA6.9CD2	File 1:	Status
$9CD3.9CF0	File 1:	Filename, $00 if free
$9CF1.9CF2	File 1:	pointer to Status ($9CA6)
$9CF3.9CF4	File 1:	pointer to TSL buffer ($9BA6)
$9CF5.9CF6	File 1:	pointer to Data buffer ($9AA6)
$9CF7.9CF8	File 1:	link next file ($9A80, File 2)

For a breakdown of Status, see $B5D1.

$9D00.9D0F   (40192, −25344)

DOS relocatable pointers; values given here are for 48K system:

$9D00.9D01	link first file buffer ($9CD3, File 1)
$9D02.9D03	pointer to DOS input hook routine ($9E81)
$9D04.9D05	pointer to DOS output hook routine ($9EBD)
$9D06.9D07	pointer to primary file name buffer ($AA75)
$9D08.9D09	pointer to secondary file name buffer ($AA93)
$9D0A.9D0B	address of LOAD length parameter ($AA60)
$9D0C.9D0D	address of DOS load ($9D00)
$9D0E.9D0F	address of File Manager parameters from the DOS commands ($B5BB)

$9D84   (40324, −25212)

Cold start routine; vectored from $03D3. Jumps to BASIC cold start at $E000 on exit; does not return to caller.

Table 2-4. HIRES1 — The First K
(first lines of eight in each row)

Address	Row	Line	Y-coord
$2000.2027	0	0	$BF
$2028.204F	8	64	$7F
$2050.2077	16	128	$3F
$2078.207F		Unused	
$2080.20A7	1	8	$B7
$20A8.20CF	9	72	$77
$20D0.20F7	17	132	$37
$20F8.20FF		Unused	
$2100.2127	2	16	$AF
$2128.214F	10	80	$6F
$2150.2177	18	144	$2F
$2178.217F		Unused	
$2180.21A7	3	24	$A7
$21A8.21CF	11	88	$67
$21D0.21F7	19	152	$27
$21F8.21FF		Unused	
$2200.2227	4	32	$9F
$2228.224F	12	96	$5F
$2250.2277	20	160	$1F
$2278.227F		Unused	
$2280.22A7	5	40	$97
$22A8.22CF	13	104	$57
$22D0.22D7	21	168	$17
$22D8.22DF		Unused	
$2300.2327	6	48	$8F
$2328.234F	14	112	$4F
$2350.2377	22	176	$0F
$2378.237F		Unused	
$2380.23A7	7	56	$87
$23A8.23CF	15	120	$47
$23D0.23F7	23	184	$07
$23F8.23FF		Unused	

$9DBF   (40383, −25153)

Warm start routine; vectored from $03D0. Jumps to BASIC warm start at $E003 on exit; does not return to caller.

$9E42   (40514, −25022)

Patch point to allow Binary HELLO slave disk INITialization. Replace value with $34 (52).

$9E81   (40577, −24959)

Keyboard input routine. This is placed in KSW ($38.39) when DOS is
in effect. It uses ctrl/Ds and CRs to control its state and gets
characters accordingly.

$9EBD   (40637, −24899)

Output routine. This is placed in CSW ($36.37) when DOS is in effect.
It outputs characters according to its state.

$A251   (41553, −23983)

MAXFILES command handler. Outstanding EXEC file is turned off,
all files closed, MAXFILES value set at $AA57, then it uses pointer at
$9D00 to rebuild file buffers with a routine at $A7D4.

$A764   (42852, −22684)

Routine to search for a free file buffer. The free buffer pointer is re-
turned in $44.45 with $45 zero if no free buffer was found. Uses
$40.41 in search.

$A792   (42898, −22638)

Sets $40.41 to address of first file buffer; uses $9D00.9D01 as its
source.

$A79A   (42906, −22630)

Given $40.41 pointing to a file buffer, it finds the next file buffer in
the chain. Upon return, $40.41 points to the next buffer.

$A7D4   (42964, −22572)

Routine to rebuild file buffers. Enter with all files closed, link to first
new buffer in $9D00.9D01, and the number of buffers in $AA57.

$A851   (43089, −22447)

Routine to connect DOS hooks; vectored from $030A. If CSW at $36.37 does not point to the DOS output routine at $9EBD, then the current output hook is removed from CSW to $AA55.AA56 and replaced by $9EBD. Similarly, if KSW at $38.39 does not point to the DOS input routine at $9E81, then the current input hook is removed from KSW to $AA53.AA54 and replaced by $9E81. This routine returns to the caller.

$AA53.AA54   (43603, −21933)

Current system output hook as copied from CSW by the routine at $A851. When DOS intercepts a PR#0 command, it is reset to COUT1; see Chapter Six for details.

$AA55.AA56   (43605, −21931)

Current system input hook as copied from KSW by the routine at $A851. When DOS intercepts an IN#0 command, it is reset to KEYIN; see Chapter Six for details.

$AA57   (43607, −21929)

Value of MAXFILES parameter; usually three. See $A251.

$AA60.AA61   (43616, −21920)

Length for LOAD and BLOAD command routines.

$AAC1.AAC8   (43713, −21823)

Table of file manager addresses:

$AAC1.AAC2   address of IOB ($B7E8)
$AAC3.AAC4   address of VTOC buffer ($B3BB)
$AAC5.AAC6   address of Directory buffer ($B4BB)
$AAC7.AAC8   address of end-of-DOS ($C000)

$AAFD   (43773, −21763)

File Manager entry; vectored from $03D6.

$AE34   (44596, −20940)

Patch point to remove pause in display during a long CATALOG. Replace byte with $60 (96).

$B3BB.B4BA   (46011, −19525)

VTOC buffer. Both File Manager and DOS use the VTOC in Track 17/Sector 0 continually to maintain the disk. See Table 2-7 for the layout of VTOC.

$B4BB.B5BA   (46260, −19276)

Directory buffer. Both File Manager and DOS use the Directory in Track 17 to maintain files on disk. See Table 2-7 for the layout of a Directory sector.

$B5BB.B6BA   (46253, −19283)

Parameters for File Manager as passed from DOS. Referenced by $03DF, the parameters are described in Section 7.

$B5D1.B5FD   (46545, −18991)

Status of current file of the File Manager. Read from the file's status buffer when File Manager is called, then restored to the file when the File Manager command is finished. Normal file status for the three DOS buffers are at $9800, $9AF3, and $9CA6.

Byte	Content
0,1	Link (T/S) to TSL beginning
2,3	Link (T/S) to current TSL sector
4	Flags: used for check pointing
5,6	Link (T/S) to current data sector
7	Link (S) to current Directory sector
8	Index to file entry in Directory sector
9,A	Number of sectors content of TSL
B,C	Relative sector number of first sector in TSL
D,E	Relative sector number of last sector in TSL, −1
F,10	Relative sector number of last sector read

Byte	Content
11,12	Sector length in bytes
13,14	File position: sector offset
15,16	File position: byte offset
17,18	Record size from OPEN
19,1A	Record number
1B,1C	Byte offset into record
1D,1E	Number of sectors in file
1F,24	Sector allocation area
25	File type
26	Slot times $10
27	Drive number
28	Volume number complemented
29	Track number

$B7B5	(47029, – 18507)	RWTS

Read/Write Track/Sector routine; vectored from $03D9. This routine disables interrupts.

B7E8.B7F8	(47080, – 18456)	IOB

The Input/Output Block parameters for RWTS ($B7B5). Referenced from Page Three ($03E3) and used by callers.

Byte	Location	Content
00	B7E8	$01, always
01	B7E9	Slot number times 16, usually $60
02	B7EA	Drive number: $01 or $02
03	B7EB	Volume number: $00 matches any
04	B7EC	Track number: $00. . .$22
05	B7ED	Sector number: $00. . .$0F
06.07	B7EE.B7EF	Address of DCT: $B7FB
08.09	B7F0.B7F1	Address of sector buffer
0A	B7F2	Unused
0B	B7F3	Bytes/sector: $00 for 256
0C	B7F4	Command code: $00 for Seek, $01 for Read, $02 for Write, $04 for Format

Byte	Location	Content
0D	B7F5	Error code: $08 for init, $10 for write protect, $20 for volume mismatch, $40 for drive (I/O) error
0E	B7F6	Volume number found
0F	B7F7	Slot (*16) found
10	B7F8	Drive found

$B7FB.B7FE   (47099, − 18437)   DCT

Device Control Table as referenced by the IOB at $B7EE. It has disk access hardware parameters:

$B7FB: device type, $00
$B7FC: phases per track, $01
$B7FD: motor ON count, $D8EF

$B800   (47104, − 18432   PRENIBBLE

Routine used by RWTS to convert a data buffer pointed to by $3E.3F to *six-bit* 21form in the disk buffers at $BB00.BC55. The algorithm shifts from 256 bytes to 342 bytes to get the results described in Section 7.2.

$B82A   (47146, − 18390)   WRITE

Routine to write a six-bit code to buffers at $BB00.BC55 to disk. It writes a data prefix, encodes the six-bit data using the Write Translate Table at $BA29.BC55, and writes a data suffix. It calls a 32-cycle routine at $B8B8 to write each byte.

$B8B8   (47288, − 18248)

Real-time routine to write a byte to disk; 32 cycles.

$B8C2   (47298, − 18238)   POSTNIBBLE

Routine used by RWTS to convert a data field in the disk buffers at $BB00.BC55 from six-bit to the data buffer pointed to by $3E.3F. The

algorithm shifts the 342 bytes into 256 full bytes; it is the inverse of PRENIBBLE at $B800.

$B8DC	(47324, − 18212)	READ

Routine to read a sector of data from disk. It waits for a data field prefix, decodes data bytes into six-bit using the Read Translate Table, and stores the result in the disk buffer at $BB00.BC55. Compare to WRITE at $B82A.

$B944	(47428, − 18108)	RDADDR

Routine that reads an address field from disk. It waits for an address field prefix, reads eight bytes, and converts them to four address bytes. The four bytes returned in Page Zero are:

$2C: checksum
$2D: sector number
$2E: track number
$2F: volume number

If error, the C-flag is set on return.

$BA96.BAFF	(47766, − 17770)

Read Translate Table used by READ at $B8DC. It contains six-bit values for encoded bytes in the $96.FF range. To use, an instruction like

LDA $BA00,X

where X is in the $96.FF range, will return the decoded six-bit value in the range $00.3F.

$BA29	(47657, − 17879)

Write Translate Table used by WRITE at $B82A. It contains encoded data values for six-bit bytes in the $00.3F range. To use, an instruction like

LDA $BA29,X

where X is in the $00.3F range, will return an encoded data value in the $96.FF range.

$BC56   (48214, −17322)

Routine to write an address field on disk. It is used by the initialization routines. It uses a real-time routine at $BCC4. It writes autosync bytes, prefix, header, and suffix bytes. To call, put header data in Page Zero:

$3E: must be $AA
$3F: sector number
$41: volume number
$44: track number

Upon return, an error sets the C-flag.

$BCC4   (48384, −17152)

Real-time routine to encode a header byte, writing it as two bytes to disk in 32 cycle loops.

$BB00.BC55   (47872, −17664)

Primary and Secondary disk buffers used by RWTS routines. When reading the disk, it stores the 342 six-bit data bytes decoded by the READ routine at $B82A; while writing, it must be given six-bit data for the WRITE routine at $B82A. The Primary buffer at $BB00.BBFF contains Byte 87 to Byte 342. The Secondary buffer at $BC00.BC55 contains Byte 86 to Byte 0 (descending sequence). Byte 342 at $BBFF is the checksum.

$BFD3.BFD5   (47107, −16429)

Patch point to remove forced BASIC language re-loads in the bank-switched RAM. Replace these three bytes with $EA (324) values. They are NOP op codes.

2.2.5 Input/Output

$C000.C00F	(49152, −16384)	KBD

When read, gives the keypress flag in Bit 7 and the character code in Bits 6 to 0. The strobe at $C010 must be reset after a keypress is detected, before another is anticipated. Be careful of conflict with 80STORE usage; see below.

$C000.C001	(49152, −16384)	80STORE

Soft switch; IIe only. Selects the second screen access to be the corresponding Auxiliary memory in 80-column mode:

$C000 selects 40-column mode
$C001 selects 80-column mode
$C018 reads this switch.

In 80-column mode, the $C055 switch will select Alternate screen memory for writing. In 40-column mode, it selects the second main screen memory for display.

$C002.C003	(49154, −16382)	RAMRD

Soft switch; IIe only. Selects one of two 63.5K memories for processor reads:

$C002 selects reads from Main memory
$C003 selects reads from Auxiliary memory
$C013 reads this switch

Used in 40-column mode. The processor's Pages Zero and One are not switched with the rest of memory.

$C004.C005	(49156, −16380)	RAMWRT

Soft switch; IIe only. Selects one of two 63.5K memories for processor writes:

$C004 selects writes to Main memory
$C005 selects writes to Auxiliary memory
$C014 reads this switch

Used in 40-column mode. The processor's Pages Zero and One are not switched with the rest of memory.

$C008.C009	(49160, – 16376)	ALTZP

Soft switch; IIe only. Selects one of two 0.5K memories for processor reads and writes:

$C008 selects Main memory
$C009 selects Auxiliary memory
$C016 reads this switch

Used in 40-column mode. Only the processor's Pages Zero and One are switched; the rest of memory is unaffected.

$C00C.C00D	(49164, – 16372)	80COL

Soft switch; IIe only. Sets I/O logic for video display mode:

$C00C sets 40-column display mode
$C00D sets 80-column display mode
$C01F reads this switch

In 80-column mode, the display interleaves bytes from Main and Auxiliary memory from corresponding addresses.

$C00E.C00F	(49166, – 16370)	CHRSET

Soft switch; IIe only. Sets I/O logic to select one of two character sets for text display:

$C00E selects Primary character set
$C00F selects Alternate character set
$C01E reads this switch

See Chapter Six for details of character sets.

$C010   (49168, – 16368)   KBDSTB

When read, clears the keyboard strobe on all models. On the IIe, it doubles as a keypress flag so that one read will both detect a keypress and clear the strobe. When written, it clears the strobe as well.

$C018.C01F   (49176, – 16360)

Reads IIe soft swtiches; reads IIe vertical blanking. Bit 7 is zero when switch is off; one when switch is on. The vertical blanking occurs when its switch is off (zero).

$C018 reads $C000.C001 80STORE
$C019 is the vertical blanking level
$C01A reads $C050.C051 TEXT
$C01B reads $C052.C053 MIXED
$CO1C reads $C054.C055 PAGE2
$C01D reads $C056.C057 HIRES
$C01E reads $C00E.C00F CHRSET
$C01F reads $C00C.C00D 80COL

$C020   (49184, – 16352)

Cassette tape output port. Reading this address toggles the OUT jack on the back panel of the Apple between zero and 25 millivolts. Don't use a write op code that will toggle the port twice upon each instruction.

$C030   (49200, – 16336)

Speaker port. Reading this address toggles the built-in speaker via a transistor amplifier on the motherboard. Don't use a write op code that will toggle the port twice upon each instruction.

$C040   (49216, – 16320)

Strobe output port. Reading this address brings Pin 5 on the games socket DIP low for a half cycle. Don't use a write op code that will create two pulses instead of one. Pin 5 is normally high.

$C050.C051   (49232, – 16304)   TEXT

Soft switch to select character text or graphics display:

$C050 selects graphics
$C051 selects text
$C01A reads this switch, IIe only

$C052.C053   (49234, – 16302)   MIXED

Soft switch to select mixed or full screen graphics:

$C052 selects full graphics
$C053 selects mixed graphics and text
$C01B reads this switch, IIe only

$C054.C055   (49236, – 16300)   PAGE2

Soft switch usually selects between first and second display screen. If in HIRES mode,

$C054 selects HIRES1 for display
$C055 selects HIRES2 for display

If in LORES mode,

$C054 selects SCREEN1 for display
$C055 selects SCREEN2 for display

If 80STORE at $C001 is set on the IIe model,

$C054 write enables Main memory
$C055 write enables Auxiliary memory

On the IIe model, $C01C reads this switch.

$C056.C057   (49238, – 16298)   HIRES

Soft switch to select graphics screen mode:

$C056 selects LORES display
$C057 selects HIRES display

$C01D reads this switch

$C058.C059   (49240, − 16296)

Annunciator port on Pin 15 of games DIP socket. Set by soft switches:

$C058 sets Annunciator 0 off (zero)
$C059 sets Annunciator 0 on (high)

$C05A.C05B   (49242, − 16294)

Annunciator port on Pin 14 of games DIP socket. Set by soft switches:

$C05A sets Annunciator 1 off (zero)
$C05B sets Annunciator 1 on (high)

$C05C.C05D   (49244, − 16292)

Annunciator port on Pin 13 of games DIP socket. Set by soft switches:

$C05C sets Annunciator 2 off (zero)
$C05D sets Annunciator 2 on (high)

$C05E.C05F   (49246, − 16290)

Annunciator port on Pin 12 of games DIP socket. Set by soft switches:

$C05E sets Annunciator 3 off (zero)
$C05F sets Annunciator 3 on (high)

$C060   (49248, − 16288)

Cassette tape input port. Bit 7 is toggled by a zero-crossing sector, using a 741 operational amplifier, at the IN jack on the back panel. Each zero crossing at the IN jack toggles Bit 7 between zero and one. The input circuit is nominally 12k ohms, 1 volt. The tape READ at $FEFD uses EORs to time the transition intervals.

$C061   (49249, − 16287)   SWO

Switched input port at Pin 2 of games DIP socket. Bit 7 gives the state of the switch: one is on, zero is off. It is used for the pushbutton on game paddles and joysticks. On the IIe model, it is also wired to the OPEN-APPLE key to indicate a forced cold start when RESET.

$C062	(49250, – 16286)	SW1

Switched input port at Pin 2 of the games DIP socket. Bit 7 gives the state of the switch: one is on, zero is off. It is used for the pushbutton on game paddles and joysticks. On the IIe model, it is also wired to the CLOSED-APPLE key to indicate a self-test to the RESET routine.

$C063	(49251, – 16285)	SW2

Switched input port at Pin 3 of the games DIP socket. Bit 7 gives the state of the switch: one is on, zero is off. On some old keyboards, it is connected to the shift key as part of a lower case scheme.

$C064	(49252, – 16284)	PDLO

Analog input port. Bit 7 is set to one by addressing $C070 and starting the four timers. At time out of the timer that connects to Pin 15 of the games DIP socket, bit 7 changes from one to zero. Time constant is 0.022 uF times the resistance at Pin 15.

$C065	(49253, – 16283)	PDL1

Analog input port. Bit 7 is set to one by addressing $C070 and starting the four timers. At time out of the timer that connects to Pin 14 of the games DIP socket, bit 7 changes from one to zero. Time constant is 0.022 uF times the resistance at Pin 14.

$C066	(49254, – 16282)	PDL2

Analog input port. Bit 7 is set to one by addressing $C070 and starting the four timers. At time out of the timer that connects to Pin 13 of the games DIP socket, bit 7 changes from one to zero. Time constant is 0.022 uF times the resistance at Pin 13.

$C067	(49255, – 16281)	PDL1

Analog input port. Bit 7 is set to one by addressing $C070 and starting the four timers. At time out of the timer that connects to Pin 12 of the games DIP socket, bit 7 changes from one to zero. Time constant is 0.022 uF times the resistance at Pin 12.

$C070	(49264, − 16272)	PDLSTRB

Analog timers strobe. A read instruction at this address generates a single strobe that starts the four timers, each with its own time constant. When started, the timer outputs go high at $C064.C067 until each times out.

$C080.C08F	(49288, − 16248)

Soft switches for bank-switched memory. Use read instructions only. The specific instructions to use for each case are given in Table 2-1.

$C090.C0FF	(49296, − 16240)

Device selects for Slots 1 to 7. Each of the sixteen addresses selects a slot by bringing its DS line, on Pin 41, low during Phase Zero:

$C090.C09F selects Slot 1
$C0A0.C0AF selects Slot 2
$C0B0.C0BF selects Slot 3
$C0C0.C0CF selects Slot 4
$C0D0.C0DF selects Slot 5
$C0E0.C0EF selects Slot 6
$C0F0.C0FF selects Slot 7

Normally, a peripheral card uses these sixteen addresses for hardware devices like interface chip registers.

$C100.C7FF	(49408, − 16128)

I/O selects for Slots 1 to 7. Each of the 256 addresses selects a slot by bringing its I/O SELECT line, on Pin 1, low during Phase Zero:

$C100.C1FF selects Slot 1
$C200.C2FF selects Slot 2

$C300.C3FF selects Slot 3
$C400.C4FF selects Slot 4
$C500.C5FF selects Slot 5
$C600.C6FF selects Slot 6
$C700.C7FF selects Slot 7

Normally, a peripheral card uses these 256 addresses for firmware to be selected by the PR#s and IN#s commands.

$C800.CFFF   (51200, – 14336)

I/O strobe on all Slots, Pin 20. The line goes low during Phase Zero for any address in this 2K space. It is used by many cards for firmware. By convention, address $CFFF is used to disable card memory, releasing the space and allowing more than one card to use it. See Section 8.2 for details.

2.2.6 Applesoft at $D000.F7FF

Applesoft may be in firmware on the motherboard, or BLOADed into bank-switched RAM. When installed, DOS is able to bank switch between Applesoft and Integer BASICs with the FP and INT commands.

$D823   (55331, – 10205)   LAM

Re-entry point, used to continue BASIC execution. Used in Lam's method for CALLing the Monitor command interpreter. See Section 3.1 for details of Lam's method.

$D995   (55701, – 9835)   DATA

Routine to advance TXTPTR to end of statement. Upon return, TXTPTR points to ":" or zero.

$DB3A   (56122, – 9414)   STROUT

Routine to print a string pointed to by Y-reg (low) and A-reg (high). String must end with a quote or a zero ($22 or $00).

$DD67   (56679, − 8857)   FRMNUM

Routine to evaluate a numerical expression at location given by TXTPTR. Result is in FAC.

$DEB8   (57016, − 8520)   CHKCLS

Routine to test the current character referenced by TXTPTR as a closing bracket, ")". If so, it returns via CHRGET advancing the TXTPTR to the next character. If not, it exits to SNERR at $DEC9.

$DEBB   (57019, − 8517)   CHKOPN

Routine to test the current character referenced by TXTPTR as an opening bracket, "(". If so, it returns via CHRGET advancing the TXTPTR to the next character. If not, it exits to SNERR at $DEC9.

$DEBE   (57022, − 8514)   CHKCOM

Routine to test the current character referenced by TXTPTR as a comma. If so, it returns via CHRGET advancing TXTPTR to the next character. If not, it exits to SNERR at $DEC9.

$DEC9   (57033, − 8503)   SNERR

Routine to print "SYNTAX ERROR" message, then warm start Applesoft at $E003. This routine does not return to the caller.

$DF6A   (57194, − 8342)

Routine to compare ARG and FAC, giving logical result in FAC. Compare code must be in Page Zero, $16:

Code in $16	FAC is TRUE, if
1	ARG > FAC
2	ARG = FAC
3	ARG < FAC
4	ARG $\geq$ FAC
5	ARG < > FAC
6	ARG $\leq$ FAC

where FAC has a value of one for TRUE, zero for FALSE.

$DFE3	(57315, −8221)	PTRGET

Routine to get a variable reference at the current TXTPTR position. Returns the address of variable contents in A-reg (high) and Y-reg (low), as well as in VARPNT at $83.84. If a variable does not exist, it is created. TXTPTR points to the next character.

$E000	(57344, −8192)	CTRLB

The BASIC cold start address; same for Applesoft or Integer. Initializes Applesoft, ignoring previous BASIC program. The BASIC signature byte gives the version in memory:

$20 (32) if Integer BASIC
$4C (76) if Applesoft BASIC

$E003	(57347, −8189)	

The BASIC warm start address; same for Applesoft or Integer. Preserves the current BASIC program.

$E07D	(57469, −8067)	ISLETC

Routine tests A-reg for ASCII *A* to *Z*. If letter, it returns with C-flag set; if not, the C-flag is clear.

$E0FE.E102	(57598, −7938)	

Constant value, −32768. Packed FP format.

$E2F2	(58098, −7438)	GIVAYF

Routine to float a signed integer in A-reg (high) and Y-reg (low) to FAC.

$E3D5	(58325, −7211)	STRINI

Routine to get space for a new string between (FRETOP) and (MEMSIZ). Call with length required in A-reg. On return, FRESPC

at $71.72 points to the new space and a complete descriptor is in DSCTMP at $9D.9F.

$E5E2	(58850, – 6686)	MOVSTR

Routine to move a string. Source address must be in Y-reg (high) and X-reg (low) with the length in A-reg. Destination address must be in FRESPC at $71.72.

$E6F8	(59128, – 6408)	GETBYT

Routine to reduce an expression in program text to a single byte value. Expression must be referenced with TXTPTR and have a value in the $00.FF range. Returns with TXTPTR advanced to the delimiter and the byte value in X-reg. If not within range, the value induces an IL-LEGAL QUANTITY error, stopping the program.

$E752	(59218, – 6318)	GETADR

Routine to correct value in FAC to an integer in LINNUM at $50.51. Result also appears in A-reg (high) and Y-reg (low).

$E7AA	(59306, – 6230)	FSUBT

Routine to subtract FP numbers. On entry, A-reg and Z-flag must reflect FAC exponent, at $9D; a JSR MOVFM at $EAF9 will do this. On exit, FAC = ARG – FAC.

$E7C1	(59329, =6207)	FADDT

Routine to add FP numbers. On entry, A-reg and Z-flag must reflect FAC exponent at $9D; a JSR MOVFM at $EAF9 will do this. On exit, FAC = ARG + FAC.

$E92D.E931	(59693, =5843)

Constant value, SQR (0.5). Packed FP format.

$E913.E917	(59667, =5869)

Constant value, 1.0. Packed FP format.

$E932.E936   (59698, −5838)

Constant value, SQR(2.0). Packed FP format.

$E937.E93B   (59703, −5833)

Constant value, −0.5. Packed FP format.

$E941   (59713, −5823)   LOG

Routine to get the natural logarithm. FAC = 1n(FAC).

$E982   (59778, −5758)   FMULTT

Routine to multiply FP numbers. On entry, A-reg and Z-flag must reflect FAC exponent at $9D; a JSR MOVFM at $EAF9 will do this. On exit, FAC = ARG * FAC.

$E9E3   (59875, −5661)   CONUPK

Routine to unpack FP number. Address of packed number must be in A-reg (high) and Y-reg (low). Result is in ARG at $A5.AA.

$EA50.EA54   (59984, −5552)

Constant value, 10.0. Packed FP format.

$EA69   (60009, −5527)

Routine to divide FP numbers. On entry, A-reg and Z-flag must reflect FAC exponent at $9D; a JSR MOVFM at $EAF9 will do this. On exit, FAC = ARG / FAC. Don't forget to test for a zero divisor; a BEQ just before the call will do this.

$EAF9   (60153, −5383)   MOVFM

Routine to unpack FP number. Address of packed number must be in A-reg (high) and Y-reg (low). Result is in FAC at $9D.A2. A-reg and Z-flag reflect exponent in $9D.

$EB2B   (60203, −5333)   MOVMF

Routine to pack FP number. Address of destination must be in A-reg (high) and Y-reg (low). Result is packed from FAC.

$EB53   (60243, −5293)   MOVFA

Routine to move FP number from ARG at $A5.AA to FAC at $9D.A2.

$EB63   (60259, −5277)   MOVAF

Routine to move FP number from FAC at $9D.A2 to ARG at $A5.AA.

$EB82   (60290, −5248)   SIGN

Routine to test sign of FAC. Result is in A-reg:

$01 if FAC > 0
$00 if FAC = 0
$FF if FAC < 0

$EB90   (60304, −5232)   SGN

Routine to test sign of FAC. Result is in FAC:

1.0 if FAC was > 0
0   if FAC was = 0
−1.0 if FAC was < 0

It uses SIGN at $EB82 and floats the result.

$EBAF   (60335, −5001)   ABS

Routine to change an FP number to its absolute value in FAC. Sign is forced positive.

$EBB2   (60338, −5198)   FCOMP

Routine to compare FAC with any packed FP number. The number must be referenced in A-reg (high) and Y-reg (low). The result is returned in A-reg:

$01 if value > FAC
$00 if value = FAC
$FF if value < FAC

$EC23	(60451, −5085)	INT

Routine to fix contents of FAC. Result is the next greatest integer: 34.6 becomes 34; −34.6 becomes −35. Result is in FAC at $9E.9F.

$ED34	(60724, −4812)	FOUT

Routine to convert the FP value in FAC to a string. The resulting string is in FBUFFR at $0100.0110.

$EE64	(61028, −4508)	

Constant value, 0.5. Packed FP format.

$EE8D	(61069, −4467)	SQR

Routine to convert the value in FAC to its square root.

$EE97	(61079, −4457)	FPWRT

Routine to calculate FP exponents. On entry, A-reg and Z-flag must reflect FAC exponent at $9D. On exit, FAC − ARG^{FAC}.

$EF09	(61193, −4343)	EXP

Routine to calculate FP exponent, base e = 2.71828 . . . , of FAC. Result is in FAC.

$EFAE	(61358, −4178)	RND

Replaces FAC with pseudorandom number, mathematically generated.

$EFEA   (61418, −4118)   COS

Routine to calculate the cosine of FAC. Result in FAC.

$EFF1   (61425, −4111)   SIN

Routine to calculate the sine of FAC. Result in FAC.

$F03A   (61498, −4038)   TAN

Routine to calculate the trigonometric tangent of FAC. Result in FAC.

$F063.F067   (61539, −3997)

Constant value, $\pi/2$. Packed FP format.

$F06B.F06F   (61547, −3989)

Constant value, 2π. Packed FP format.

$F070.F074   (61552, −3984)

Constant value, 0.25. Packed FP format.

$F09E   (61598, −3938)   ATN

Routine to calculate the arctangent of the FAC.

$F364   (62248, −3288)

Routine to remove ONERR GOTO stack entries as part of the RESUME statement. Can be called instead of a RESUME.

$F7D9   (63449, −2087)   GETARYPT

Routine to find an array variable. TXTPTR must point to the first character of the name to be found. The address of the array, the location of the array name in storage, is returned in LOWTR at $9B.9C. TXTPTR points to the next character past the name in program text.

2.2.7 Monitor at $F800.F8FF

All Monitors — Standard, Autostart, and IIe — can be referenced at the following addresses unless otherwise noted. By keeping to these entry points, the possibility of trouble when changing Monitors will be reduced. Making calls to other points should be done with care.

$F800   (63488, − 2048)   PLOT

Routine displays a LORES pixel on Screen1 using COLOR at $30. Caller puts line number in Y-reg and column number in A-reg; ranges of lines to $2F (47) and columns to $27 (39).

$F819   (63513, − 2023)   HLINE

Routine draws a horizontal line in LORES on Screen1 using COLOR at $30. Start and end coordinates must be given:

start X-coordinates in Y-reg
start Y-coordinates in A-reg
end X-coordinates in H2 at $2C
end Y-coordinates in A-reg

$F828   (63528, − 2008)   VLINE

Routine draws a vertical line in HIRES on Screen1 using COLOR at $30. Start and end coordinates must be given:

start X-coordinates in Y-reg
start Y-coordinates in A-reg
end X-coordinates in Y-reg
end Y-coordinates in V2 at $2D

$F832   (63538, − 1998)   CLRSCR

Routine to clear Screen1, row by row, to zeros. In LORES, this gives a black screen.

$F836   (63542, − 1994)   CLRTOP

Routine to clear the top twenty rows of Screen1 (forty HIRES lines) to zeros. This blacks the LORES display in mixed mode while the four rows of text are let alone.

$F864	(63588, −1948)	SETCOL

Routine to set COLOR at $30 to the doubled nibble value in the A-reg. A-reg must have code zero to fifteen.

$F871	(63601, −1935)	SCRN

Routine to get color code of the current LORES pixel. The coordinates of the pixel must be given as: X-coordinates in Y-reg and Y-coordinates in A-reg. Upon return, the code will be in the A-reg.

$F941	(63809, −1727)	PRNTAX

Prints a four digit hex number. Enter with high byte in A-reg and low byte in X-reg.

$F948	(63816, −1720)	PRBLNK

Prints three spaces.

$F94A	(63818, −1718)	PRBL2

Prints spaces (blanks). Number of spaces must be in A-reg.

$FB1E	(64286, −1250)	PREAD

Routine to read one analog input. Requires analog port number (0, 1, 2, 3) in X-reg and a resistance across that port − up to 150 K ohms. Value from $00 to $FF proportionate to the resistance is returned in Y-reg.

$FB2F	(64303, −1233)	INIT

Resets soft switches, screen window, and puts cursor at lower left of screen. Equivalent to a BASIC statement of TEXT.

$FB40	(64320, − 1216)	SETGR

Routine to set soft switches for LORES graphics in mixed mode. Clears the 40 by 40 pixel area with CLRTOP at $F836. Equivalent to a BASIC statement of GR.

$FBB3	(64435, − 1101)	

Monitor signature byte; identifies version:

$38 (56) in Standard
$EA (234) in Autostart
$06 (6) in IIe

$FBDD	(64477, − 1055)	BELL1

Routine to make "beep" sound. Tone of 1000 Hz.

$FC22	(64546, − 990)	VTAB

Routine to set cursor. Uses CV at $24, CH at $25, and WNDLFT at $20 for the Screen1 text.

$FC58	(64600, − 936)	HOME

Clears screen within scroll window; places cursor at upper left.

$FC58	(64680, − 856)	WAIT

Routine delays according to contents of A-reg:

$$\#cycles = 0.5(26 + 27A + 5A^2)$$

where one cycle is 0.977778 microseconds. See Section 8.1.

$FCC9	(64713, − 823)	HEADR

Routine to write a tape header tone and sync bit. Length of tone depends on A-reg: $40 is typical, for ten seconds. X-reg should be zero and C-flag should be set when called. See Section 8.1 for details.

$FD0C	(64780, −756)	RDKEY

Routine to advance the cursor of the built-in terminal, then input one character via the KSW hook at $36.37. See Section 6.1 for details on how the hooks work.

$FD1B	(64795, −741)	KEYIN

Routine to get one character from the built-in keyboard. It sets the random number at $4E.4F according to the time it waits for the keystroke. Before returning, it replaces the screen cursor with the previous character. The new character is returned in the A-reg.

$FD6A	(64874, −662)	GETLN

Routine to input a record. It displays the prompt character from $0020, then gets characters using the RDKEY routine at $FD0C. Any ESC ($1B) characters received initialize the escape sequence for the following character. The IIe model has more escape sequences in GETLN than previous models. Also, the IIe permits lower case. Previous models converted any lower case characters from the keyboard to upper case. This can be corrected with a patch at $FDE3: change it to a $FF value. See Section 6.2 for more on inputting. When a CR is received, the routine returns with the record in Page Two.

$FDDA	(64986, −550)	PRBYTE

Prints a two-digit hex number. Enter with value to be printed in A-reg. Together with PREAD at $FB1E, this routine is used often in testing and debugging machine language algorithms.

$FDED	(65005, −531)	COUT

System output call. It invokes the routine whose address is in CSW at $38.39. By convention, the character to be output must be supplied in A-reg.

$FDF0	(65008, −528)	COUT1

Routine to display a character using the built-in video. It interprets the character given in A-reg: displaying printable characters and invoking various control character routines. It uses all the Page Zero cursor and window parameters.

$FE2C	(65068, −468)	MOVE

Routine to move a block of memory according to the Monitor M command. To use directly, set A1 at $3C.3D to the source beginning address, A2 at $3E.3F to the source ending address, and A4 at $42.43 to the destination start address.

$FE89	(65161, −375)	SETKB

Routine to reset the input hook at $36.37 to the address of KEYIN at $FD1B. This is equivalent to the IN#0 command.

$FE93	(65171, −365)	SETVID

Routine to reset the output hook at $38.39 to the address of COUT1 at $FDF0. This is equivalent to the PR#0 command.

$FECD	(65229, −307)	WRITE

Routine to save a block of memory to tape. Set A1 at $3C.3D to the beginning address and A2 at $3E.3F to the ending address. The routine writes a 10 second header followed by the contents of the designated block of memory. The checksum of EORing all bytes is written last.

$FEFD	(65277, −259)	READ

Routine to read a block of memory from tape. Set A1 at $3C.3D to the beginning address and A2 at $3E.3F to the ending address. You must know the exact size of the tape file to do this. A running checksum is made at $2E using incoming bytes EORed together. If the final checksum fails to match the one at the end of file, an ERR message is output. Errors can be trapped by detecting a change in CH at $36; see Section 8.1.

$FF3A   **(65338, − 198)**   **BELL**

Routine to output a ctrl/G, $87, to the current output device.

$FF65   **(65381, − 155)**   **MON**

Cold start of Monitor command interpreter. It rings the bell and clears the D-flag before making the warm start described below.

$FF69   **(65385, − 151)**   **MONZ**

Warm start of Monitor command interpreter. It prompts with an asterisk − "*" − and interprets the resulting record as typed in by the user. The first character is the command mnemonic and may be followed by parameters. This routine uses several utility routines throughout the Monitor. To return to BASIC, use ctrl/C or 3D0G.

$FF70   **(65392, − 144)**

Entry point to command interpreter. The input buffer at $0200 must contain the command string. This point is used by Lam's method in entering Monitor commands from BASIC. See Section 3.1 for more information.

$FFFA.FFFB   **(65530, − 6)**   **NMI**

Hardware NMI vector. Apple uses $03FB to allow users to trap NMIs in Page Three.

$FFFC.FFFD   **(65532, − 4)**   **RESET**

Hardware RESET vector. Apple traps this to its own routines which vary considerably from model to model. See Section 3.4 for a complete description.

$FFFE.FFFF   **(65534, − 2)**   **IRQ**

Hardware IRQ/BRK vector. Apple traps this to its own routines which vary from model to model. See Section 3.4.

Table 2-5. HIRES1 — The Second K
(second lines of eight in each row)

Address	Row	Line	Y-coord
$2400.2427	0	1	$BE
$2428.244F	8	65	$7E
$2450.2477	16	129	$3E
$2478.247F		Unused	
$2480.24A7	1	9	$B6
$24A8.24CF	9	73	$76
$24D0.24F7	17	133	$36
$24F8.24FF		Unused	
$2500.2527	2	17	$AE
$2528.254F	10	81	$6E
$2550.2577	18	145	$2E
$2578.257F		Unused	
$2580.25A7	3	25	$A6
$25A8.25CF	11	89	$66
$25D0.25F7	19	153	$26
$25F8.25FF		Unused	
$2600.2627	4	33	$9E
$2628.264F	12	97	$5E
$2650.2677	20	161	$1E
$2678.267F		Unused	
$2680.26A7	5	41	$96
$26A8.26CF	13	105	$56
$26D0.26F7	21	169	$16
$26F8.26FF	.	Unused	
$2700.2727	6	49	$8E
$2728.274F	14	113	$4E
$2750.2777	22	177	$0E
$2778.277F		Unused	
$2780.27A7	7	57	$86
$27A8.27CF	15	121	$46
$27D0.27F7	23	185	$06
$27F8.27FF		Unused	

Table 2-6. HIRES1 — The Third K
(third lines of eight in each row)

Address	Row	Line	Y-coord
$2800.2827	0	2	$BD
$2828.284F	8	66	$7D
$2850.2877	16	130	$3D
$2878.287F		Unused	
$2880.28A7	1	10	$B5
$28A8.28CF	9	74	$75
$28D0.28F7	17	134	$35
$28F8.28FF		Unused	
$2900.2927	2	18	$AD
$2928.294F	10	82	$6D
$2950.2977	18	146	$2D
$2978.297F		Unused	
$2980.29A7	3	26	$A5
$29A8.29CF	11	90	$65
$29D0.29F7	19	154	$25
$29F8.29FF		Unused	
$2A00.2A27	4	34	$9D
$2A28.2A4F	12	98	$5D
$2A50.2A77	20	162	$1D
$2A78.2A7F		Unused	
$2A80.2AA7	5	42	$95
$2AA8.2ACF	13	106	$55
$2AD0.2AF7	21	170	$15
$2AF8.2AFF		Unused	
$2B00.2B27	6	50	$8D
$2B28.2B4F	14	114	$4D
$2B50.2B77	22	178	$0D
$2B78.2B7F		Unused	
$2B80.2BA7	7	58	$85
$2BA8.2BCF	15	122	$45
$2BD0.2BF7	23	186	$05
$2BF8.2BFF		Unused	

Table 2-7. HIRES1 — The Fourth K
(fourth lines of eight in each row)

Address	Row	Line	Y-coord
$2C00.2C27	0	3	$BC
$2C28.2C4F	8	67	$7C
$2C50.2C77	16	131	$3C
$2C78.2C7F		Unused	
$2C80.2CA7	1	11	$B4
$2CA8.2CCF	9	75	$74
$2CD0.2CF7	17	135	$34
$2CF8.2CFF		Unused	
$2D00.2D27	2	19	$AC
$2D28.2D4F	10	83	$6C
$2D50.2D77	18	147	$2C
$2D78.2D7F		Unused	
$2D80.2DA7	3	27	$A4
$2DA8.2DCF	11	91	$64
$2DD0.2DF7	19	155	$24
$2DF8.2DFF		Unused	
$2E00.2E27	4	35	$9C
$2E28.2E4F	12	99	$5C
$2E50.2E77	20	163	$1C
$2E78.2E7F		Unused	
$2E80.2EA7	5	43	$94
$2EA8.2ECF	13	107	$54
$2ED0.2EF7	21	171	$14
$2EF8.2EFF		Unused	
$2F00.2F27	6	51	$8C
$2F28.2F4F	14	115	$4C
$2F50.2F77	22	179	$0C
$2F78.2F7F		Unused	
$2F80.2FA7	7	59	$84
$2FA8.2FCF	15	123	$44
$2FD0.2FF7	23	187	$04
$2FF8.2FFF		Unused	

Table 2-8. HIRES1 — The Fifth K
(fifth lines of eight in each row)

Address	Row	Line	Y-coord
$3000.3027	0	4	$BB
$3028.304F	8	68	$7B
$3050.3077	16	132	$3B
$3078.307F		Unused	
$3080.30A7	1	12	$B3
$30A8.30CF	9	76	$73
$30D0.30F7	17	136	$33
$30F8.30FF		Unused	
$3100.3127	2	20	$AB
$3128.314F	10	84	$6B
$3150.3177	18	148	$2B
$3178.317F		Unused	
$3180.31A7	3	28	$A3
$31A8.31CF	11	92	$63
$31D0.31F7	19	156	$23
$31F8.31FF		Unused	
$3200.3227	4	36	$9B
$3228.324F	12	100	$5B
$3250.3277	20	164	$1B
$3278.327F		Unused	
$3280.32A7	5	44	$93
$32A8.32CF	13	108	$53
$32D0.32F7	21	172	$13
$32F8.32FF		Unused	
$3300.3327	6	52	$8B
$3328.334F	14	116	$4B
$3350.3377	22	180	$0B
$3378.337F		Unused	
$3380.33A7	7	60	$83
$33A8.33CF	15	124	$43
$33D0.33F7	23	188	$03
$33F8.33FF		Unused	

Table 2-9. HIRES1 — The Sixth K
(sixth lines of eight in each row)

Address	Row	Line	Y-coord
$3400.3427	0	5	$BA
$3428.344F	8	69	$7A
$3450.3477	16	133	$3A
$3478.347F		Unused	
$3480.34A7	1	13	$B2
$34A8.34CF	9	77	$72
$34D0.34F7	17	137	$32
$34F8.34FF		Unused	
$3500.3527	2	21	$AA
$3528.354F	10	85	$6A
$3550.3577	18	149	$2A
$3578.357F		Unused	
$3580.35A7	3	29	$A2
$35A8.35CF	11	93	$62
$35D0.35F7	19	157	$22
$35F8.35FF		Unused	
$3600.3627	4	37	$9A
$3628.364F	12	101	$5A
$3650.3677	20	165	$1A
$3678.367F		Unused	
$3680.36A7	5	45	$92
$36A8.36CF	13	109	$52
$36D0.36F7	21	173	$12
$36F8.36FF		Unused	
$3700.3727	6	53	$8A
$3728.374F	14	117	$4A
$3750.3777	22	181	$0A
$3778.377F		Unused	
$3780.37A7	7	61	$82
$37A8.37CF	15	125	$42
$37D0.37F7	23	189	$02
$37F8.37FF		Unused	

Table 2-10. HIRES1 — The Seventh K
(seventh lines of eight in each row)

Address	Row	Line	Y-coord
$3800.3827	0	6	$B9
$3828.384F	8	70	$79
$3850.3877	16	134	$39
$3878.387F		Unused	
$3880.38A7	1	14	$B1
$38A8.38CF	9	78	$71
$38D0.38F7	17	138	$31
$38F7.38FF		Unused	
$3900.3927	2	22	$A9
$3928.394F	10	86	$69
$3950.3977	18	150	$29
$3978.397F		Unused	
$3980.39A7	3	30	$A1
$39A8.39CF	11	94	$61
$39D0.39F7	19	158	$21
$39F8.39FF		Unused	
$3A00.3A27	4	38	$99
$3A28.3A4F	12	102	$59
$3A50.3A77	20	166	$19
$3A78.3A7F		Unused	
$3A80.3AA7	5	46	$91
$3AA8.3ACF	13	110	$51
$3AD0.3AF7	21	174	$11
$3AF8.3AFF		Unused	
$3B00.3B27	6	54	$89
$3B28.3B4F	14	118	$49
$3B50.3B77	22	182	$09
$3B78.3B7F		Unused	
$3B80.3BA7	7	62	$81
$3BA8.3BCF	15	126	$41
$3BD0.3BF7	23	190	$01
$3BF8.3BFF		Unused	

Table 2-11. HIRES1 — The Eighth K
(eighth lines of eight in each row)

Address	Row	Line	Y-coord
$3C00.3C27	0	7	$B8
$3C28.3C4F	8	71	$78
$3C50.3C77	16	135	$38
$3C78.3C7F		Unused	
$3C80.3CA7	1	15	$B0
$3CA8.3CCF	9	79	$70
$3CD0.3CF7	17	139	$30
$3CF8.3CFF		Unused	
$3D00.3D27	2	23	$A8
$3D28.3D4F	10	87	$68
$3D50.3D77	18	151	$28
$3D78.3D7F		Unused	
$3D80.3DA7	3	31	$A0
$3DA8.3DCF	11	95	$60
$3DD0.3DF7	19	159	$20
$3DF8.3DFF		Unused	
$3E00.3E27	4	39	$98
$3E28.3E4F	12	103	$58
$3E50.3E77	20	167	$18
$3E78.3E7F		Unused	
$3E80.3EA7	5	47	$90
$3EA8.3ECF	13	111	$51
$3ED0.3EF7	21	175	$10
$3EF8.3EFF		Unused	
$3F00.3F27	6	55	$88
$3F28.3F4F	14	119	$48
$3F50.3F77	22	183	$08
$3F78.3F7F		Unused	
$3F80.3FA7	7	63	$80
$3FA8.3FCF	15	127	$40
$3FD0.3FF7	23	191	$00
$3FF8.3FFF		Unused	

CHAPTER THREE

Machine Language

3.1 THE 6502 PROCESSOR

3.1.1 Architecture

To program the Apple directly in machine language requires a knowledge of the 6502 *processor* — its instructions and its architecture. Although this knowledge takes time not needed when you learned BASIC, it allows you to write better, faster, and simpler programs. By using an Assembler, you can create machine language routines that can be easily maintained through the use of structured programming techniques.

This chapter gives you machine language techniques. The remaining chapters apply these techniques throughout the Apple II.

Before getting into the processor, you should know about the heart of the Apple — its *clock*.

All computers run with clocks. The clock becomes an input to just about everything in the computer so that the complex signals can be kept in step — synchronized — with each other. The processor, memories, the memory management circuits, the I/O logic, the video generator — all the circuits in the computer need the clock to tell them exactly when to do something.

Simply, a clock is just a square-wave oscillator in the computer, usually crystal controlled. The 6502 processor used in the Apple has a clock circuit built-in, but it is not used. Instead, the Apple has an ex-

ternal clock using a circuit on the motherboard with a 14.318 megahertz crystal. The resulting signal is divided down to produce several clock frequencies needed throughout the Apple. By having a single oscillator, these frequencies are kept synchronized to each other. Fig. 3-1 shows the Apple II clock signals.
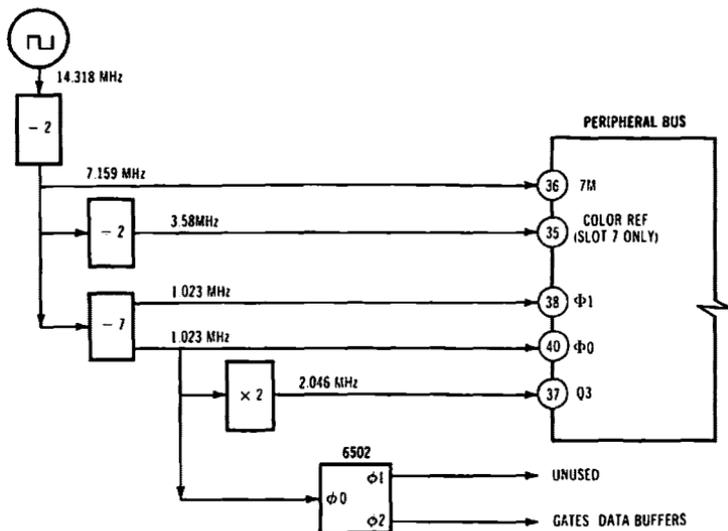
Fig. 3-1. Apple II clock signals.

The original 14.318 MHz is divided by two so as to make a perfectly symmetrical 7.159 MHz square wave. In the Apple, it is called 7M and appears at Pin 36 on the peripheral slots. For practical purposes, this is the master clock signal; the 14.318 MHz signal is too dirty and inaccessible.

The 7M clock is used by the video generator, both as a dot generator for character display and, when divided to 3.58 MHz, as the color subcarrier.

Most important to the processor is a divide-by-seven circuit that produces two square waves, in sync, at a frequency of 1.023 MHz. One is called *Phase Zero* and the other is called *Phase One*. Each phase is the complement of the other; that is, when one is high the other is low. All data transfer to and from the 6502 processor takes place during Phase Zero. The processor changes its address then during Phase One so that it has settled by the time Phase Zero comes again. This way, the processor reads and writes data at various addresses.

This scheme lets one oscillator produce all the clock signals needed. One drawback is the loss of the 6502's internal clock. The 6502 normally generates a Phase One and a Phase Two complementary pair of square waves when connected to a crystal of one megahertz or so. Unfortunately, Apples have a Phase Zero signal that is not exactly the same as the native Phase Two it replaces. So, before any time-critical peripherals can be used, you should try them out in your model. The earlier models deviated more from proper timing than does the IIe model. Boards designed for older Apples may not work on the IIe. Similarly, if you make your own cards, read Section 8.2 carefully.

One other signal is derived from the master clock for use in memory timing. Called Q3, it is multiplied from Phase Zero to be 2.046 MHz and it is not symmetric. It is important for video display timing. If needed by a peripheral, it is available on Pin 37.

The Apple clock uses a 14.318 MHz crystal to time its five lines. Fig. 3-2 shows the timing diagram. The master clock of 7.159 is called 7M and is divided to 3.58 MHz for COLOR REF. The 2.046 MHz line called Q3 is asymmetrical for display timing. At 1.023 MHz, Phase Zero times data transfer for the processor and Phase One times RAM refresh and I/O access. The processor uses Phase One to change its address.
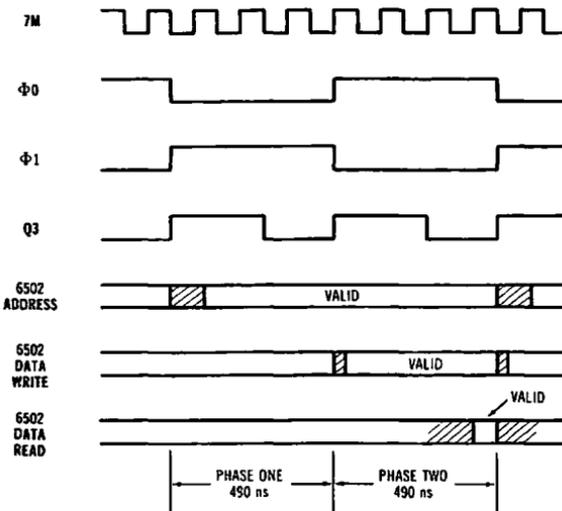
Fig. 3-2. Apple II clock timing diagram.

On the 6502 processor chip, there are five control pins, three of which are connected to the Apple. These three are called *interrupts*.

The interrupt lines let the Apple force the processor to execute routines located in memory at specific addresses. They are called IRQ for interrupt request, NMI for *non-maskable* interrupt, and RES for reset. All these lines are normally high. Bringing any one low causes the processor to stop whatever it is doing and get an interrupt address to use for further program execution.

The IRQ and NMI interrupt procedures in the pro enough to remember the addresses of the old routine tinued later. The RES interrupt is used at power up press, so it doesn't have to remember any previous r three interrupts will force the processor to execute only the IRQ and NMI routines can then recall the routine when they are finished. This feature lets the interrupt from a peripheral, do whatever the hard return from the interrupt by continuing the execution gram was running at the time.

The RES interrupt is the one the Apple uses to start is the most used interrupt. The IRQ may be used by isn't used by any of the built-in I/O. The NMI is use puters to aid in debugging, but is not used on the A available, however; it is just rarely used.

There are other control lines on the 6502. Called SYNC, they are used in debugging and for special co essor in special applications. Because they are not c original Apple, their use isn't covered in this book.

The remaining pins on the 6502 are the eight da seventeen address lines. They are connected to the N ment and I/O Logic on the Apple with a data bus ane

The 6502 works by generating sixteen-bit address Phase One, then either reading or writing eight bit Phase Two. A read/write control line is determined b Phase One to tell whatever is being addressed which d are to go. This R/W line on the 6502 is the seventeer

The first sixteen lines of address are called A0 to *A* gives the *lowest significant bit* of the address and A15 *significant bit*. With all sixteen lines, the process address from binary 0000000000000000 to binary 11 Using hex notation, these are $0000 to $FFFF. The

gives the direction at the same time in Phase One. The bits are encoded by *positive logic* in which a high level represents a binary one and a ground level represents a binary zero. The R/W line goes high for a processor read and low for a processor write.

The data lines on the 6502 are similar to the address lines. They have positive logic: a ground for zero, a high TTL level for one. Like the address lines, data lines are connected to the Apple by a bus.

The data lines are, however, quite different in their use. Addresses are generated during Phase One and only by the processor. Data are generated during Phase Two and may be in either direction. During a read, data enters the 6502 from the data bus; during a write it leaves the 6502. So, the address bus in the Apple is *unidirectional* and the data bus is *bidirectional*. On top of all that, the data bus is only eight bits and the address bus is seventeen bits.

On the Apple II, Phase Zero is used as being (almost) the same as Phase Two.

Hardware connected to the busses must *gate* or enable data transfers during Phase Two. And, the hardware must be selected from the address bus so that it transfers only after it has been addressed during Phase One. If you look at the schematics of peripheral cards, you will find Phase One used to enable data transfers. This is because Phase One is low during Phase Two, and many devices are enabled by a low level. Regardless of the details, all devices have address decoding and data transfer enabling during Phase Two.

This complements the 6502 that generates the addresses during Phase One. When they have settled and Phase Two comes along, data transfer is made with the addressed device. This is how the 6502 works with the Apple II: generating addresses, transferring data between the addressable devices in the system.

Inside the 6502, circuits accept interrupts, generate addresses, and transfer data. The 6502 can be instructed to do this in many ways. It also has internal storage registers to manipulate both addresses and data. See Fig. 3-3 for a block diagram of the 6502's insides.

In addition to the pinouts, the 6502 has internal data and address busses. These allow transfers among the various registers and the *Arithmetic-Logic Unit* (the ALU). This ALU is the workhorse of the processor: changing register values by arithmetic and logical operations like addition, AND-ing, decrementing values, and so on. When you program the 6502, you can change many register contents and then manipulate them with ALU calculations.
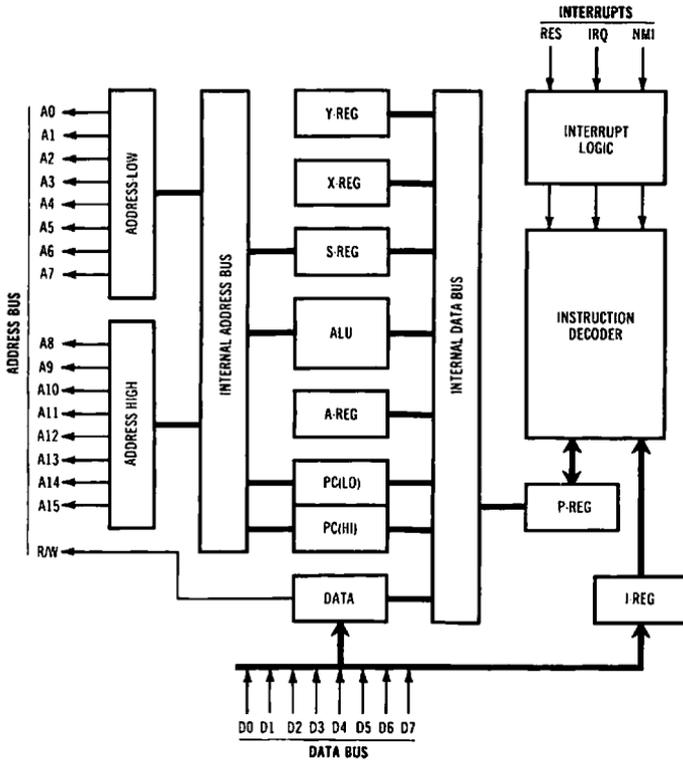
Fig. 3-3. Block diagram of 6502.

While the ALU and registers do the work, it is another chunk of logic that runs the processor. This *boss* is the *Instruction Decoder*. It uses an *Instruction Register* to store a special data byte called an *op code* or instruction. This op code in the IR tells the decoder exactly what to do. If an interrupt occurs on one of the three interrupt lines — IRQ, NMI, RES — then the decoder is forced to service that interrupt. In any case, all the decisions about what to do are made in the 6502 by the Instruction Decoder.

The whole works is tied together by buffers, latches, and control lines. Of these, three buffers — address-low, address-high, and data — are used between the external and internal busses. Each buffer and register is eight bits in size. Buffers isolate the internal 6502 from the outside world, so it can work by itself during Phase One as well as when transferring data during Phase Two. The I-reg is the instruction

register and selectively reads the data bus, while the data buffer can both read and write. The address buffers write only to the address bus.

Like the buffers, the registers are eight bits each that contain one byte at a time. The two PC registers hold sixteen bit addresses, so you can think of PC as one long register called the *Program Counter*. The decoder uses the PC to keep the address of its next instruction. The current instruction is in the I-reg. And, the decoder keeps track of its status in the P-reg, the Processor Status. Within the P-reg are eight bits called flags that turn various features of the processor on and off. The decoder sets and tests these flags during instruction execution. And, the decoder can invoke the remaining registers and ALU by transferring data among them. Knowing the contents of these registers at any time is the key to following any machine-language program.

When programming, you only concern yourself with the six registers — A, Y, X, S, PC, and P. These are shown in a programming model in Fig. 3-4; they are the ones you work with in your programming.
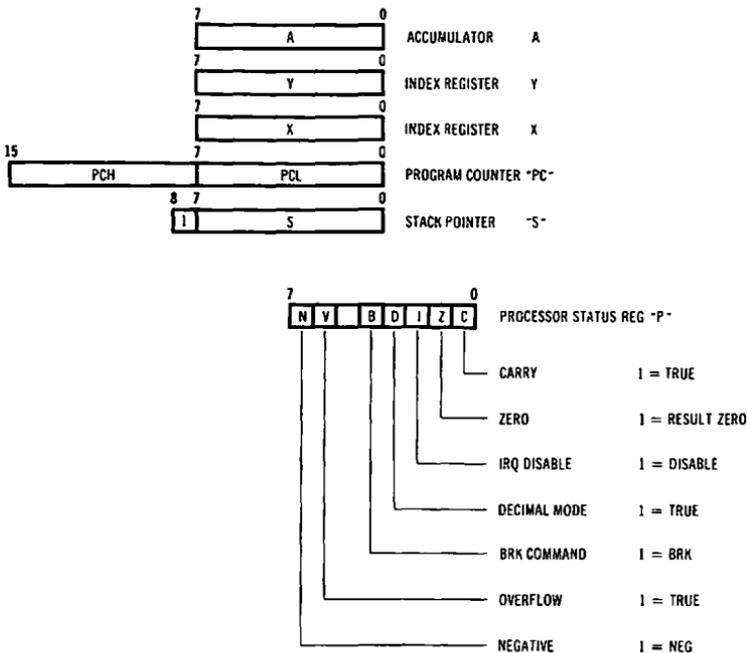
Fig. 3-4. Programming model.

The A-register is often called the *Accumulator* because of its heavy usage in arithmetic operations. This is the register commonly used to transfer bytes and modify them with the ALU to add, subtract, AND, OR, and Exclusive OR.

The *Y-register* and *X-register* are used often for address manipulation. They can be incremented or decremented easily in steps of one using special, fast, ALU instructions.

The *S-register* is a special kind of address register that remembers where the 6502 keeps its own information, in a special RAM area of memory. When used, the 6502 always puts $01 out as the address-high whenever it puts out the contents of S-reg as address-low. This way, the S-reg acts as a pointer to the $0100.01FF chunk of RAM. Another name for the S-reg is the *stack pointer.*

The PC is sixteen bits, so it can point anywhere in memory. The decoder puts its contents on the address bus from the buffers whenever it wants to *fetch* another instruction. It can point anywhere in the address space: from $0000 to $FFFF.

The *P-register* or *Processor Status* contains eight bytes, seven of which act as flags. A flag modifies the action of one or more instructions. Each flag is summarized in Fig. 3-4 and described more fully in Table 3-1. You can ignore the details of the P-reg on first reading; they are for later reference and study.

What is of importance to the understanding of the processor at this stage is exactly how the processor functions in fetching and executing a sequence of instructions.

Table 3-1. Processor Status Flags

Bit	Flag	Set = 1	Clear = 0
0	C	Last ALU instruction had a carry result.	Last ALU instruction had a no-carry result.
1	Z	Last result was zero.	Last result was nonzero.
2	I	IRQ interrupts are disabled.	IRA interrupts are enabled.
3	D	Arithmetic of A-reg set to perform in BCD.	Arithmetic of A-reg set to perform in binary.
4	B	Last IRQ caused by BRK.	Last IRQ caused by hardware.
5	-	Unused	Unused
6	V	Arithmetic overflow from bit 6 of A-reg. Also, see BIT.	Arithmetic no-overflow from bit 6 of A-reg. Also, by BIT instruction.
7	N	Last result set a bit 7.	Last result cleared a bit 7.

Here's how the 6502 does an instruction. The instruction begins with the address of the next instruction in the program counter. This is either the result of the previous cycle, or it was forced there by the interrupt logic.

Regardless of its origin, the decoder puts the contents of the PC onto the internal address bus. At the proper time during Phase One of the clock, the high and low address buffers are loaded from the internal address bus, thereby putting the PC contents onto the external address bus. Simultaneously, the R/W line is brought high to signify a read request.

The addressed memory puts the contents of the requested location on the data bus during Phase Two of the same clock cycle. The decoder grabs the contents of the data bus in the I-reg. At the end of the first clock cycle, the 6502 has requested and read a byte from memory into the I-reg. By this action, it has fetched its instruction op code.

On the next clock cycle, the decoder executes the new instruction. According to the value in the I-reg, the decoder will perform a sequence of tasks, taking up to six clock cycles to complete. It may read, modify, and write registers. If the op code directs it to just modify a register, it will finish the task in one cycle. Reads and writes each take longer, while a read/modify/write instruction takes the longest time to complete.

Implied in the execution of all instructions is the change to the PC. It is always modified to point to the following instruction op code in memory, either by incrementing it or changing it altogether with a new value. In any case, the instruction ends with the address of the next instruction in the PC.

Take an example. Suppose the op code that was fetched was $AD. This tells the decoder that it is a three-byte-long instruction that takes four clock cycles to complete. On execution, it loads the A-reg from memory. To do this, the decoder first reads the two bytes following the op code in memory and uses these two bytes together as the address of the desired byte. So after execution, the PC will point to the third byte following its initial value.

Continuing the example, the decoder increments the address buffers by one after it has the op code. This lets it fetch the low byte of the desired address during Phase Two of the second clock cycle. Then the decoder fetches the high byte during the third cycle. With both bytes in hand, the decoder puts them on the address line and fetches the byte it finally wants into the A-reg during the fourth clock cycle. With the

A-reg finally replaced by the read value, and with the PC pointing to the next byte in the instruction sequence, the instruction cycle is completed.

3.1.2 Memory Mapping

The 6502 processor has an address space of 65536. This figure is usually referred to as 64K and is the total number of address values the processor can generate. This is the number of all possible combinations of low and high levels on the sixteen lines of the address bus.

Look at a few values to see how this works. The lowest address is $0000, in hex notation, and is generated when the processor brings all address lines low. If the A0 line went high while all remaining lines were held low, an address of $0001 would be generated. If only A1 were high, $0002 would be the address value. Similarly, only A2 high generates $0004, only A3 high generates $0008, only A4 high generates $0010, and so forth. Only A15 high generates a $8000. After these powers of two, combinations of lines generate other values: A0 and A1 generate $0003, for example, if they are the only lines held high. All lines held high give the greatest address possible — $FFFF.

For each of the 65536 different ways of setting the address bus, there is one and only one address value generated in the processor's address space. This one-to-one mapping is called the memory map of the processor. It shows exactly what memory locations, hardware, soft switches, and other system features correspond to the addresses. By reading the memory map of the Apple, you can get a picture of where things are and decide how best to use available memory in your programming.

It is easier to follow memory maps if you break down the 65536 addresses into 256 *pages* of 256 addresses each.

The sixteen address lines connect to two buffers in the processor. Each of these buffers is eight bits. One buffer connects to lines A0 through A7 and the other buffer connects to lines A8 through A15. The buffer with the higher lines holds the page number and the one with the lower lines holds the address within that page.

A couple of examples. In hex, address $0023 is in Page Zero ($00) with a page address of $23. Address $2040 is in Page $20 with page address of $40. Address $FBDB is in Page $FB at page address $DB. With hex notation, it's easy.

With a given memory map, the system may or may not have memory at any given address. An Apple has hardware and possibly ROM in the $C000.CFFF range, for instance. That in fact is one of the first things a memory map should tell you: where is RAM? where is ROM? where is hardware? Then more detailed maps can break down the address space further.

In the case of memory, each address within its range tells the memory chip which *location* within itself to access. Normally, each address has one and only one memory location holding eight bits of data. When addressed, the memory chip then transfers to or from the given location by accessing the data bus.

In the case of hardware, the Apple addresses are often duplicated in a device. The device can have one of several addresses simply because the least significant lines, from A0, aren't connected. Some features added to the IIe model use these otherwise disconnected lines, so some programs may have trouble running on the IIe for this reason. The point to be made here is that addresses aren't always decoded from the address bus on a one-to-one basis.

The memory map used in this book is that of Fig. 3-5 unless otherwise noted. The so-called soft switches can alter the memory map when you want to change it in your programs. Most of the information on how to do that is given in Chapter Two.

Regardless of which memory map is in effect, the 6502 demands that RAM and ROM be provided at specific addresses. The Apple maps always provide RAM for processor work space and ROM for interrupt routines.

The address of the processor consists of 256 pages, from $00 to $FF. Of these, Page Zero and Page One must always be RAM. Page Zero is used for address pointers and fast instructions. Page One is used to remember processor registers.

At the other end of the address space, Page $FF must have a ROM. The highest six locations must contain the three address pointers to the three interrupt routines for RES, MNI, and IRQ. When one of these interrupts occurs, the address is loaded into the PC from two of these locations, forcing the processor to execute the interrupt routine at that address. The Apple always has one ROM chip active at the top of memory — $F800.FFFF — for this reason. This ROM contains the three pointers or *interrupt vectors* together with their routines. Without such a ROM, the Apple just could not start itself.
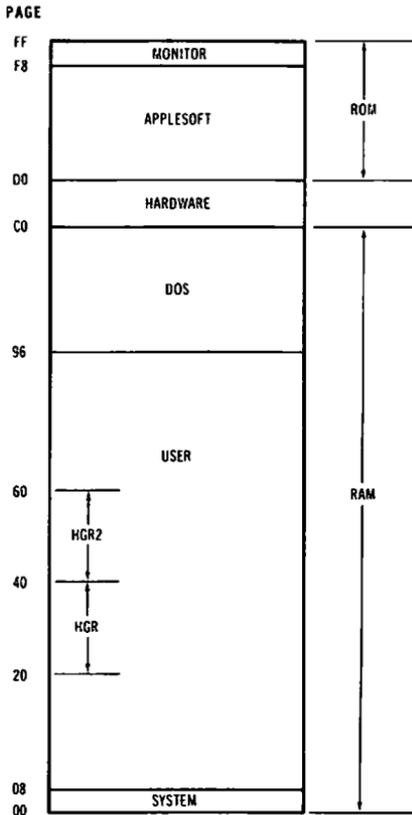
With RAM always at low memory and ROM always at high

PAGE

Fig. 3-5. Apple II plus memory map.

memory, a 6502 computer ends up with its hardware decoded some-where in the middle of memory. For the Apple II, this is done in the $C000.CFFF range.

You can see this division of memory in Fig. 3-5. If you don't learn any other memory map of the Apple II, you should learn this one. This is the most common map, and the one normally created when the Apple II Plus or the Apple IIe is started with a DOS. The earlier Apple II standard model had Integer BASIC resident in the space now occupied by Applesoft; otherwise, it too has the same map.

The RAM range of $0000.BFFF occupies 48K. If you have an old Apple II with only 32K of memory, it resides in the $0000.7FFF range, the $8000.BFFF addresses being unoccupied. An Apple with 16K of RAM will have RAM in the $0000.3FFF range, with $4000.BFFF un-

occupied. Unless stated otherwise, this book assumes a 48K Apple with a BASIC language, usually Applesoft, installed in the ROM area.

The ROM area from $D000 to $FFFF contains the language Applesoft at $D000.F7FF and a Monitor at $F800.FFFF. The Monitor may be one of three versions: Standard, Autostart, or IIe. If Integer BASIC is resident instead of Applesoft, the ROMs for $E000.F7FF will be installed, perhaps with a $D800.DFFF ROM as well. A socket for a $D000.D7FF is not used by Integer BASIC. If the $D8 socket is empty, you can get the ROM, called *Programmer's Aid #1,* from an Apple dealer; it comes with a manual.

The input/output area at $C000.CFFF is divided into two parts. Built-in I/O consisting of keyboard, speaker, soft switches, cassette, and games socket lives in the $C000.C07F range. Peripheral I/O can use the remaining space, over the $C800.CFFF range. Each slot is allocated chunks of memory for its own use; see Chapter Two for the details.

Knowing the basic memory map of Fig. 3-5 is essential to getting around in the Apple with machine language programs.

3.1.3 Instructions

To execute a machine language program that exists somewhere in memory, you must somehow get the address of its first instruction into the 6502's PC. One way to do this is with the RES interrupt, which is how the Apple gets started in the first place. Once started, there are other ways to change the contents of the PC; they will be covered later. For now, assume you have a machine language routine to execute and assume that its address is in the PC.

Here's how the program execution works. First, it takes one clock cycle for the processor to fetch the first byte into the I-reg. This byte, pointed to by the PC, must be an op code value because the processor will attempt to decode it as soon as it is fetched to the I-reg.

The entire machine language instruction may be one, two, or three bytes in size. If only one byte, then it consists entirely of an op code. On the other hand, if more than one byte, it has what is called an operand as well as an op code. The first byte is always the op code. The operand byte or bytes that follow may be data for one of the registers or they may be an address. The processor will read these operand bytes whenever the op code tells it to. Each op code identifies the instruction to the processor by telling it exactly what to do. Often,

it tells the processor to fetch certain operand bytes as part of the instruction.

Regardless of the length, the instruction is completed with the PC pointing to the next instruction. Usually, this is the one immediately following in memory. Other times, the op code tells the processor to modify the PC in another manner, like putting the operand bytes into the PC to make a jump instruction.

For all instructions, the cycle is the same. One clock cycle fetches the op code byte and more clock cycles may be needed to execute the op code. There may or may not be operand bytes.

You write machine language programs, then, by putting the sequence of bytes into memory that make up the instructions you want. Each instruction has an op code. Depending on the op code, you complete each instruction with any operands that it requires.

Some instructions that have only the op code byte are:

$CA   decrement X-reg value by one
$C8   increment Y-reg value by one
$18   clear C-flag in P-reg to zero
$F8   set D-flag in P-reg to one
$EA   no-operation, two clock cycles long

By putting a sequence of these bytes into memory somewhere, you would be *loading* a program. Each byte would be a complete instruction, consisting of an op code that requires no operand.

Such one-byte instructions are limited in what they can do for you. You want to operate on addresses and data, not just registers. For this, instructions with operands are needed. These can be varied for different addressing methods, so you will learn them more slowly than one-byte instructions.

First, you can start with a few longer instructions that you could use in writing short, simple programs.

A two-byte instruction you can use is the *load immediate to A-reg*. The op code is $A9 and you follow it immediately with a one-byte operand. The value of the operand will be put into the A-reg by the processor when it executes the instruction. If you put

$A9
$FF

into two consecutive memory locations and the address of the first byte into the PC, then the processor will execute it. The result would be to make the contents of the A-reg $FF in value and increment the PC by two.

If you used a $80 as the operand instead of $FF, then the processor would put the $80 into the A-reg.

Next, here is a three-byte instruction, a jump. It has $4C as its op code and two bytes as its operand. The operand is the address of the next instruction — a forced address. When it executes, the processor puts the two operand bytes into the PC. This results in the program *jumping* to the address given by the operand. Machine language programs use $4C like BASIC uses the "GOTO" command. For example, if the 6502 executes

$$\$4C$$
$$\$00$$
$$\$03$$

it will replace the address in the PC with $0300. This results in the next instruction's op code being fetched from $0300 instead of the following location. Instead of $0300, you can jump anywhere you like. Just put the address — low byte followed by high byte — following the $4C as its operand.

So, to program in machine language, you enter sequences of instructions to memory. Each instruction must have an op code and be followed by as many operand bytes as required. The operand depends on its op code for its length and on you for its value.

Here is an example of a machine language routine.

To call, you must load the registers with any values that the routine requires, then jump to the routine. Specifically, the routine at $FDED wants the A-reg set to the character code to be output.

Location	Content
$0300	$A9
$0301	$2C
$0302	$4C
$0303	$ED
$0304	$FD

A more compact notation would be

$0300: A9 2C 4C ED FD

like the way it would be entered to the Monitor.

To write machine code like this means you have to memorize all the op codes you may need. Then you have to look up any character codes and addresses you need for operands. This is difficult and slow. The task of reading the resulting machine code in hex is even worse.

Instead, you can use an *assembler notation* instead of pure hex when you write machine programs. In assembler notation, the routine just given looks like

$0300: LDA #$2C
JMP $FDED

instead.

Simple assembler notation like this uses two tricks to make reading and writing easier. First, the code is arranged in three columns: address, op code, and operand. Second, the op code is written as a *mnemonic* instead of its hex value.

The mnemonics are easy to remember. They replace the hex op codes as you write routines. After the routine has been written, you can easily look up the mnemonics to get the op codes they represent.

One mnemonic can represent several op codes. For instance, there are eight different ways to load the A-reg. The "#" in the example signifies the immediate way; there are others. However, the exact op code can always be found from the mnemonic and the context. Having fewer mnemonics than op codes makes memorizing them even easier.

As you learn machine programming, use the Monitor's L command. You can disassemble the routines listed in Section 2.2. See the hex notation on the left; the assembler notation on the right. Compare them, and see if you can match the op codes and mnemonics as you learn them. Do the op codes always match the mnemonics in their proper context? If so, it means that you recognized the instruction correctly.

3.1.4 A Routine to Modify Memory

This is a routine that uses two instructions having LDA and STA as mnemonics. There are eight different op codes for each one, so these

instructions are more closely described as the load A-reg absolute and the store A-reg absolute.

The LDA absolute has $AD as its op code. When executed, it loads the A-reg with the contents of the memory location whose address follows the op code. For example, the code

$0300: AD 34 12

tells the processor to load the A-reg with the contents of memory at location $1234. The address $1234 is the operand of the LDA absolute instruction.

Similarly, a STA absolute instruction has an op code of $8D. Executing that one causes the contents of the A-reg to be stored in memory at the address given by the operand. So,

$0303: 8D 35 12

tells the processor to store the A-reg at $1235. Just like the LDA absolute, the STA absolute has the address of the memory as its operand.

Here is an example of a short program to move the contents of one memory location to another:

```
$0300:  AD 20 10  LDA  $1020
$0303:  8D 30 10  STA  $1030
$0306:  4C 69 FF  JMP  $FF69
```

See the hex code on the left and the assembler notation on the right. This is how it might appear when disassembled by the Monitor L command.

The first instruction loads the contents of $1020 into the A-reg. The second stores the contents to $1030. The third instruction jumps to the Monitor's warm start point. The result is that a copy of the contents of $1020 exists in $1030. The copy in the A-reg of the two instructions are still there when it jumped to the Monitor.

Use the same routine, but write it to move the contents of $F800 to $1000.

Look at the code you wrote. Assume that it is executing and suppose that $0300 is in the PC at the beginning of the first instruction cycle. Follow the code exactly like the processor does.

With a PC of $0300, the 6502 loads the contents of $0300 into the I-reg, a value of $AD. Then, the instruction decoder recognizes it as a LDA absolute instruction. The next two bytes, $00 and $F8, are fetched from $0301 and $0302. These two bytes are put on the address bus and a read takes place. The byte read from $F800 is $4A, and it is put into the A-reg. At the end of the first instruction, the PC has been incremented to $0303 and the A-reg has $4A.

This marks the beginning of the second instruction. The PC points to $0303, so its contents are loaded into the I-reg to become the new instruction op code — $8D. The instruction decoder sees the $8D as the STA absolute, so it fetches the next two bytes for an address to use. These bytes in $0304 and $0305 are $00 and $10, respectively. Using them, the decoder addresses $1000 and writes the contents of the A-reg. Because the A-reg contains $4A and the address $1000 is a RAM location, this results in $1000 having its contents changed to $4A. The PC is advanced automatically to $0306.

This sets the processor for the third instruction and it fetches the contents of $0306 as the new op code. This is $4C, and the decoder recognizes it as the jump instruction. The next two bytes are therefore fetched and stuffed into the PC itself. The result is a $FF69 in the PC; the program has jumped to the Monitor routine's instruction at that point.

If you ran this routine and could examine the contents of $1000 when finished, you should find it containing $4A, which is the same value as $F800.

By *walking down* a program like this and creating the processor's scenario for yourself, you can learn to read any program listing. It is a sure way to debug difficult routines and predict their results.

3.1.5 Hack and Run

To create your own machine programs, you must do four things: *code, assemble, load,* and *test.* Once the program tests good, you can BSAVE it to disk for future use. Of the several methods of performing these four steps, the simplest and easiest one for short routines is the *hack and run* method given here.

To code a machine program, use quad paper or a special coding form like that of Fig. 3-6. This form has four columns on the right for assembler notation: LABEL, MNEMONIC, OPERAND, and COM-

6502 Program Code						Date _____	Page _____ of _____	
PROGRAMMER: _____						PROGRAM: _____		
ADDR	B1	B2	B3	LABEL	MNEMONIC	OPERAND	COMMENTS	

Fig. 3-6. Programming form.

MENTS. The leftmost columns — ADDR, B1, B2, and B3 — are ignored during coding and are left blank.

First of all, because sheets of paper with machine code on them all look the same, you will want to identify them properly. Do this at the top of the form on each sheet you code.

Begin writing any routine with a short comment that says what the program does. Give the exact call sequence telling the reader how to *invoke* the routine. Any routine that has either the purpose or call sequence unknown is useless in future.

Each instruction goes on a separate line, starting on the first line. The mnemonic is the verb of the instruction — it tells the processor what to do. If it is a 6502 mnemonic it will be translated into an op code during the assembly step. Otherwise, you can write another kind of mnemonic called an *assembler directive.*

An assembler directive, sometimes called a *pseudo-op,* is an instruction to the assembler, not to the processor. For instance, ORG tells the assembler where to start the program, locating it in memory. Another one, DS, defines a storage area of a given number of bytes. The ASC directive identifies characters to be translated into their ASCII codes at assembly time. And so on. The directives used in this book are

summarized in Table 3-2. Directives in other books and listings may vary from these.

Table 3-2. Commonly Used Assembler Directives

Syntax	Purpose	Example
ASC string	Create string	MESSAG ASC 'HELLO'
DS expression	Define storage area	BUFFER DS $16 + SIZE
DW expression	Define word, address form	DW $1234
DFB expression	Define byte value	DFB 22
EQU address	Declare a label	COUT EQU $FDED
ORG address	Locate start of assembly	ORG $0300

Instructions having operands can express them in various ways. If the numeric value of the operand is known, you can simply write it there. You can express it in a hex form like the examples so far. Or, you could use the decimal or even the binary form of the number. If you don't know which number you want, you can substitute a label for it. The expression you choose can be translated into a hex number when you assemble the code later. Some label and number references that might appear as operands in an assembler program are:

$FF	byte, expressed in hex
255	byte, expressed in decimal
%11111111	byte, expressed in binary
MAXVAL	labeled reference

If a "#" appears in front of the operand, remember that it means *immediate* and it is there to provide meaning for the mnemonic. Sometimes, characters appear like

LDA #'A'

where the op code will be LDA immediate — $A9 — and the operand will be the ASCII code for 'A' which is $41.

The advantage of the assembler coding is that you don't have to know the op codes or the operand hex values when you first write your program. You can look them up later.

Look at the example of Fig. 3-7. The right side of the coding form (comments column) has the assembler program. The ORG directive tells the assembler where the program will finally load into memory.

ADDR	B1	B2	B3	LABEL	MNEMONIC	OPERAND	COMMENTS
							Demonstration of Assembly
FF69				MONZ	EQU	$FF69	*monitor warm start*
F800				HERE	EQU	$F800	*source address*
1000				THERE	EQU	$1000	*destination address*
					ORG	$0300	
0300	AD	00	F8	MOVE	LDA	HERE	*copy contents of memory*
0303	8D	00	10		STA	THERE	*from HERE to THERE*
0306	4C	69	FF		JMP	MONZ	*and then exit to Monitor*

6502 Program Code — Date _____ Page _1_ of _1_
PROGRAMMER: P. IRWIN   PROGRAM: FIGURE 3-7

Fig. 3-7. Coding form example.

Three labels are listed and used as operands, so they are declared in the EQUate statements at the beginning. Two of these labels were invented — HERE and THERE. The other label, MONZ, is a given label; it was looked up in Section 2.2 at $FF69. The routine itself is labeled as MOVE, although it is not referenced here. Make your labels meaningful. If you need more labels within a routine, add numbers to the first label. For instance, if MOVE had to be labeled at other instructions than the first, they would be called MOVE1, MOVE2, MOVE3, etc. Look at the examples in this book.

Table 3-3 summarizes the rules for coding assembler. The rules emphasize writing simple and clear routines.

After coding the routine, it can be assembled. You can assemble in several ways, but the *hand-assembly* method is the first one you should use. Hand assembly doesn't depend on having any utilities like the Miniassembler or a commercial text assembly package. It develops understanding in a way no other method can. After hand assembling routines successfully, you will be able to test and debug your routines from disassemblies and dumps.

Here's how to hand assemble a routine successfully. On the coding sheet containing the routine in assembler form, write the address of the first location of the program. You get the address from the ORG

Table 3-3. Rules for Coding Assembler

Rule	Procedure
1.	Identify all coding sheets.
2.	Comment with purpose and call sequence. The routine should do only one task and be called only one way.
3.	List all external labels your routine references by using EQUate directives. Leave room for those you miss.
4.	Use an ORG directive to tell the assembler where the routine will begin.
5.	Code the routine itself. Use operand labels and comments to show *what* the routine does.
6.	The exit point, usually a jump (JMP) or a return (RTS), should be the last executable instruction of the routine.
7.	Label the entry point at the first location and assign any further labels the same name with a number appended.
8.	Use DS, DW, and ASC directives for storage and literal values at the end of the routine when required.

directive and put it in the ADDR column on the line of the first executable instruction. See the $0300 in Fig. 3-7 for an example.

Next, look up the op codes. For each mnemonic, you can infer the addressing mode when necessary. The op codes are given in Tables 3-4 and 3-5. Enter the op codes in the B1 column opposite its mnemonic, on the same line.

Put the addresses in the ADDR column. To get each address after the first, add the length of each instruction to its address. The resulting sum is the address of the following instruction. In the case of Fig. 3-7, all the lengths are three; so, the addresses are $0300, $0303, and $0306. Each instruction then is located in memory for further assembly and debugging.

In the case of directives, the DS, DW, and ASC all declare storage space. You must add their lengths to their addresses to get the next address in each case. The length is given as the operand of the DS. The DW is always two bytes in size. For ASC, count the characters between the quotes. Otherwise, treat these directives like op code mnemonics.

Once all addresses have been found you can go through the code, one line at a time, and get the hex values for all operands. They go into columns B2 and B3. You will need tables to convert decimal and binary to hex notation. And, you need character conversion tables to

Table 3-4. Unique 6502 Instructions

Mnemonic	Op code	Addressing	Flags
Branch			
BCC	90	Relative	------
BCS	B0	Relative	------
BEQ	F0	Relative	------
BMI	30	Relative	------
BNE	D0	Relative	------
BPL	10	Relative	------
BVC	50	Relative	------
BVS	70	Relative	------
P-register bit			
CLC	18	Implied	-----C
CLD	D8	Implied	--D---
CLI	58	Implied	---I--
CLV	B8	Implied	-V----
SEC	38	Implied	-----C
SED	F8	Implied	--D---
SEI	78	Implied	---I--
Program flow			
BRK	00	Implied	---I--
JMP	4C	Absolute	------
JMP	6C	Indirect	------
JSR	20	Absolute	------
NOP	EA		------
RTI	40	Implied	Stack*
RTS	60	Implied	------
Transfer			
TAX	AA	Implied	N---Z-
TAY	A8	Implied	N---Z-
TSX	BA	Implied	N---Z-
TXA	8A	Implied	N---Z-
TXS	9A	Implied	------
TYA	98	Implied	N---Z-
Stack			
PHA	48	Implied	------
PHP	08	Implied	------
PLA	68	Implied	N---Z-
PLP	28	Implied	Stack*

*Restored from stack

Table 3-5. Accumulator, Memory, and Index Instructions

	ACCUMULATOR/IMPLIED*	IMMEDIATE	ZERO PAGE	ZERO PAGE, X	ABSOLUTE	ABSOLUTE, X	ABSOLUTE, Y	INDIRECT, X	INDIRECT, Y	Flags
ADC	–	69	65	75	6D	7D	79	61	71	NVZC
AND	–	29	25	35	2D	3D	39	21	31	N-Z-
ASL	0A	–	06	16	0E	–	–	–	–	N-ZC
BIT	–	–	24	–	2C	–	–	–	–	76Z-
CMP	–	C9	C5	D5	CD	DD	D9	C1	D1	N-ZC
CPX	–	E0	E4	–	EC	–	–	–	–	N-ZC
CPY	–	C0	C4	–	CC	–	–	–	–	N-ZC
DEC	–	–	C6	D6	CE	DE	–	–	–	N-Z-
DEX	CA*	–	–	–	–	–	–	–	–	N-Z-
DEY	88*	–	–	–	–	–	–	–	–	N-Z-
EOR	–	49	45	55	4D	5D	59	41	51	N-Z-
INC	–	–	E6	F6	EE	FE	–	–	–	N-Z-
INX	E8*	–	–	–	–	–	–	–	–	N-Z-
INY	C8*	–	–	–	–	–	–	–	–	N-Z-
LDA	–	A9	A5	B5	AD	BD	B9	A1	B1	N-Z-
LDX	–	A2	A6	B6#	AE	–	BE	–	–	N-Z-
LDY	–	A0	A4	B4	AC	BC	–	–	–	N-Z-
LSR	4A	–	46	56	4E	5E	–	–	–	N-Z-
ORA	–	09	05	15	0D	1D	19	01	11	N-Z-
ROL	2A	–	26	36	2E	3E	–	–	–	N-ZC
ROR	6A	–	66	76	6E	7E	–	–	–	N-ZC
SBC	–	E9	E5	F5	ED	FD	F9	E1	F1	NVZC
STA	–	–	85	95	8D	9D	99	81	91	----
STX	–	–	86	96#	8E	–	–	–	–	----
STY	–	–	84	94	8C	–	–	–	–	----

* implied
# zero page. Y

N negative
V overflow
Z zero

C carry
6 V if bit 6
7 N if bit 7

look up characters in ASC directive strings. The best tool for this is the Reference Summary card supplied with this book. Labels as operands should all be found from the LABEL column. If not, something is missing from the assembler coding, usually an EQUate directive.

When finished assembling, your coding sheet should resemble Fig. 3-7.

To recapitulate, assembly is done in two passes through the code. On the first pass, addresses are found for all the op codes and assembler directives. The op codes are identified. Then, on the second pass, the operands are resolved by evaluating their expressions and looking up the addresses that correspond to the labels.

Once assembled on paper, you can enter it. Use a CALL − 151 to enter the Monitor from BASIC.

Enter the hex code using the address of the routine. For example, the routine of Fig. 3-7 can be entered as

300:AD 00 F8 8D 00 10 4C 69 FF

Verify the code by disassembling it with the L command:

300L

The code should disassemble properly. If there is an error, correct it before proceeding.

If you have a long routine you don't want to re-enter, then BSAVE it to disk before testing.

Now, use the call sequence to give your routine any initial conditions it wants. Then run it with the G command:

300G

The routine should return to the Monitor when finished, giving you an asterisk — "*" — followed by a cursor. If not, you have problems. You have to *walk through* your program as described before. If you can get and use a *Step/Trace utility,* so much the better.

If your routine returns normally, you still can't assume it did its job. What was it supposed to do? You must have a specific purpose for the routine that can be tested. In the case of Fig. 3-7, for instance, you can examine the contents of locations $F800 and $1000 to see if it moved from one location to the other. If so, they will both have the same value. It would be a good idea to force the destination, $1000, to contain another value before the routine is run.

Then, what if it doesn't work properly? Record any results, however false. Then with these results in hand go through the code. Why does

the routine produce the observed result? Once you know that, you can usually find a way to change or rewrite the routine to get the result you want. Most errors are caused by mechanically copying or looking up an incorrect value. The remainder are caused by a misconception on the programmer's part.

Always retest a routine completely after making any changes, no matter how slight those changes are.

With practice the four steps — coding, assembling, entering, and testing — can be done quickly for short routines. In fact, experienced programmers can hand assemble quicker than using a disk-based assembler package when they want. With many op codes memorized, the programmer can often enter in hex without going through the assembler stage. This facility with the hack and run method earns such programmers the title — hacker.

One way of approaching the skill of the hacker without invoking a large, disk-based assembler is with a utility available with Integer BASIC called the Miniassembler. It uses the Monitor's disassembler so it has much the same format. It is easy to learn and use.

To use the Miniassembler, activate Integer BASIC; use the INT command to DOS. Then CALL − 151 to enter the Monitor. Invoke the Miniassembler with

<div align="center">**F666G**</div>

It responds with a "!" prompt. Whenever you want to leave and return to the Monitor, type

<div align="center">**$FF69G**</div>

to the Miniassembler. In fact, you can give any Monitor command from the Miniassembler if you prefix it with a "$".

To enter an instruction into memory, type the location's address followed by a colon. Follow on the same line with the mnemonic and any operand required. Values must be in hex; no labels allowed. For following instructions, just type a space instead of an address with colon followed by the instruction. For instance, our example of Fig. 3-7 could be entered to the Miniassembler by typing

<div align="center">**!300:LDA F800**
! STA 1000</div>

```
! JMP FF69
!$FF69G
*
```

where the "!" and "*" are prompts; don't type them.

The Miniassembler looks up 6502 mnemonics for you, keeps adding the instruction lengths to a location counter for following instructions, does relative address calculations (more about that later), and reformats each line into the disassembler format as you enter them. Any two-byte operands are rearranged for you in address format, low byte followed by high byte.

Use the Miniassembler to hack and run. It lets you experiment quickly and easily by writing short routines almost as fast as you can think them up.

The disassembler gives twenty lines with each L command. You can hack up to twenty lines easily with the Miniassembler. While you can write routines longer than twenty lines this way, they won't fit on the screen. So, they become awkward to write and debug. Then, too, you can feel the need for labels when you design longer routines or write several routines that call each other. As a rough guide, twenty instructions is the practical limit for hack and run routines.

For longer projects, you need a two-pass text assembler.

A text assembler from a quality software house should give you the features you will need. It will have a good, easy to use editor that lets you document extensively and edit your text files in a line-oriented fashion. The assembler will recognize the commonly needed directives and use standard 6502 mnemonics. Some provide extensive features, but they should not interfere with using the common ones; you should not have to wade through a series of bells and whistles to set up a simple assembly.

Compared to the Miniassembler, a text editor/assembler package does the job in a fancier way. It has two passes instead of one, so that you can use labels. It also interprets assembler directives. You can make all the comments you want; unlike BASIC, assembler comments don't take up final program space. Most assembler packages provide printer output, selectable at assembly time. The disadvantage to an editor/assembler package is the long operating time, even for short and simple routines.

A two-pass assembler works the same as you would if you did a hand assembly. First, you write the assembler code on a text file using

the editor. This file is called the *source* file of your program. When you run the assembler, it reads your source file and creates another file called an *object* file. The assembler makes two passes of the source file to do this: one pass to build a table of all your labels and another pass to resolve them and create the object file. With a good, simple assembler, the object file is a binary file that you can BLOAD into memory to test or use. Each step of the process is the same as it was for hand assembly.

There are assemblers that will let you use external labels. They produce relocatable files that have a small label table with them, left over from the first pass. This lets you assemble routines that call each other separately, then link their relocatable files together later on to make the final loading file. To do this, another stage is needed after assembly — a stage called *linkage editing*. So, the package has a linkage editor and perhaps a librarian as well.

For most work, especially for the beginner, extra features like linkage editing are not necessary. On a small computer like the Apple II, simplicity and ease of use are important.

Often, you will have a short routine in machine language that you want to run from a BASIC program. You can do this by using the CALL command, but loading the machine code at first can be a bit tricky.

First, you can BLOAD it from a binary disk file. This is simple enough, but means your program now has two files, a BASIC one and a binary one. Maintenance would be much easier if you could somehow include the machine routine into the BASIC program. Then, you would have the entire program contained in one file.

The classical method, one which is still used on other microcomputers, is to *POKE* the routine into memory from the BASIC program's initial routine. The address and contents of each byte must be translated from hex to decimal notation before writing the sequence of POKE statements. Although it is slow and error-prone to write, this method was once popular among Apple programmers needing short machine routines.

The best place for single, short machine routines is at $0300, 768 in decimal. If the example of Fig. 3-7 is to be POKEd into place, the statements used would be

POKE 768,173 : POKE 769,0 : POKE 770,248 :
POKE 771,141 : POKE 772,0 : POKE 773,16 :
POKE 774,76 : POKE 775,105 : POKE 776,255

where 173 is the decimal form of $AD, for instance. When run, the routine is entered by the POKEs and the program can invoke it anytime by a CALL 768 statement.

The POKE method is rarely used on the Apple. Instead, an easier one called *Lam's method* is employed.

To enter machine code using Lam's method, you prepare a string that is exactly like the one you would enter to the Monitor. Then you call a BASIC subroutine written to put the command string into the keyboard buffer. This subroutine also calls the Monitor so that it thinks you gave it a keyboard command line. When the Monitor has entered the code into memory for you, it gets a final command that sends control to a BASIC routine that continues program execution. There are differences between Integer and Applesoft versions, but both do the same job.

From Integer BASIC, you can create a machine routine like this. DIMension a string called HEX$ for at least 80 characters. Write the following utility subroutine:

```
500 FOR H = 1 TO LEN(HEX$(H))
510 POKE 511+H, ASC(HEX$(H))
520 NEXT H
530 POKE 72,0
540 CALL -144
550 RETURN
```

Then, each time you want to enter code, make up a string:

```
30100 HEX$ = "300:4C D8 FD N E88AG"
30110 GOSUB 500
```

In this example, a JMP $FDDB was created at $300. Type the spaces shown; they are important. The N ends the memory entries. The E88AG makes Integer continue running your BASIC program.

For an Applesoft BASIC program, the same algorithm does the job. The utility subroutine becomes:

```
500 FOR H = 1 TO LEN(HX$)
510 POKE 511+H, ASC(MID$(HX$,H,1))+128
520 NEXT
530 POKE 72,0
540 CALL -144
550 RETURN
```

The call sequence for this subroutine is

```
30100 HX$ = "300:4C D8 FD N D823G"
30110 GOSUB 500
```

The main difference is the address of the continue routine. For Integer, it is at $E88A; for Applesoft it is at $D823.

Whenever you have a short routine, put it in Page Three, from your BASIC program, by using Lam's method. It is the best way to include machine routines that you wrote with the hack and run method.

3.2 ADDRESSING

3.2.1 The Addressing Modes

There are eleven different addressing modes of the 6502 processor. Each is described below.

IMPLIED — These are one-byte instructions because they don't need operands. From the op code, the processor knows which address, if any, to use. Many implied instructions, like INX, INY, DEX, DEY, TAX, TYA, etc., act on registers only. A few, like PHA, PLP, and RTS, use register contents to find their addresses. Some set and clear flags in the P-reg: CLI clears the I-flag, SED sets the D-flag, and so forth. All these instructions are only one byte in length; you don't have to give an explicit address.

RELATIVE — These are branch instructions, two bytes in length. These include BMI, BEQ, BCC, and so forth. The operand is a signed number that tells the processor to either add or subtract the PC to reach the branch address. The branch op code tests a flag. If the test fails, nothing more is done and the PC is pointing to the next instruction as per normal. But, if the test is true, then the relative address contained in the operand is added to the PC to make it point to the branch address instead.

A relative address may be any value from − 128, represented by $80, up to − 1, represented by $FF. Positive relative addresses then range from zero, $00 to + 127 represented by $7F. To calculate the relative address, subtract the address of the next instruction from the address of the branch instruction. For instance, consider the following:

```
                 BEQ THERE
          HERE   LDA #1
                 . . .

                 . . .
          THERE  LDA #2
                 . . .
```

The operand pointing to THERE in the BEQ instruction must be assembled as one byte. That byte is calculated as

THERE minus HERE

once the absolute addresses of THERE and HERE are known in the routine. This gives the instruction a single byte relative address operand.

IMMEDIATE — A two-byte instruction with a "#" prefixed to the operand in assembler form. It is used to read a literal byte directly from the operand. The operand byte is *not* an address, but is the actual data read by the instruction. For example, an LDA #$7F would cause a $7F to be loaded into the A-reg during execution. Other examples include: LDX #0, LDY #$FF, and ORA #$80. The last one, ORA, is a logical OR with the A-reg; it turns on Bit 7 in the A-reg using the $80 value.

ABSOLUTE — A three-byte instruction where the operand gives the address of the data to be handled. The address is always in low-byte to high-byte order. For instance,

LDA $1234

assembles as $AD, $34, $12 in that order. When executed, it loads the contents of location $1234 into the A-reg.

ZERO PAGE — Sometimes called zero page absolute, because it acts like the absolute mode. However, it is a two-byte instruction with the operand containing the low order byte of the address. The high order byte is implied to be zero; so, this mode addresses Page Zero locations only. For example, the instruction

LDA $50

assembles as $A5, $50 and is the same as

LDA $0050

which assembles as $AD, $50, $00. The zero page mode does execute faster. And it takes up only two bytes of program memory instead of three. The zero page mode op code calculates the *effective address* from the single byte. In this example, the effective address is $0050.

ABSOLUTE INDEXED BY X — A three-byte instruction that calculates the effective address as the sum of the operand and the X-reg. For example, if the assembler code is

LDX #$15
LDA $1234,X

then the LDA instruction would assemble as $BD, $34, $12. Upon execution, the $BD op code causes the effective address to be calculated as the sum of $1234 and $15. This is $1249, so the contents of location $1249 will be loaded into the A-reg to complete the instruction. The effective address is always taken as being the sum of the operand and the contents of the X-reg.

ZERO PAGE, INDEXED BY X — This is a two-byte instruction that calculates an effective address in Page Zero only. It only calculates the low-order byte of the effective address; the high-order byte is always zero. For example,

LDX #$23
LDA $34,X

will generate $B5, $34 from the LDA instruction. Upon execution, the $B5 calculates the effective address by first adding $34 and the operand to $23, which is the contents of the X-reg. The sum of $57 is then used as the low byte to make the effective address of $0057. Then the contents of location $0057 is read from memory and put into the A-reg.

Be careful using this mode. If the sum is greater than $FF, you do not address Page One; instead, the effective address *wraps around* to point into Page Zero again! For instance,

LDX #$FE
LDA $50,X

when executed, will load the contents of $004E into the A-reg, not the contents of $014E. If you want the boundary crossed into Page One, use the Absolute Indexed by X mode instead. However, the Zero Page indexed lets the X-reg function the same as a signed index. The example here functions the same as if the address $50 were indexed by a value of *minus two* when the X-reg has $FE in it.

ABSOLUTE INDEXED BY Y — A three-byte instruction that is similar to the Absolute Indexed by X. The effective address is calculated as the sum of the operand and the contents of the Y-reg. So,

<div align="center">

LDY #$45

LDA $2300,Y

</div>

will assemble the LDA as $B9, $00, $23. When run, the 6502 calculates the effective address as $2345. The A-reg is then loaded with the contents of location $2345.

INDIRECT — There is only one, the *jump indirect,* and it is three bytes in size. It has an op code of $6C. As an example,

<div align="center">

JMP ($0036)

</div>

assembles as $6C, $36, $00. Upon execution, the processor reads $0036 as the effective address's low byte. Then it reads the contents of $0037 as the high byte of the effective address. The effective address is put into the PC to complete the instruction.

Taking the example further, you can see it more explicitly. If $0036 contained $00, and $0037 contained $C5, then the effective address would be $C500. The jump instruction puts that into the PC so the next instruction is executed at $C500. The result is that the program jumps to the routine at $C500. If another address was stored in $0036.0037, then that is the address used for the jump. The locations $0036.0037 given by the operand point to the routine to be jumped to, rather than being the routine itself. The indirect mode is shown by using brackets.

INDIRECT INDEXED BY Y — A two-byte instruction that uses a Page Zero pointer and the Y-reg to calculate its effective address. The effective address is the sum of the pointer value from Page Zero and the contents of the Y-reg. As an example, the assembler code

```
LDY #$45
LDA #0
STA $50
LDA #$23
STA $51
LDA ($50),Y
```

represents part of a routine that uses indirect indexed. The Y-reg has $45, and the Page Zero pointer $50.51 has $2300 when the first five instructions have been executed. The sixth instruction, which assembles as $B1, $50, loads the A-reg with the contents of memory at $2345. The effective address, $2345, is calculated by first getting the pointer from the Page Zero location given by the operand, $50. This pointer is $2300. Then, the contents of the Y-reg, which is $45, is added. Compare the way this works with the Indexed by Y mode above. Where the Indexed by Y mode uses the absolute value of its operand to add to the Y-reg, the Indirect Indexed by Y mode uses its operand to point to the value to be added to the Y-reg.

INDEXED INDIRECT BY X — Another two-byte instruction that uses a Page Zero pointer and a register to calculate the effective address. This time, the X-register is used. Indirection now takes place *after* the indexing, so this mode is not the same as Indirect Indexed. The effective address is taken from a Page Zero pointer which in turn is calculated as the sum of the operand and the contents of the X-reg.

This mode is rarely used. In the few cases where it is used, most of the time the X-reg is set to zero to make it a simple indirect instruction. For example,

```
LDA #$34
STA $50
LDA #$12
STA $51
LDX #0
LDA ($50,X)
```

sets up the A-reg with the contents of $1234. The last instruction assembles as $A1, $50. On execution, it gets the operand value, $50, and adds the contents of the X-reg. In this case that is zero, so the sum is also $50. Then it fetches the effective address from $50 and $51 in Page Zero. In this example, that gives $1234. Suppose the X-reg were

set to $16 instead. In that case, the effective address would have been fetched from $66.67 in Page Zero because $66 is $50 plus $16.

3.2.2 Mnemonics

See Tables 3-4 and 3-5 for their op codes.

ADC   Add Memory to A-reg with Carry
AND   Logical AND Memory with A-reg
ASL   Arithmetic Shift Left (memory or A-reg)
BCC   Branch if C-flag is clear
BCS   Branch if C-flag is set
BEQ   Branch Equal Zero; branch if Z-flag set
BIT   Test Bits in Memory with A-reg
BMI   Branch Minus; branch if N-flag set
BNE   Branch Not Equal zero; branch if Z-flag clear
BPL   Branch Plus; branch if N-flag clear
BRK   Break to IRQ vector with B-flag set
BVC   Branch if V-flag clear
BVS   Branch if V-flag set
CLC   Clear C-flag
CLD   Clear D-flag
CLI   Clear I-flag
CLV   Clear V-flag
CMP   Compare Memory with A-reg
CPX   Compare Memory with X-reg
CPY   Compare Memory with Y-reg
DEC   Decrement Memory by one
DEX   Decrement X-reg by one
DEY   Decrement Y-reg by one
EOR   Exclusive OR Memory with A-reg
INC   Increment Memory by one
INX   Increment X-reg by one
INY   Increment Y-reg by one
JMP   Jump to new location
JSR   Jump to Subroutine, saving return address
LDA   Load A-reg with memory
LDX   Load X-reg with memory
LDY   Load Y-reg with memory
LSR   Logical Shift Right Memory or A-reg, one bit

NOP	No-Operation
ORA	Logical OR Memory with A-reg
PHA	Push A-reg onto stack
PHP	Push P-reg onto stack
PLA	Pull A-reg from stack
PLP	Pull P-reg from stack
ROL	Rotate Memory or A-reg, one bit left
ROR	Rotate Memory or A-reg, one bit right
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract Memory from A-reg; borrow from C-flag
SEC	Set C-flag
SED	Set D-flag
SEI	Set I-flag
STA	Store A-reg in memory
STX	Store X-reg in memory
STY	Store Y-reg in memory
TAX	Transfer A-reg to X-reg
TAY	Transfer A-reg to Y-reg
TSX	Transfer S-reg to X-reg
TXS	Transfer X-reg to S-reg
TXA	Transfer X-reg to A-reg
TYA	Transfer Y-reg to A-reg

Each instruction is described and explained in the context of programming in the following sections of this chapter. You can refer back to this list of mnemonics as necessary and many of the simpler ones like DEX and STY won't be explained further; you can understand them from their definitions.

3.3 PROGRAM FLOW

3.3.1 The CMP Instruction

Much of the art of programming consists in arranging alternate sequences of actions and decisions. At the machine language level, action instructions set flags in the P-reg and decision instructions read these flags. The secret to programming effectively with machine instructions is in knowing the P-reg flags — how they are set and how they are tested.

While there are seven active flags in the P-reg, only three of them are used often in controlling program flow. These are the N-, Z-, and C-flags.

The N-flag indicates that the last action instruction had a negative result. The value of the resulting byte was between $80 and $FF, inclusive, if the N-flag is set. If the result was positive, $00 to $7F, then the N-flag is clear. The N-flag therefore reflects the value of either zero or one of the Bit 7 in the last result.

Many actions change the N-flag. These actions are listed in Tables 3-4 and 3-5. They are indicated by an "N" in the Flags column.

The Z-flag also has many actions that can change its value. Whenever the result of an instruction is a zero, the Z-flag is set. Conversely, a nonzero result will clear the Z-flag.

A couple of examples. If the processor executes a LDX #$60 instruction, it will clear the N-flag and clear the Z-flag. It does so because the result of $60 is positive and nonzero. If a DEY instruction acted on $00 in the Y-reg, then the N-flag would be set and the Z-flag would be cleared. In this case the result is $FF in the Y-reg, which is negative and nonzero. Similarly, a LDA #$00 would set the Z-flag and clear the N-flag.

The C-flag is a little different. First, you can see from Table 3-5 it is changed by A-reg arithmetic and logic instructions. And, it can be changed by the compare instructions: CPX, CPY, and CMP. Arithmetic and logic will be done later; here the compares are important. In particular, the CMP is studied because it shows how the C-flag works.

In the P-reg, the N-flag is Bit 7, the Z-flag is Bit 1, and the C-flag is Bit 0. Bit 0 is the least significant bit (LSB).

Here is a routine that uses the N-flag to connect its action and decision. It looks at the keyboard and waits until a key is pressed. Then it prints the hex code for the keyboard character and returns to the Monitor:

```
0303: AD 00 C0    LDA  $C000
0300: 10 FB       BPL  $0300
0305: A2 10 C0    LDX  $C010
0308: 20 DA FD    JSR  $FDDA
030B: 4C 69 FF    JMP  $FF69
```

First, the keyboard is fetched from $C000. This action is tested by the BPL, which branches back to repeat the action at $0300 while the key-

board was not used. When a key is pressed, it gives a character greater than $7F; this sets the N-flag instead of clearing it. So, when a real keyboard character is in the A-reg, control simply falls through to the next instruction at $0305.

The LDX instruction resets the keyboard — a job any keyboard input routine must perform. The routine at $FDDA prints the hex code of the A-reg. It is called with a JSR instruction that does the same job for you as a GOSUB would in BASIC. Finally, the routine jumps to the Monitor at $FF69.

Look a little closer at the BPL instruction at $0303. The op code is $10; simple enough. But the operand is $FB. This is an address to give the BPL instruction a branch address of $0300 like the assembler form on the right says. The next instruction taken when the N-flag is set is $0305. So, when the N-flag is clear, the branch address is $0305 plus the operand $FB. The addition of this relative address to the PC is done with signed arithmetic; so, $FB is taken to mean − 5 to give a branch address of $0305 − 5 or $0300. You can enter and run this routine to see it work.

If you use one of the compare instructions — CMP, CPX, CPY — you can force all three flags. This gives you the N-, Z-, and C-flags to analyze with branch instructions. With the C-flag available, you can compare any two values and branch accordingly.

The CMP instruction makes a subtraction

A minus M

where A is the A-reg and M is the memory. If the result is zero, the Z-flag is set; otherwise, the Z-flag is cleared. If the result is greater than or equal to zero, the C-flag is set; otherwise, it is cleared. The N-flag reflects the result's Bit 7. The result does not replace the original value of the A-reg; unlike a subtraction instruction, the register being compared remains unchanged.

Similarly, you can compare the X-reg or the Y-reg with memory. The CPX and CPY both force the three flags without changing the register value being compared. Whenever you want to compare values without destroying the register, or you want to re-examine a register after other actions have altered the flags, then you can use a compare to do the job. They are the direct way to set these flags.

Usually, the other instructions that you use set the flags. Check Tables 3-4 and 3-5 and see those flags altered by the various instructions. Some of these are explicit: SEC and CLC in particular.

The decision instructions are the branches. One of these, the BPL, gave the example of the keyboard routine earlier. The BPL tests the N-flag; the BMI also tests the N-flag, but it branches when the result has Bit 7 set.

There are two branches for testing the Z-flag. The BEQ will branch if the Z-flag is set; that is, a zero result. And, the BNE will branch if the Z-flag is clear; when the result was not zero.

The C-flag is tested with either a BCC or a BCS. By using a BCC, the branch occurs when the C-flag is clear. A compare instruction where the register is less than the memory will also do this. Similarly, a BCS branches when the C-flag is set. Comparing a register with an equal or smaller memory will set the C-flag.

The compare is always unsigned; you are subtracting absolute values. So, $84 is greater than $56, for example. To work with signed numbers, you need the N-flag. But most compares are done with unsigned numbers using the C-flag. These results are summarized for you in Table 3-6.

Table 3-6. The Results of a CMP

	N-flag	C-flag	Z-flag
A < M	Either	Cleared	Cleared
A = M	Cleared	Set	Set
A > M	Either	Set	Cleared

If you are learning a machine language for the first time, the following experiments will help you. Enter and run them; they are well worth your time.

First, assemble the following routine:

```
              ORG  $0300
     TEST:    LDX  #$FF
              LDA  ARG1
              CMP  ARG2
              BEQ  TEST1
              LDX  #0
     TEST1:   STX  RESULT
              JMP  $FF69
     ARG1:    DS   1
     ARG2:    DS   1
     RESULT:  DS   1
```

The purpose is to test the BEQ instruction with different values of ARG1 and ARG2. The CMP instruction has ARG1 as its A-reg value and ARG2 as its memory value. The X-reg is loaded with $FF at the beginning. If the branch is taken, it puts that $FF into RESULT memory. If the branch is not taken, it becomes $00 (zero) instead; RESULT is zeroed. So, you can see if the branch was taken or not by examining RESULT in memory.

The parameters are: ARG1 at $0312, ARG2 at $0313, and RESULT at $0314. To make a run, use the Monitor to set ARG1 and ARG2 to the values of A-reg and to the memory that you want. Then, run the routine at $0300 using the G command. And finally, examine the contents of RESULT to see whether or not the branch was taken.

For the BEQ instruction, what will be the result when ARG1 is greater than ARG2? When they are equal? When ARG1 is greater than ARG2?

Make three more runs, but change the branch instruction first. Instead of a BEQ, use a BNE. What results do you expect now?

Again, change the branch instruction for three more runs. Use a BCC. Then, change to a BCS for three more runs.

You can use this routine for any experiments that you may need to understand the branches. For now, you should verify the results summarized in Table 3-6. Later, you can always run more routines if you want to figure out the actions of other flags.

By using just the C-flag and the Z-flag, you can make five different decisions. In a program, you can branch for one of five conditions: greater than; greater than or equal; equal; less than or equal; less than. The branches for each condition are in Table 3-7. Use the code for the case you want.

Table 3-7. Branch After Compare

Condition	Branch on Condition
A-reg < memory	BCC CASE1
A-reg ≤ memory	BCC CASE2
	BEQ CASE2
A-reg = memory	BEQ CASE3
A-reg ≥ memory	BCS CASE4
A-reg > memory	BEQ NEXT
	BCS CASE5
	NEXT:

3.3.2 The Stack

In machine code, a JMP instruction does what a GOTO does in BASIC. And, a JSR does what a GOSUB does. The difference between JMP and JSR is that the JSR remembers where it jumped from. This way, a return instruction called RTS can recall the old address for the PC in the processor.

The JSR and RTS lets you make subroutines easily. And there are other instructions that save and recall values automatically, with implied addressing. You don't have to keep track of their storage in RAM because the processor uses the S-reg to do that automatically. The RAM used for this storage is Page One; it is called the *stack*. Similarly, the S-reg is called the *stack pointer*. By pointing to the stack, the S-reg remembers where the next unused location is available for storage.

One pair of stack instructions is the PHA and PLA. The PHA is called *push A-reg*; it stores the A-reg in Page One. Then, the *pull A-reg* instruction, PLA, loads the A-reg from Page One. Another pair is PHP and PLP. They push the P-reg and pull the P-reg from the stack. These push and pull instructions do the same thing as store and load instructions, but they use and modify the S-reg as well.

A push is done by storing the byte to Page One, using the S-reg as the address-low. After the store, the S-reg is decremented by one to point to the next location.

A pull is done by first incrementing the S-reg by one. Then, the byte is read by using the S-reg as address-low and $01 as address-high.

In use, the push and pull instructions save and recall bytes from the A-reg or P-reg. By automatically changing the S-reg by one each time, the processor always keeps track of the bytes stored in Page One on a last-in, first-out basis. You can think of the bytes being stacked, one atop the other, with the last one pushed on the top. This last byte is the first one available to be pulled from the stack.

For example, suppose you used the PHA instruction to push three bytes onto the stack from the A-reg:

```
LDA   #$01
BHA
LDA   #$02
PHA
LDA   #$03
PHA
```

The S-reg then points to the next location beyond the one containing the $03 value. Pulling a byte at this time will get the $03 from the stack and leave the stack pointer (the S-reg) pointing there. Another pull increments the S-reg again and loads the $02 from the stack. The third pull does the same thing, fetching the $01. The S-reg is now pointing to the $01's location as the first free memory available, just as it did before the three pushes were made.

A push stores a byte and decrements the stack pointer; a pull increments the stack pointer and loads a byte.

Addresses are pushed and pulled by the JSR and RTS instructions. They are sixteen bits instead of eight, so the processor has to push twice from the PC to do a JSR and pull twice to do an RTS.

Here are the scenarios of the processor performing a JSR and an RTS.

Assume the processor is executing the JSR at $0300:

```
$0300:    JSR    $FDF0
$0303:    CLC
```

It reads the operand bytes, $F0 followed by $FD. Then, the PC is pointing to the last byte of the operand, which is the $FD in location $0302. The 6502 then pushes the high byte of the PC on the stack, $03 in this case. Then it pushes the low byte of the PC on the stack, $02 here. After that, it makes a jump by putting the operand bytes into the PC: the $FDF0 for this case. The next instruction is then at location $FDF0; the JSR is complete.

The subroutine runs, perhaps calling others, until it reaches an RTS.

Executing the RTS, the processor pulls the old address from the stack: first the low byte, then the high byte. They are stuffed into the PC to give an address of $0302 in this example. Now, the processor again points to the last byte of the JSR instruction. To complete the instruction cycle for the RTS, the processor increments the PC by one so as to point to the next instruction. In this case, the PC becomes $0303 to point to the CLC instruction there. That completes the RTS instruction.

The one point you should watch for if you use the stack pointer from within the subroutine is that the JSR has stacked the address of its third byte, *not* the address of the next instruction as a simpler description of the JSR might lead you to believe. If you load the PC

from the stack by using an RTS, remember that the address you load must therefore be one less than the address of the instruction to be executed next.

Such trickery is, fortunately, invisible to the normal use of the JSR and RTS instructions.

The other instructions that use the stack are the ones associated with interrupts. IRQ and NMI interrupts push return addresses and the P-reg onto the stack. The RTI instruction pulls the P-reg and returns from the interrupt routine to the one that was interrupted.

3.3.3 Structures

The way any routine you write does its job is called the algorithm of the routine. The simplest possible routines have simple algorithms that cannot be broken down further; these algorithms follow a form called a structure. To write any routine, you must design an algorithm that uses one or more of these structures. Knowing the structures and what each will or will not do is necessary if you want to write uncomplicated programs. Otherwise, the routines may have algorithms that can become impossible to test, debug, use, and maintain.

With structures, you can put instructions together to do more intelligent tasks than they can do by themselves.

The simplest way a routine can be written is with a sequence of actions, without decisions. For example, you can move the contents of one memory location to another:

```
MOVE:   LDA  HERE
        STA  THERE
        RTS
```

A longer routine could move several bytes of memory:

```
MOVES:  LDA  HERE
        STA  THERE
        LDA  HERE + 1
        STA  THERE + 1
        LDA  HERE + 2
        STA  THERE + 2
        RTS
```

You can use indirect indexed addressing to generalize this routine. Then, it could be used for any three contiguous memory locations, not just at HERE and THERE:

```
MOVES:  LDY  #0
        LDA  (HEREZ),Y
        STA  (THEREZ),Y
        LDY  #1
        LDA  (HEREZ),Y
        STA  (THEREZ),Y
        LDY  #2
        LDA  (HEREZ),Y
        STA  (THEREZ),Y
        RTS
```

In this case, you must put the first address of the three source bytes in the Page Zero pointer at HEREZ and HEREZ + 1. Then, put the address of the first destination byte in the Page Zero Pointer, THEREZ and THEREZ + 1.

Such a call sequence of setting Page Zero pointers before making a JSR is quite common. Setting them with immediate values, in assembler notation where the desired addresses are given by labels, can be done like:

```
LDA  #>SOURCE      ;low byte
STA  ZHERE
LDA  #<SOURCE      ;high byte
STA  ZHERE + 1
LDA  #>DEST        ;low byte
STA  ZTHERE
LDA  #<DEST        ;high byte
STA  ZTHERE + 1
JSR  MOVES
```

In the notation of the DOS Toolkit Assembler from Apple, the ">" selects the low byte of a labeled address and the "<" selects the high byte.

Similarly, you can write a SWAP routine to exchange the contents of three locations with three other locations. It can have the same call sequence, using HEREZ and THEREZ as Page Zero pointers:

```
SWAP:   LDY  #0
        LDA  (HEREZ),Y
        PHA
        LDA  (THEREZ),Y
        STA  (HEREZ),Y
        PLA
        STA  (THEREZ),Y
        LDY  #1
        . . .
        LDY  #2
        . . .
        RTS
```

where the ellipsis . . . indicates that the block of code swapping one pair of bytes is repeated. Note how the stack is used for temporary storage of the byte from one location until the A-reg becomes available for it again.

Other sequential routines you may write include ones to save and recall all the registers, setting soft switches for a particular screen configuration, setting Monitor parameters in Page Zero, and so on.

Use the sequential routine structure to do simple, low level tasks. It is fast in execution. It won't handle any decisions or a large number of repetitions, however.

To handle large numbers of repetitions, you use a loop. There are two kinds of loops: one decides before acting; the other acts before deciding. The second type of loop is the easiest to write and it is called a DO-UNTIL loop.

The DO-UNTIL loop is the kind used in BASIC where the decision to leave the loop is made at the bottom, in the NEXT statement. Most DO-UNTIL loops in machine code look like a BASIC FOR-loop, when using a counter. One of the index registers is often used for DO-UNTIL loops.

As an example, here is the MOVE routine written using a loop:

```
MOVE:    LDX  #$1F
MOVE1:   LDA  HERE,X
         STA  THERE,X
```

```
                        DEX
                        BPL    MOVE1
                        RTS
```

This routine copies 32 bytes, starting at HERE, to the 32 bytes starting at THERE. The decision at the bottom of the loops is made after the DEX instruction, which forces the Z-flag and the N-flag. Since all X-values wanted are positive, having values in the $00.1F range, the BPL will fail to branch when the count changes from $00 to $FF.

If the range is beyond $00.7F, you cannot use the N-flag. Instead, use the Z-flag, which is also changed by the DEX, DEY, INX, and INY instructions:

```
            MOVE:       LDX    #$A0
            MOVE1:      LDA    HERE-1,X
                        STA    THERE-1,X
                        DEX
                        BNE    MOVE1
                        RTS
```
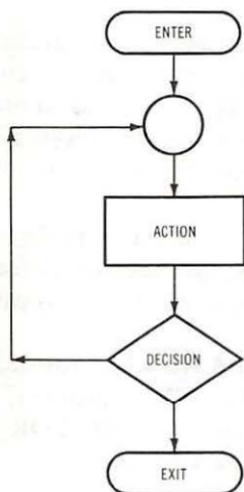

Fig. 3-8. The DO-UNTIL structure.

This copies $A0 bytes. The X-reg is initialized with $A0, which is the number of bytes to copy, instead of one byte less. And, the operand addresses are one less than the lowest address of interest. This is be-cause the X-reg ranges from $A0 down to $01 as it copies. When it be-

comes zero, the loop is finished. This is better because you aren't restricted to $7F to satisfy the N-flag.

You can use a loop counter with Page Zero pointers as well. For an index, use the Y-reg; this lets you use the indirect indexed addressing mode:

```
MOVE:      LDY   #$A0
MOVE1:     LDA   (HEREZ),Y
           STA   (THEREZ),Y
           DEY
           BNE   MOVE1
           RTS
```

Remember, HEREZ and HEREZ + 1 must be initialized to point to one location *before* the lowest address of the move. The same is true for THEREZ and THEREZ + 1.

Indexes are great when the count is 256 or less. But if the count is greater than 256, you will have to use a separate counter, preferably in Page Zero.

The DO-UNTIL structure appears in Fig. 3-8. Use it whenever you need a loop that must repeat its action at least once.

Sometimes, you need a loop that must be able to avoid its own action. Suppose you want to set the size of the MOVE in the call sequence instead of forcing it with an immediate value. If the number of bytes to be moved is zero, the DO-UNTIL loop isn't quite smart enough to quit before moving the first byte. To handle such a job, you use another loop structure called the DO-WHILE.

As an example of a DO-WHILE loop, here is a MOVE routine that copies from HERE to THERE, and expects the number of bytes in the X-reg:

```
MOVE:      CPX   #0
           BEQ   MOVE1
           LDA   HERE – 1,X
           STA   THERE – 1,X
           DEX
           JMP   MOVE
MOVE1:     RTS
```

First, the CPX forces the flags; especially the Z-flag. The BEQ will branch to the RTS exit instruction if the X-reg is zero. So, even if this is the first time through the routine will exit before taking any action. The action consists of the move followed by the decrement of the index. After all actions, the JMP forces the loop back to the top to test again.

Even though the DO-WHILE is not used as often as the DO-UNTIL, keep it in mind. With only a little extra programming effort the DO-WHILE loop will make an otherwise bug-prone loop *fail-safe*. The DO-WHILE structure is shown in Fig. 3-9.

When you don't need a loop, you can make a simple decision by using an IF-THEN-ELSE structure. This simple structure has a decision for one of two possible actions. More complex decisions can then be made by using a series of simple IF-THEN-ELSE structures.

The CMP experiments described earlier use this structure. Each routine had two possible outcomes: either the result was zero or it was 255 in value. While that particular one assumed one of the outcomes, then replaced it with the other when called upon, you can write IF-THEN-ELSE structured routines that don't assume any details of each action.

For instance, take the experiment comparing ARG1 and ARG2. If you wrote it to call one of two possible routines, then each consequential action could be written as a separate subroutine. You would have complete control in making the results anything you wanted for the two outcomes.

```
         TEST:   LDA   ARG1
                 CMP   ARG2
                 BEQ   TEST1
                 JSR   FAILED
                 JMP   TEST2
TEST1:   JSR PASSED
TEST2:   RTS
```

If the branch is made, the subroutine PASSED is called and then the routine exits. If the branch is not made, the other subroutine, FAILED, is called. Upon return from the FAILED subroutine, a jump to the exit point is made. So, ARG1 and ARG2 are compared; one of two subroutines — PASSED or FAILED — is the consequence.

Fig. 3-9. The DO-WHILE structure.
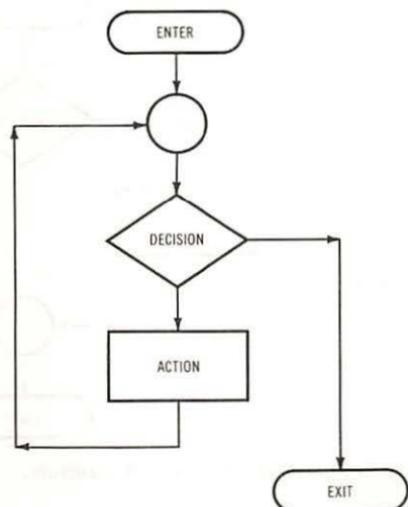
You can make the subroutines anything you want. You can just make

```
PASSED:  LDA  #$FF
         STA  RESULT
         RTS
```

and make

```
FAILED:  LDA  #0
         STA  RESULT
         RTS
```

to get the same performance as the original.

The IF-THEN-ELSE structure is shown in Fig 3-10. You can use it to separate out actions and decisions for simpler routines. This structure can be compounded into multiple decisions when you need complex logic.

One such complex structure is called the CASE. It extends the simple IF-THEN-ELSE structure from two outcomes to several outcomes: three, four, or as many as needed. For example, suppose you wanted to examine an input character from the keyboard in a graphics
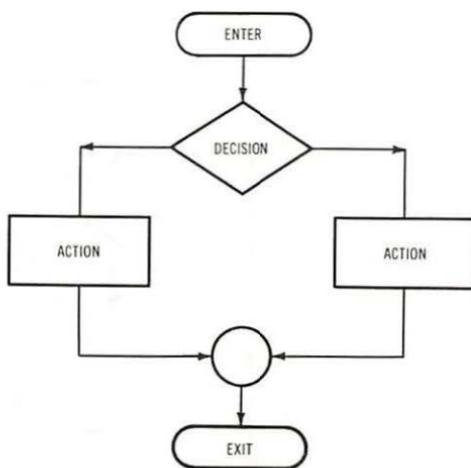
Fig. 3-10. The IF-THEN-ELSE structure.

program. You want to use the four keys to change the cursor on the screen, but ignore all the other keys. Assuming the character is given in the A-reg, here's the CASE routine to select one of four cursor movement routines:

```
CURSOR:   CMP   #'I'           ;if I then UP
          BNE   CURS1
          JSR   UP
          JMP   CURSX
CURS1:    CMP   #'J'           ;if J then LEFT
          BNE   CURS2
          JSR   LEFT
          JMP   CURSX
CURS2:    CMP   #'M'           ;if M then DOWN
          BNE   CURS3
          JSR   DOWN
          JMP   CURSX
CURS3:    CMP   #'K'           ;if K then RIGHT
          BNE   CURSX
          JSR   RIGHT
CURSX     RTS
```

Each of the four characters is tested against the A-reg. If any one is found, its corresponding subroutine is run. If none are found, no sub-

routine is run. After running any subroutine, control is directed to CURSX where the routine exits. The caller must give the character in the A-reg.

Such a CASE structure isolates the decision of which routine to run from the details of the routine itself. You could use this same CASE on LORES, HIRES, or even text cursor routines. Just give it the appropriate subroutines for UP, DOWN, LEFT, and RIGHT in your program. The CASE structure itself appears in Fig. 3-11.
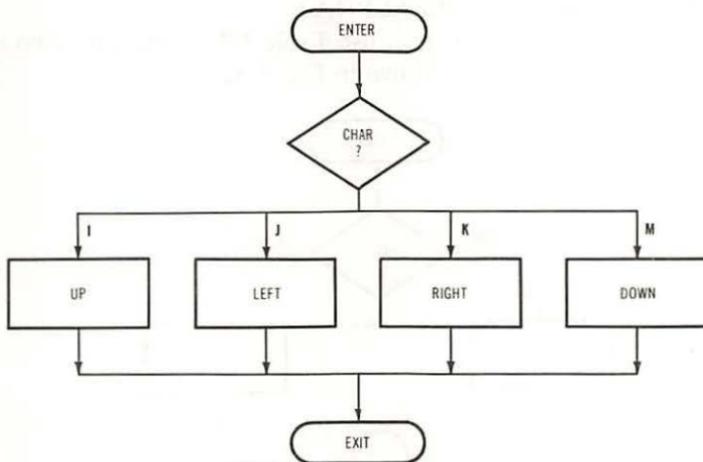
Fig. 3-11. CURSOR— example of a case structure.

Another extension of the IF-THEN-ELSE structure is the range test. The IF-THEN-ELSE uses a branch instruction or two to make a simple decision like one from Table 3-7. But, to see if a value falls within a range, you need two tests. However, there are only two outcomes — FAIL or PASS. So, you can combine the two tests you need in one algorithm to simplify the call structure of your program.

Here is a common range test. It makes sure that a character code in the A-reg is a letter, from A to Z.

```
ALPHA:    CMP   #'A'        ;range test
          BCC   ALPHA2      ;from "A" to "Z"
          CMP   #'Z'        ;inclusive
          BEQ   ALPHA1
          BCS   ALPHA2
```

```
ALPHA1:  JSR   PASS
         JMP   ALPHA3
ALPHA2:  JSR   FAIL
ALPHA3:
```

The first test branches if the A-reg is less than A to ALPHA2 because that is an obvious failure. Then the Z test passes any characters equal to Z by the BEQ to ALPHA1. This is followed by failing any that are greater than Z by the BCS to ALPHA2.

To make any other range test, use Table 3-7 to select the two tests you need. The range test is shown in Fig. 3-12.
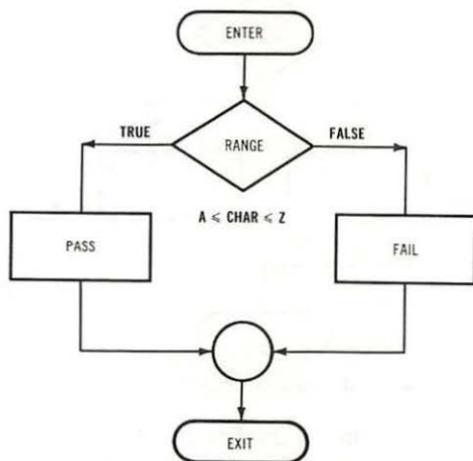
Fig. 3-12. ALPHA— example of a range structure.

When programming, break down the job into the simplest routines you can. If possible, use only one of these six structures for each routine. Each routine should have only *one entry point,* the first executable statement in the routine. And it should have only *one exit point,* as well. Use jumps and branches to the last instruction in the routine, which is usually an RTS. One of these six structures should be right for each of your routines. Use Table 3-8 to help you select the one you want.

Table 3-8. Program Structure Selection

Structure	Primary Use	Advantages
Sequence	Low-level	High speed
Do-Until	Most common loop	Short, easy to write
Do-While	Alternate loop	Stronger test
If-Then-Else	Decision Logic	Easy to follow
Case	Interpreters	Simplifies actions
Range	Data Editors	Simplifies actions

3.3.4 Methods

There are many tricks of the trade and fancy methods used to write routines using the structures. Here are a few of the more common ones.

Some routines must have a constant and known execution time. For instance, the routines that read and write bytes to the disk in DOS must do so at intervals of exactly thirty-two clock cycles. A routine that has such a measured execution time is called a *real-time* routine.

Real time is the solution for hardware service needs when times are too short for interrupts. Or where interrupts aren't available, such as when writing a sound generator for the built-in speaker. And, real time can be used in utilities like the Monitor's WAIT routine at $FC58.

To calculate the routine's execution time, use the number of clock cycles for each instruction given in Table 3-9. Add up the total number of cycles for the entire routine and multiply by 0.977778 microseconds.

Here are some short delay routines.

The shortest routine you can have is one with just an RTS instruction. When called, the JSR takes six cycles, and the RTS takes six cycles. So, the entire call takes twelve cycles.

You can increase the length of the call two cycles at a time, by including NOP instructions. For example,

```
WAIT:    NOP
         NOP
         RTS
```

takes sixteen clock cycles to call. For longer times, just add more NOPs.

Instruction Addressing Modes and Related Execution Times
(Courtesy MOS Technology, Inc.)

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect,X)	(Indirect),Y	Absolute Indirect
ADC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
AND	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
ASL	2	.	5	6	.	6	7	.	.	.	.	.	.
BCC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BCS	.	.	.	.	.	.	.	.	.	2**	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	2**	.	.	.
BIT	.	.	3	.	.	4	.	.	.	.	.	.	.
BMI	.	.	.	.	.	.	.	.	.	2**	.	.	.
BNE	.	.	.	.	.	.	.	.	.	2**	.	.	.
BPL	.	.	.	.	.	.	.	.	.	2**	.	.	.
BRK	.	.	.	.	.	.	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BVS	.	.	.	.	.	.	.	.	.	2**	.	.	.
CLC	.	.	.	.	.	.	.	.	2	.	.	.	.
CLD	.	.	.	.	.	.	.	.	2	.	.	.	.
CLI	.	.	.	.	.	.	.	.	2	.	.	.	.
CLV	.	.	.	.	.	.	.	.	2	.	.	.	.
CMP	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
CPX	.	2	3	.	.	4	.	.	.	.	.	.	.
CPY	.	2	3	.	.	4	.	.	.	.	.	.	.
DEC	.	.	5	6	.	6	7	.	.	.	.	.	.
DEX	.	.	.	.	.	.	.	.	2	.	.	.	.
DEY	.	.	.	.	.	.	.	.	2	.	.	.	.
EOR	.	2	3	4	.	4	4*	4*	.	.	6	5	.
INC	.	.	5	6	.	6	7	.	.	.	.	.	.
INX	.	.	.	.	.	.	.	.	2	.	.	.	.
INY	.	.	.	.	.	.	.	.	2	.	.	.	.
JMP	.	.	.	.	.	3	.	.	.	.	.	.	5
JSR	.	.	.	.	.	6	.	.	.	.	.	.	.
LDA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
LDX	.	2	3	.	4	4	.	4*	.	.	.	.	.
LDY	.	2	3	4	.	4	4*	.	.	.	.	.	.
LSR	2	.	5	6	.	6	7	.	.	.	.	.	.
NOP	.	.	.	.	.	.	.	.	2	.	.	.	.
ORA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
PHA	.	.	.	.	.	.	.	.	3	.	.	.	.
PHP	.	.	.	.	.	.	.	.	3	.	.	.	.
PLA	.	.	.	.	.	.	.	.	4	.	.	.	.
PLP	.	.	.	.	.	.	.	.	4	.	.	.	.
ROL	2	.	5	6	.	6	7	.	.	.	.	.	.
ROR	2	.	5	6	.	6	7	.	.	.	.	.	.
RTI	.	.	.	.	.	.	.	.	6	.	.	.	.
RTS	.	.	.	.	.	.	.	.	6	.	.	.	.
SBC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
SEC	.	.	.	.	.	.	.	.	2	.	.	.	.
SED	.	.	.	.	.	.	.	.	2	.	.	.	.
SEI	.	.	.	.	.	.	.	.	2	.	.	.	.

Instruction Addressing Modes and Related Execution Times
(Courtesy MOS Technology, Inc.)

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect,X)	(Indirect),Y	Absolute Indirect
STA	.	.	3	4	.	4	5	5	.	.	6	6	.
STX*	.	.	3	.	4	4	.	.	.	.	.	.	.
STY**	.	..	3	4	.	4	.	.	.	.	.	.	.
TAX	.	.	.	.	.	.	.	.	2	.	.	.	.
TAY	.	.	.	.	.	.	.	.	2	.	.	.	.
TSX	.	.	.	.	.	.	.	.	2	.	.	.	.
TXA	.	.	.	.	.	.	.	.	2	.	.	.	.
TXS	.	.	.	.	.	.	.	.	2	.	.	.	.
TYA	.	.	.	.	.	.	.	.	2	.	.	.	.

* Add one cycle if indexing across page boundary
**Add one cycle if branch is taken. Add one additional if branching operation crosses page boundary

You may need an odd number of cycles, such as for the pair of CLC and BCC, which takes five cycles. So,

```
        WAIT    CLC
                BCC WAIT1
        WAIT1   RTS
```

takes 17 cycles, and

```
        WAIT    NOP
                NOP
                CLC
                BCC WAIT1
        WAIT1   RTS
```

takes 21 clock cycles to call.

For longer wait times, you can use a loop. The time taken each pass through the loop will vary with the path taken, so the calculation of the total real time is a bit more involved. In general, the loop

```
        WAIT:   . . .
                body of n cycles
                . . .
```

<div align="center">

DEX
BNE WAIT
RTS

</div>

uses $n + 5$ cycles each branch and $n + 4$ cycles when the branch fails. The number of times through the loop is given by the X-reg. The formula is,

$$t = (n + 5)X + 11$$

where,   n is number of cycles in body of loop,
          x is contents of X-reg when called,
          t is total number of cycles, including JSR and RTS.

This calculation gives the relation between the time taken in the loop and the time taken for the entire routine. Solving for x or n will give you the information needed to make a delay for any given time.

You can delay routines to control the pitch on speaker routines, to slow the output to a printer interface, or to slow a video display like Applesoft's SPEED= feature.

In programming, you spend most of your time working with formats of various data; very little time with any *clever* algorithms needing a lot of math. So, the better organized your data is the less programming time needed to support it. When you combine this with the fact that there are only two kinds of data structures, both simple, then you can avoid a lot of work by carefully structuring your data to follow its own function.

For example, consider the problem of code conversion. If you want to use an Apple computer to communicate with an IBM computer system, you must change each ASCII character to an EBCDIC equivalent before you can output it to your modem. And each character received from the line must be changed from EBCDIC to ASCII before your Apple routines can make use of it. An algorithm could CMP each possible character in a "humongous" CASE structure, but there can be as many as 128 characters to compare — a mess to program! Since the problem appears to you in the form of a *table* where each entry represents the EBCDIC code of its *ordinal* (position in the table), it makes sense to use the simple code table in memory and use indexed addressing to do the lookup. The index address

LDA EBCDIC,X

looks up a table called EBCDIC and returns the X-th entry from the table in the A-reg. So, if each entry is the EBCDIC equivalent of its ASCII ordinal (X-reg) then this single instruction does the job of converting from ASCII to EBCDIC.

To continue the example, you can use the same table to convert from EBCDIC to ASCII when receiving from the IBM computer. This time you have the entries that you want to match, which will give you the position in the table. Here you need a loop to do the search. Assuming the EBCDIC character is in the A-reg:

```
                              ; largest entry
ASCII:      LDX  #$7F         ; number of entries
ASCII1:     CMP  EBCDIC,X     ; WHILE not found
                              DO
            BEQ  ASCII2
            DEX               ; decrement ordinal
            BNE  ASCII1       ; ENDWHILE
ASCII2:     RTS
```

the ASCII code is returned in the X-reg. If the search fails and an EBCDIC entry can't be found (X-reg contains zero) then the ASCII code NUL is returned, which the Apple usually ignores.

Such a table is quite simple to use. It has a fixed size and is not changed at all by the program using it. Other tables may be *variable* so that entries may be made and deleted during the program execution. The simplest example of a variable table is a *stack*. You saw the processor stack function already, where the table is in Page One and the pointer to the next available spot is in the S-reg inside the processor. But you can make your own stack using the zeroth location of the page as the pointer, that is if the stack only has one page. Otherwise, use a Page Zero pointer. Two simple routines maintain the stack — STACK and UNSTACK.

The STACK routine

```
STACK:   LDX  #0
         STA  (STAKZ,0)    ;put on stack
         INC  STAKZ        ; and push
```

```
              BNE   STACK1
              INC   STAKZ + 1
      STACK1: RTS
```

places the contents of the A-reg on the stack by writing to the next available location given by STAKZ in Page Zero. After that, it increments the Page Zero pointer by one, taking care to increment the high byte if necessary, thereby *pushing* the byte onto the stack. Reading from the stack works just the opposite: *popping* the last byte from the stack, then fetching it to the A-reg.

```
      UNSTACK:  LDX   #$FF
                DEC   STAKZ      ; pop stack
                CPX   STAKZ
                BNE   UNSTACK1
                DEC   STAKZ#1
      UNSTACK1: LDX   #0
                LDA   (STAKZ,0)  ; and fetch
                RTS
```

Stacks can be used for many kinds of data. One popular use of a data stack is to hold parameters during subroutine calls. This keeps them safe; in fact the routine may even call itself and keep its parameters separate for each call.

Stacks are great where a LIFO access can be used, but won't do the job where entries must be inserted and deleted anywhere in the table. In that case, all entries below the insertion point must be pulled down to make room for the new entry. Similarly, deletion consists of pushing all the lower entries back up a notch. If not done too often, it could be the way to go.

Another way you use tables is in computed JSRs. This is a technique of calling one of several subroutines like you do with computed GOSUBs in BASIC. With this method, you call a dispatching routine that selects one of many subroutines according to a simple number in the A-reg: 0, 1, 2, 3, ... The addresses of the subroutines are kept in a table in the usual low-byte/high-byte order. An assembler will stuff the table for you automatically if you just list the labels, making the method easy to use and maintain.

There are two easy ways to write a computed JSR dispatcher. The first one uses RAM for temporary storage of the subroutine's address.

This RAM may be Page Zero, but is is not necessary; you can use Page
Three or anywhere you have space. For example,

```
SUBR:     DW    SUBR0        ;list of subroutines
          DW    SUBR1
          DW    SUBR2
          etc.
CALSUB:   ASL                ;A-reg * 2
          TAX                ; as 0, 2, 4, 6, . . .
          LDA   SUBR,X
          STA   TEMP
          LDA   SUBR+1,X
          STA   TEMP+1
          JMP   (TEMP)
```

where DW is an assembler directive − pseudocode − to insert the
address location of the label as two bytes in low/high order. The ASL
multiplies the A-reg by two, simply by shifting all bits one position
left. Some assemblers want this mnemonic to include an A as an oper-
and: ASL A. The RTS of each routine called will return you to the
routine that called CALSUB in the first place.

The other method eliminated the problem of finding an unused
chunk of RAM for TEMP. Instead, the call address is put on the stack
and the routine desired is jumped to by an RTS instruction. This
works, but in a "sneakier" manner than the indirect jump method
above.

The trick to letting an RTS do the jump is to make the addresses in
the table one less than where the routines actually start. This is be-
cause the RTS increments the PC-reg by one at the end of its instruc-
tion. This is why the addresses of the Apple Monitor routines that are
dispatched from the top end of ROM are listed as being one less than
their start addresses. The CALSUB routine can be rewritten this way
as

```
SUBR:     DW    SUBR0 − 1
          DW    SUBR1 − 1
          DW    SUBR2 − 1
          etc.
```

```
CALSUB:  ASL                  ;mult by 2
         TAX
         LDA    SUBR + 1,X    ;stack addr – hi
         PHA
         LDA    SUBR,X        ;stack addr – lo
         PHA
         RTS                  ;a sneaky JSR!
```

The address is pushed on the stack the same way as the JSR instruction: hi-byte, then lo-byte. The address location is one *less* than the next executable instruction as well. If you use this routine be sure to *document* it.

If you have to keep a loop counter over a range beyond 256, or if you need to keep a pointer on a more permanent basis than an index will allow, use Page Zero. Anytime you want to reference memory with that pointer, use indirect addressing, like:

```
         LDX    #0
         LDA    (ZPOINT,X)
```

Initialize the pointer by putting the address of the first memory location you will access into ZPOINT and ZPOINT + 1. For instance, to point to $4000:

```
         LDA    #$00          ;low byte
         STA    ZPOINT
         LDA    #$40          ;high byte
         STA    ZPOINT + 1
```

Remember, if you are using Apple's Toolkit Assembler, use ">" for low byte and "<" for high byte. If the BUFFER was EQUated to $4000, you would write

```
         LDA    #>BUFFER
         STA    ZPOINT
         LDA    #<BUFFER
         STA    ZPOINT + 1
```

instead.

To increment a Page Zero pointer by one, use a routine such as:

```
ZINCR      INC  ZPOINT
           BNE  ZINCR1
           INC  ZPOINT + 1
ZINCR + 1  RTS
```

And, to decrement it,

```
ZDECR      DEC  ZPOINT
           LDA  ZPOINT
           CMP  #$FF
           BNE  ZDECR1
           DEC  ZPOINT + 1
ZDECR1     RTS
```

Each routine changes the low byte of the pointer first. Then, each routine tests for a page boundary crossing. When incrementing, this is when the low byte becomes zero; hence the BNE. When decrementing, the page changes when the low byte becomes $FF (from $00). That is not detected by a simple branch, so a CMP is used.

Much of the power of 6502 programming is in managing the Page Zero pointers. Page Zero is like registers in larger processors; you can do all kinds of things with them using indirect addressing.

3.4 INTERRUPTS

3.4.1 How They Work

Interrupts get the processor's attention. By using them, the *outside world* can tell the processor when a peripheral needs service, when to reset with initialization routines, and when to execute single instructions during the debugging process. By choosing the proper interrupt for a job and by writing the proper routine, you can control the 6502 processor.

The interrupt you would use to service a peripheral is called the IRQ — Interrupt ReQuest. You may, for example, have a slow printer that needs the processor to send it characters one at a time. The IRQ can be used to ask for these characters while allowing the processor to run other programs at the same time.

The printer has a line to request that a new character be sent; that line can be used to generate IRQs. The IRQ routine can then send the character and return to the interrupted program. The entire effect is to allow the program to run with interrupts small enough to be un-noticed. To the user, it is as if the printer received the characters without getting them from the processor.

This needs some smart hardware on a peripheral card. It must enable or disable IRQs as needed. It must handle all the data and con-trol lines to the device itself. And, it must provide the IRQ routine memory for you to program. The peripheral I/O is an entire topic in itself, and because of its importance, Section 8.2 is dedicated to it.

The interrupt that is always used in any computer is the RESET. It runs the initialization routines needed to get the 6502 processor going and set up the Monitor or operating system. The first thing any 6502 RESET routine must do at power up is to initialize the stack and clear the D-flag:

```
COLD:    LDX   #$FF
         TXS
         CLD
         . . .
```

The $FF value clears the stack. Remember, push instructions like JSR will decrement the S-reg; pull instructions like RTS increment it. If the D-flag isn't forced clear, it may be the cause of strange bugs, making BCD calculations instead of binary ones. All RESET routines must have these three instructions.

Once these three instructions are done, you can write the RESET routine to do whatever your system needs. It must be programmed into ROM to be available at power up.

The third use for interrupts is program debugging. On the 6502, two interrupts are available for debug routines — the BRK and the NMI. The BRK is a software interrupt and the NMI is a hardware interrupt, like IRQ and RESET.

Some microcomputers use the NMI for debugging. The NMI is generated once each instruction by connecting it with a switch to a pin labeled RDY on the 6502. On interrupt, the NMI routine gives the programmer a Monitor to examine registers and memory without further interrupts. This feature isn't used on the Apple, so NMI could be used as a means of gaining control of the machine regardless of the actions taken by the current program.

Instead of using the NMI, the Apple Monitor uses the BRK to support debugging. To use BRK, you insert the op code of a BRK instruction into your routine at the point you want to examine. It is used much like the STOP statement in BASIC. When the BRK is executed, it causes an unmaskable IRQ interrupt and sets the B-flag. The Apple Monitors, especially the Standard, recognize the B-flag in the IRQ routine and save all the registers before entering the Monitor. A very useful feature.

The interrupts: servicing hardware, handling initialization, and providing debugging breakpoints give you complete control over your computer.

Here's what happens when an interrupt occurs.

The processor will complete its current instruction before recognizing any interrupt. Depending on the interrupt, the processor will fetch one of three addresses from memory, see Table 3-10.

Table 3-10. Hardware Vector Addresses

Address	Vector
FFFA	Vector address low for NMI
FFFB	Vector address high for NMI
FFFC	Vector address low for RESET
FFFD	Vector address high for RESET
FFFE	Vector address low for IRQ and BRK
FFFF	Vector address high for IRQ and BRK

For an IRQ, the address is fetched from $FFFE.FFFF, if the I-flag is clear. Otherwise, the interrupt is ignored until the I-flag gets cleared by the program. A BRK instruction will also cause the address to be fetched from $FFFE.FFFF. The BRK is not inhibited by the I-flag; it sets the B-flag. Once fetched, the address is put into the PC after the processor saves the old PC and P-reg on the stack.

Similarly, an NMI allows the current instruction to complete. Then, it saves the PC and the P-reg on the stack. Finally, it reads the address of the NMI routine from the vector at $FFFA.FFFB into the PC. Nothing inhibits an NMI.

For a RESET, there is no procedure to save a current program. The RESET is intended to service power up. The address at $FFFC.FFFD must be in ROM as well as the routine it references. This address is simply fetched into the PC after several clock cycles in which the 6502

gets itself synchronized into the instruction fetch/execute cycle.

Here is how to use the BRK instruction. When used for debugging, as with the Apple Monitor, replace one of the instructions in your routine with a BRK. The BRK op code is $00. Then run the routine and you will get the Monitor prompt — "*" — when the break occurs. At that point, you can examine memory or register contents. The registers at the time the BRK occurred are saved by the BRK routine. You use ctrl/E to examine them.

When used as a software interrupt in your custom separating system, you must distinguish the BRK from the IRQ with the B-flag. If the B-flag is set, then a BRK instruction is the cause. At the end of your routine, you can return to the next instruction after the break. Just use the RTI instruction; it restores the P-reg and the PC.

There is one caution to observe when using the BRK instruction. When it occurs, the BRK bumps the PC by two locations, not one. So, you should follow the BRK with a NOP in your code. Then, the RTI will return you to the next instruction, following the NOP. If you don't make this allowance, the results can be disastrous!

You can trap the BRK from other IRQs with a routine like this:

```
IRQS:    PHP
         PLA              to get the P-reg
         AND   #$10       to isolate B-flag
         BNE   IRQS1      B-flag set?
         JSR   BREAKS      yes . . break routine
         JMP   IRQSX
IRQS1:   . . .            no . . . valid IRQ
         . . .                handler here
         . . .
IRQSX:   RTI              meanwhile, back at the ranch
                           . . .
```

where the BREAKS routine deals with the software BRK instruction interrupt, and IRQS1 deals with IRQs from hardware sources. Both kinds of interrupts are ended with an RTI.

3.4.2 The Monitor Interrupts

In the three Monitor versions — Standard, Autostart, and IIe — interrupts are little used. The exception is the RESET, which varies

significantly from model to model. The NMI and IRQ are handled essentially the same way in all three.

The NMI interrupt is exactly the same in all three Monitors. The vector at $FFFA.FFFB points to $03FB for any user JMP instruction. Usually, the RESET routine sets this as a JMP $FF69 to give you the Monitor command interpreter if an NMI should occur without your providing any routine of your own. So, to use the NMI, you must put a JMP to your own interrupt service routine at $03FB instead.

The IRQ interrupt is vectored from $FFFE.FFFF to the Monitor's IRQ routine. The IRQ routine saves the A-reg at $45 in Page Zero and tests the B-flag to see if a BRK has occurred. If not, then it jumps at $03FE, the address of your IRQ routine.

If the B-flag is set, the routine saves all registers at $46.49 and $3A.3B in Page Zero. Then it cancels the interrupt by pulling the P-reg, and PC from the stack. After that, it runs the break routine.

The break routine displays the PC and current instruction where the BRK occurred. Then it displays the registers from the $46.49 locations. Finally, it jumps to the Monitor command interpreter at $FF65. In the Standard Monitor, the break routine must always be run; in later Monitors the break routine may be replaced. To use your own break routine, replace the $FA59 address of the OLDBRK found at $03F0. Remember, the old Standard Monitor does not have this replacement feature.

The IRQ/BRK logic of the Monitor is given in Fig. 3-13 for the Autostart version. The Standard version does not have BRKV at $03F0; control falls through to OLDBRK at $FA59 in all cases. The IIe uses the same logic as does Autostart.

The RESET routine is run whenever the Apple II is powered up or when you use the RESET key. On the Standard Monitor, the RESET routine simply initialized the built-in terminal. After the terminal was initialized, it ran the command interpreter, which gave an audible beep and a "*" prompt on the screen. A ctrl/B was used to cold start the native BASIC; ctrl/C to warm start. If you wanted to bootstrap a disk in Slot Six, you had to type

<div align="center">6ctrl/P</div>

Also on the Standard Monitor, RESET always works the same way, regardless of whether it comes from a power up or a keypress.
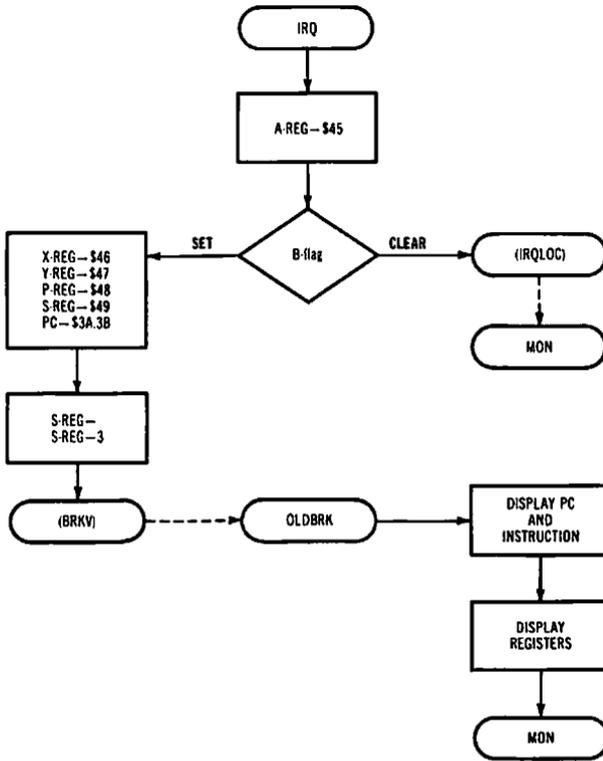
Fig. 3-13. IRQ/BRK logic of Autostart Monitor.

Next came the Autostart Monitor. RESET is more involved in Autostart because it functions differently for power up. Also the keyboard RESETs provide several features.

All RESETs are vectored from $FFFC.FFFD. Like the others, Autostart clears the D-flag, initializes the built-in terminal, and defaults its parameters. Some initialization added to Autostart includes setting annunciators on the games socket and clearing the keyboard strobe.

Then, instead of going to the Monitor command interpreter, Autostart does the following.

First, it tests the contents of $03F4. This is called PWREDUP and it tells the routine if this is a power up or not. If it is a power up, it has a random value. If it is a keypress, then the previous RESET will have

set it to the EOR of $A5 and the high byte of the BASIC warm start address. Since this is $E003, the EOR is $45.

If it is a keypress, then it tests for the BASIC warm-start vector, SOFTEV, at $03F2.03F3. It should be $E003. If not, it sets it to $E003 and does a cold start to the BASIC at $E000. If it is a warm start, it uses the SOFTEV as an indirect jump to make the warm start.

If it is a power up, it prints the APPLE][on the screen, initializes the Page Three vectors — BRKV, SOFTEV, and PWREDUP — and searches the slots for a disk controller card. It will bootstrap the card it finds in the largest slot number. If no card is found, it does a cold start of BASIC, using the $E000 value it put into SOFTEV.

The three Page Three Locations are new with Autostart. SOFTEV at $03F2.03F3 is set to $E000 to cold start BASIC, and to $E003 to warm start it. In either case, the high byte value of $E0 is EORed with $A5 to make a $45 for PWREDUP at $03F4. Then, if the earlier test of PWREDUP fails, a disk bootstrap or cold start can be chosen.

One consequence of this logic is that repeated keyboard RESETS will force a cold start, ignoring the disk card. It is the only way to stop the disk drive from running forever if you don't have a valid bootstrap disk mounted.

Another consequence is the ability of the software to grab the warm-start vector, SOFTEV, for itself. DOS does this. When boot-strapped, DOS puts $9DBF into SOFTEV and $38 into PWREDUP, replacing the RESET routine's values. Then, if RESET is pressed at the keyboard, the test for PWREDUP recognizes a keyboard RESET. Then, because SOFTEV does not have the BASIC cold-start address of $E000, a warm start is done by an indirect jump to SOFTEV. This way, DOS warm start at $9DBF is run.

On the IIe version, the logic is much the same as Autostart's RESET. The major difference is the addition of a forced cold-start test.

In Autostart, if Page Three got clobbered, or the warm entry routine went wrong somewhere, the Apple would hang up. The only way to recover is to switch the power off; repeated RESETs at the keyboard won't reach BASIC or the Monitor. Unfortunately, your program is erased when RAM is powered off, and the lifetime of the rocker power switch is shortened.

To prevent powering off to recover from a crash, the IIe Monitor has another test in the RESET routine. Before it tests the PWREDUP byte, it examines the OPEN-APPLE key. If you press this key during RESET, it alters PWREDUP so it forces a cold start. This overcomes the problem nicely.

In addition to erasing PWREDUP, the IIe RESET routine deliberately erases locations in each page of RAM, throughout memory. The logic of the Apple IIe RESET is given in Fig. 3-14.
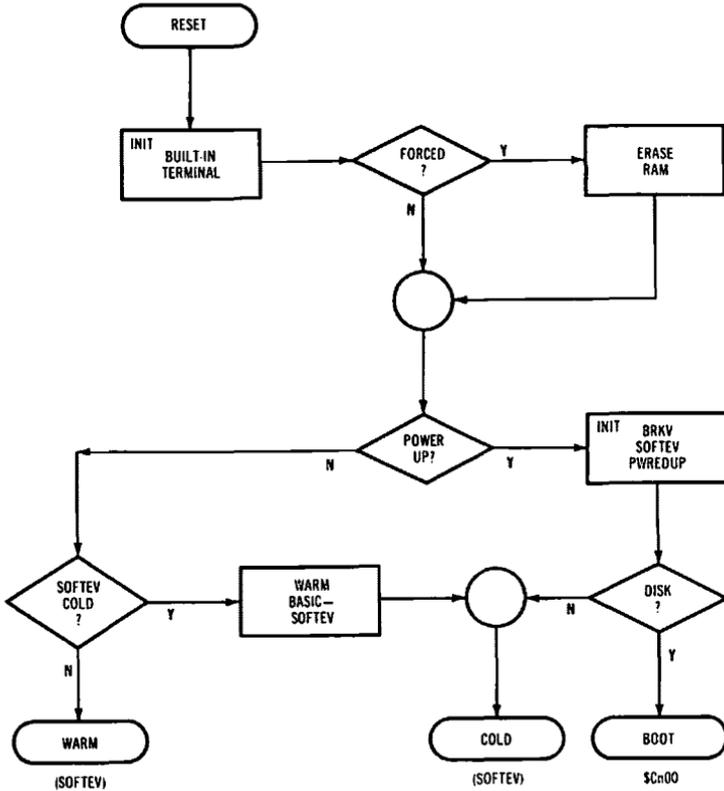
Fig. 3-14. Reset logic of Apple IIe Monitor.

3.5 PARAMETERS

3.5.1 Passing by Value

In order to get a routine to do something specific, you may have to give it values called parameters. Often, the routine will have a result to give when it returns; such a result is also called a parameter. Some routines have no parameters; they just do a specific job using hardware. But, most routines you use have one or more parameters.

A simple way to pass parameters to and from a routine is with the registers. Often, only one byte is passed in the A-reg this way. The advantage that registers have is that the parameter is where the routine can access it quickly.

One example of passing a value in the A-reg is the output call COUT at $FDED. By putting the code of the character to be output into the A-reg, you can call COUT using a JSR. It sends the character, still in the A-reg, to the routine of the current output device.

Another example of passing a value in the A-reg is the routine to copy a given length of memory from HERE to THERE. In that routine, the X-reg was used as the counter during the copy, but was not set by the routine itself. Instead, it required the caller to set the X-reg for it, so it will copy the number of bytes that it is told to copy each time. The number of bytes is passed to that routine as a parameter — as a value in the X-reg.

Other routines you might write and use that use the registers to pass parameters include code converters, searchers, soft-switch setups, graphics displays, and cursor control routines.

Instead of putting parameters into registers, you can keep them in memory. The routine being called can use the value from the agreed-upon location and return it there as well. Parameters in memory are a little more permanent than registers. Remember, registers are used heavily, so you can't keep parameters there for several calls.

The Page Zero locations are ideal for working addresses. Routines can share access to them; increment and decrement routines given earlier are an example of using Page Zero values as parameters. Arithmetic routines can use Page Zero locations for long registers that can contain several bytes each. Special Page Zero addressing modes give fast access.

Often, Page Three is used in the Apple to keep parameters, especially by the system. By storing keystroke characters in Page Two, keyboard input routines pass them to other routines that scan them for particular usage, like a BASIC parser. The Page Two block used as a parameter storage area like this is called a *buffer*. DOS keeps three file buffers at $9600.9CFF for its file management routines.

Using memory to store parameters is a good way to deal with large amounts of data.

When you want to pass register values without destroying them, you need the stack. This might not happen often, but when it does, you can make the routine "bulletproof" with this method.

Within the routine being called, the first thing you do is push all the registers onto the stack:

```
PHP
PHA
TXA
PHA
TYA
PHA
```

If it is an interrupt, the PHP is already done; don't include it here.

Next, you set the X-reg to point into the stack. This lets you pick up any register value of the caller in your A-reg by choosing the appropriate Page One Address:

```
                  TSX
LDA  $0101,X  to fetch Y-reg
LDA  $0102,X  to fetch X-reg
LDA  $0103,X  to fetch A-reg
LDA  $0104,X  to fetch P-reg
LDA  $0105,X  to fetch (return-1)-low
LDA  $0106,X  to fetch (return-1)-high
```

You can return to the caller after pulling his registers back from the stack:

```
PLA
TAY
PLA
TAX
PLA
PLP
RTS
```

Instead of the PLP/RTS pair, an RTI can be used. Do this to return from interrupt calls; elsewhere it confuses the reader.

An easy way to pass parameter values to a routine is by listing them immediately in the program itself. This is a common method because of the casualness you have in calling. The routine itself needs to be well written in order to set itself up to reach the parameter values, but

this is often worthwhile. For instance, when you want to send a command string to DOS, or a display to the screen. The proper routines in machine language can make these tasks easier to program than their BASIC equivalents.

Passing values immediately in the calling program is good for one direction, caller to subroutine. Don't use it the other way. You *could* invent a method of returning parameters to the caller, but it's rather useless; you could just use a simple mailbox to greater effect and programming ease. Use this method one-way only.

Here's how such a call would be made. Suppose you had a routine that positioned the screen's text cursor from row and column numbers passed this way.

```
JSR   GOTOXY   ;sets cursor
DFB   26       ;column
DFB   11       ;row
next instr.
```

The subroutine GOTOXY has only one way of finding its parameters, by getting the return address from the stack. To do this, it needs a Page Zero pair of locations to store it as the pointer to the caller's code.

```
GOTOXY:  TSX               ;look at stack
         LDA   $0101,X    ;return addr-lo
         STA   A1         ;in Page Zero
         LDA   $0102,X    ;return addr-hi
         STA   A1+1       ;in Page Zero
```

At this point, the Page Zero pointer, A1, has the location of the calling routine at the second address byte of the JSR. The row and column numbers are in the following two bytes. These can be accessed by incrementing a Y-reg or by incrementing the A1 pointer itself. The second method turns out to be the best.

```
         LDX   #0
         INC   A1
         BNE   GOTO1
         INC   A1+1
GOTO1:   LDA   (A1,X)    ;column
```

```
              STA   CH        ;set horiz. cursor
              INC   A1
              BNE   GOTO2
              INC   A1 + 1
     GOTO2:   LDA   (A1,X)    ;row
              STA   CV        ;set vert. cursor
              JSR   VTAB      ;moves cursor
```

Mission accomplished; the cursor has been moved according to the two parameters. Next, the problem is how to return.

The return address on the stack will cause the RTS instruction to continue execution with our parameters. Somehow, it must reach the code *following* the parameters instead of landing into them. If the address on the stack was replaced with the present contents of A1, that would work because A1 points to the last parameter byte. The next byte in the caller is the next instruction op code, so moving A1 to the stack and doing an RTS will work fine.

Alternately, the A1 pointer can be bumped once again to provide an address for an indirect jump. The stack will have to be cleaned up by pulling it twice to remove the JSR's return, but that's simple enough. The advantage of using the indirect jump method is that it can be simple.

```
              PLA             ;get rid of
              PLA             ;return address
              INC   A1
              BNE   GOTO3
              INC   A1 + 1
     GOTO3:   JMP   (A1)      ;return
```

If the parameter list is long, use a subroutine to bump the pointer.

The trick to using this method of picking up parameter values is to pass each one in order of use, preferably accessing each one from the list once and only once. A little subroutine to bump the pointer and fetch the byte is quite handy when working with long or complex strings.

```
     PICKUP:   INC   A1        ;bump addr-lo
               BNE   PICKUP1   ;new page?
               INC   A1 + 1    ; yes-bump addr-hi
```

```
PICKUP1:  LDX  #0        ; no-continue
          LDA  (A1,X)    ;pickup new byte
          RTS
```

The special techniques for handling strings are not difficult; you can find them in Chapter Six.

This powerful method is used in Chapter Six to manage strings, in Chapter Seven to generate DOS commands, and it is the way BASIC reads your commands and statements.

3.5.2 Passing by Reference

After passing parameters directly, by value, the most common method used is passing by *reference*. Instead of the actual value, this method gives the routine the address where its parameters may be found.

Passing by reference can be done with the registers. The two bytes of an address fit into two registers, then the called routine can store them into Page Zero with its first two instructions. This way, it has a Page Zero pointer all set to pick up parameters using indirect addressing:

```
SUBR:   STX  A1        address-low
        STA  A1+1      address-high
        LDY  #0        init. pointer
SUBR1:  LDA  (A1),Y    fetch parameter byte
        . . .
```

Here, up to a page of parameters are passed. The caller simply puts the address of the parameters into the X-reg (low) and A-reg (high) to give them to the subroutine.

Occasionally, the parameters may be in Page Zero. For example, arithmetic routines using several bytes in Page Zero to reduce access time. By passing the location within Page Zero in the X-reg, the subroutine can use Zero-Page-X addressing mode to reach each byte rapidly:

```
LDA  $00,X   reads zeroth byte
LDA  $01,X   reads first byte
LDA  $02,X   reads second byte
etc.
```

The Zero-Page-X mode is fast and has all the arithmetic and logic instructions, making it ideal for fast arithmetic.

Yet another way of passing by reference in a register is to indicate just which byte in a given chunk of memory the parameter of interest lives. Suppose that the address of a parameter buffer is being passed in the X-reg and the A-reg, like the first example. The exact byte within that buffer could also be passed as well, using the Y-reg:

```
SUBR    STX   A1        addr-lo
        STA   A1 + 1    addr-hi
SUBR1   LDA   (A1),Y    get the Y-th byte
```

If the Y-reg is used to pass the length, it can be decremented to fetch all bytes from the buffer.

The address only is passed in the registers, so there is enough capacity. Parameters can be passed in both directions — caller to subroutine and subroutine to caller.

The drawback to the register method is that you can only pass one address per call. Alternately, you can pass by reference using memory.

Page Zero is heavily used, especially by the system in the Monitor, BASIC, and DOS, to pass parameters by reference. These Page Zero references are called *pointers*. Each pointer identifies a different parameter. Routines use indirect addressing to reach those parameters, especially indirect indexing with the Y-reg scanning several bytes.

Some system pointers are kept in Page Three as well. And DOS, BASIC, and the Monitor all have pointers tucked away within their own memory areas. These are often copied to Page Zero when calling their routines.

Many applications use the stack to press parameters by reference; the addresses are pushed and pulled on the stack. BASICs do this; the Pascal system does this extensively. It is a very elegant method, so it is suited to well-defined software designs.

A routine passing, say, two parameters to a subroutine on the stack would call it like this:

```
LDA   #PARM1   high byte
PHA
LDA   #PARM1   low byte
PHA
```

```
LDA   #PARM2   high byte
PHA
LDA   #PARM2   low byte
PHA
JSR   SMART
. . .
```

The subroutine must pickup the addresses from the stack and put them in Page Zero for access:

```
SMART   TSX                  get stack pointer
        LDA   $0103,X   Parm2-low
        STA   A2
        LDA   $0104,X   Parm2-high
        STA   A2 + 1
        LDA   $0105,X   Parm1-low
        STA   A1
        LDA   $0106,X   Parm1-high
        STA   A1 + 1
```

Then, it must clean up the stack, allowing for a return:

```
        LDA   $0102,X   return-high
        STA   $0106,X
        LDA   $0101,X   return-low
        STA   $0105,X
        PLA
        PLA
        PLA
        PLA
```

The result is to have the parameters in Page Zero pointers that are for the routine's use: A1 and A2 used here. The return address has been moved and the stack pointer pulled four times — two parameters of two bytes each. This will let you do an RTS normally.

Sometimes you have long strings or other parameters that you want to pass to a routine from several points in your program. Passing immediately by value would waste a lot of space, because the string would have to be repeated each time. If the parameter was a choice of buffers, it would be difficult to pass other than by reference. Such

parameters are passed by stack reference, just described. Or, an easier method for the calling routine can pass by reference, immediately. This works much like passing values immediately:

```
        JSR   SUBS
        DW    MESSAG
```

The DW puts the address in low byte, high byte order. The SUBS routine must pickup the parameter as in the value passing method:

```
SUBS    TSX
        LDA   $0101,X   addr-lo
        STA   A1
        LDA   $0102,X   addr-hi
        STA   A1 + 1
        JSR   PICKUP    get parm-lo
        STA   A2
        JSR   PICKUP    get parm-hi
        STA   A2 + 1
        JSR   PICKUP    bump to next instr
        . . .
```

At this point, the parameter address is in A2 and the next instruction address in A1. To return, get rid of the stack address and jump at the A1 address instead:

```
        . . .
        PLA             clean up stack
        PLA
        JMP (A1)        next instruction
```

Remember the PICKUP routine. It crawls through the caller's code, using A1, incrementing and fetching bytes.

You can pass several parameters this way. Here, A2 was used for the first; you will need other Page Zero pointers for successive parameters.

3.5.3 Modularity

Routines have three major parts: an initialization, an algorithm, and an exit. The initialization part fetches any parameters, saves registers, initializes loop counters, and anything else the algorithm needs to do its job for the caller. The algorithm is the function of the routine; it uses one or more of the structures developed in Section 3.3. And, the exit returns parameters, restores registers, cleans up the stack, and anything else needed to make a normal return to the caller. By designing the simplest routine with these three parts, you can write the easiest one to maintain.

To keep routines simple and easy to use, arrange them so that the initialization is at the beginning of the memory with the first location containing the first instruction. Don't jump into the middle of a routine; if you must, use a JMP as the first instruction.

Algorithms are best if they use only one structure. If you must make a compound structure, explain it in your comments. Usually, this involves nesting one within the other, like having a sequence within a DO-UNTIL instead of JSRing to the sequence. Some complex structures used in logic and arithmetic are given later, in Section 3.6. Otherwise, the simple structures will handle most programming needs.

The exit should be from the last instruction in the routine. Literal data and buffer space needed by the routine can follow the exit. Where there is a lot of detail in returning, tasks should proceed in the inverse order of their corresponding setup. For instance, if the initialization part saved all registers on the stack, then loaded a parameter from memory, the return part should save the parameter to memory *before* restoring all registers from the stack. The last instruction is usually an RTS; it may be a JMP or an RTI, however.

When you have several routines to use, the safest way to manage them is when they are simply written, each with only one entry point and only one exit point. Often, branches and jumps will be needed within the routine to meet this condition. Although that means some extra programming, the time saved later in using such routines more than pays for it.

Routines with clearly defined initialization, algorithm, and return parts like this are called *modular*. They can be easily used as modules with simple call sequences by other routines.

One special class of modular routines occurs where they must be capable of interrupting themselves. An IRQ routine, for instance, may

service hardware that interrupts often enough that there will be cases of interrupts happening before the previous one is finished. If each one *must* be serviced, the interrupt routine must be capable of interrupting itself. Such a class of routines is called *re-entrant*.

In a re-entrant routine, all registers used within it must be stacked. Only then can further interrupts be allowed by clearing the I-flag:

```
IRQS    PHA
        TXA
        PHA
        TYA
        PHA
        CLI
        . . .
```

Once this is done, the body of the routine may be further interrupted; its registers are safe. When the service is complete, return by reversing these steps:

```
        . . .
        SEI
        PLA
        TAY
        PLA
        TAX
        PLA
        RTI
```

If you use any Page Zero pointers, they must be constant. You can't alter them from within. Same goes for writing to other memory locations. Any memory you want to write must be saved before the CLI and restored after the SEI. You will probably be pressed for time, so these extra steps will hurt. Keep memory write needs to a minimum; you must write for speed.

Never clear the I-flag in an interrupt routine that is not re-entrant. Always set the I-flag in re-entrant routines before restoring and returning.

3.6 ARITHMETIC

3.6.1 Number Bases

The key to understanding arithmetic algorithms is in the way numbers are stored and represented. This way is called *positional notation;* it underlies the way you handle format, make calculations and convert numbers between different bases. You can already do these things on paper, so reviewing the concepts will give you the understanding needed to do them with your Apple.

Consider a whole number expressed in base ten — a decimal number. For an example of 39201, you know just what number these five symbols represent because each symbol has a different weight: the 1 is simply one, the 3 is thirty thousand. So, the number appears to us as a sum of five terms:

$$3 \times 10,000 \text{ plus}$$
$$9 \times 1,000 \text{ plus}$$
$$2 \times 100 \text{ plus}$$
$$0 \times 10 \text{ plus}$$
$$1 \times 1$$

Each digit, 0 to 9, therefore represents itself times a multiple of ten. The multiple is given by the *position* of the digit in the number; that's why it's called positional notation.

Positional notation acts as a neat way to keep numbers on paper because it's fast and easy to read and write. In computers, it is efficient since the position of the digit represents the multiple of ten without any extra storage to carry that information. It is even more efficient if it represents multiples of two instead of ten.

By changing the base of the positional notation of numbers from ten to two, memory can store larger numbers in the same space. But we have to nail down our concepts of positional notation a little tighter in order to use it in a base other than our familiar ten. Then converting between base ten and base two for a number may be needed. Fortunately, base two is easy to understand and program even if it is awkward for us humans to read and write.

A number represented in base two breaks down the same way that it would in base ten. There are only two digits possible in each position, 0 or 1, instead of the ten we had. Each position has a weight of a power of two:

$$1 \times 64 \text{ plus}$$
$$0 \times 32 \text{ plus}$$
$$1 \times 16 \text{ plus}$$
$$1 \times 8 \text{ plus}$$
$$0 \times 4 \text{ plus}$$
$$0 \times 2 \text{ plus}$$
$$1 \times 1$$

This number is 1011001 in binary or base two form. Again, the greatest weight is attached to the leftmost digit; the rightmost digit has a weight of one. If you broke the same number down in base ten, you would get

$$8 \times 10 \text{ plus}$$
$$9 \times 1$$

as its decimal representation. Positionally, you would write it as 89 — eighty-nine. The important thing is that 1011001(base 2) *equals* 89(base10) because they each represent the same number, eighty-nine.

Regardless of the base, a number is represented by positional notation using a set of *digits*: ten digits for decimal, two digits for binary or any other base number of digits. Each position contributes a term to the number consisting of the digit multiplied by the weight of its position. This weight is the base raised to the power of the position. So, eighty-nine in base two can be expanded as

$$1011001 =$$
$$(1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

by writing 64 as 2^6, 32 as 2^5, etc. Note that 2^0 equals one.

The rightmost position is called the *least significant* position. For whole numbers like we have here, it is also the *units* position because its position is zero and its weight is always one. The weight of any position is given by

$$(\text{base})^{\text{position}}$$

so regardless of the base,

$$(\text{base})^0 = 1$$

always.

In the Apple, a byte of eight bits often represents a number in binary notation. With eight bits, there are eight positions from the *least significant bit* on the right to the *most significant bit* on the left. The least significant bit has position *zero* while the most significant bit has position *seven*. The entire byte can represent 256 different numbers. For example, the number eighty-nine would be represented as 01011001. The most significant bit is in position 7 and has a weight of 2^7 or sixty-four. The next bit in position 6 has a weight of 2^6 or thirty-two. Similarly, the next-to-least significant bit in position 1 has a weight of 2^1 or two. The least significant bit has the lowest weight, 2^0 or one.

Compare this with the way we represent numbers in decimal notation. A number like 39201 is short for

$$(3 \times 10^4) + (9 \times 10^3) + (2 \times 10^2) + (0 \times 10^1) + (1 \times 10^0)$$

where the most significant digit is 3 with a weight of 10,000 and the least significant digit is 1 with a weight of one.

This shows how positional notation works: each digit chosen from a base number of digits and weighted by that base raised to the power of its position.

Let's look at one more base to be sure of how positional notation works. Let's look at *base sixteen.*

Base sixteen positional notation is what we use as *hex* notation. It is based on sixteen digits — 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Each digit has a successive counting value; we just added six to our familiar ten. Now, each position will have a weight of sixteen raised to the power of the position. For example, if 79 is a hex number, then

$$\$79 = (7 \times 16^1) + (9 \times 16^0) = 112 + 9 = 121$$

expands it and allows us to express it in base ten. Or,

$$\$7F = (7 \times 16^1) + (15 \times 16^0) = 112 + 15 = 127$$

expands the hex number 7F in decimal form. A larger number works the same way:

$$\$F941 = (15 \times 16^3) + (9 \times 16^2) + (4 \times 16^1) + (1 \times 16^0)$$
$$= 15 \times 4096 + 9 \times 256 + 4 \times 16 + 1 \times 1$$
$$= 61440 + 2304 + 64 + 1 = 63809$$

It's just more work to convert to decimal notation.

The secret to converting from hex to decimal then is just to expand the hex number using decimal notation and then use decimal arithmetic to reduce the expression.

Going the other way — converting decimal to hex — is trickier because you still want to use decimal arithmetic. Take the decimal number and divide it by sixteen. The remainder is the least significant digit in base 16. Repeat until the quotient is exhausted, using the remainder each time for the next significant digit. As an example, here is the number we just saw:

$$
\begin{array}{ll}
16\sqrt{63890} & 1 \\
16\sqrt{3988} & 4 \\
16\sqrt{249} & 9 \\
16\sqrt{15} & F \\
0 &
\end{array}
$$

Remember to write remainders in hex notation: 10 as A, 11 as B, 12 as C, 13 as D, 14 as E, and 15 as F. From the example, 63809 = $F941.

Binary numbers are usually represented as hex numbers because hex is more compact and easier to read. Each hex digit is four bits in size and can be converted directly. To convert, partition a hex number into digits or a binary number into four-bit *nibbles* (sometimes spelled *nybbles*) and look up the equivalent:

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

The 6502 processor in the Apple has a feature that allows you to keep numbers in either binary or decimal. Normally, we use binary

numbers and express them in hex notation, but if you want to do your arithmetic with decimal numbers instead, you can. The decimal numbers are represented in a special way called *Binary Coded Decimal* or BCD.

Binary Coded Decimal is the way calculators store numbers. The way that you read binary numbers using hex notation, by representing each nibble as a hex digit, is the same way that you can read BCD numbers. Each BCD digit is kept in a separate nibble of four bytes:

DIGIT	BCD	DIGIT	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

It looks just like hex, but without any representation for ten to fifteen, A through F, in the nibble. When told to work with BCD, the processor uses only these digits.

Each byte contains eight bits when binary representation is used. When BCD is used, each byte contains two decimal digits, representing numbers from 0 to 99. When the processor is instructed to add it carries after passing 9, instead of counting to $F as in hex before carrying. The result is a decimal representation that gives each byte a capacity of only 99 instead of the 255 ($FF).

3.6.2 Addition and Subtraction

To add numbers in the Apple, you use the ADC instruction. This instruction has a lot of addressing modes. This allows you to add memory with the A-reg together with that in the C-flag and then to have your sum waiting in the A-reg at the end of the instruction. The C-flag is useful in carrying from one addition step to the next; otherwise you must always do a CLC instruction before adding. For example:

```
CLC
LDA  #$25
ADC  #$13
```

results in $38 as the contents of the A-reg. And

```
        CLC
        LDA   #$27
        ADC   #$36
```

gives $5D in the A-reg. Addition carries automatically from one bit to the next; the C-flag is carried into the addition and it is set or cleared depending on the sum. In both the above, the C-flag is cleared, but

```
        CLC
        LDA   #$73
        ADC   #$95
```

would give you an A-reg of $08 with the C-flag set. The carry came from the most significant bit of the addition.

```
        $73 = 01110011
        $95 = 10010101
        sum = 00001000 + 1 carry
```

For an addition of *two one-byte* numbers like this, the carry set tells us that we have an *overflow* from the addition: the result won't fit in its space.

To learn more about single-byte addition, try the following:

```
TEST:   CLC
        LDA   ARG1
        ADC   ARG2
        STA   RESULT1
        BCC   TEST1
        LDA   #$FF        ;flag carry set
        JMP   TEST2
TEST1:  LDA   #0          ;flag carry clear
TEST2:  STA   RESULT2
        JMP   $FF69       ;monitor
```

Try different ARG1 and ARG2 contents. Look at the sums in RESULT1 and the carrys in RESULT2. What happens if the SEC (set carry) instruction replaces the CLC? Try it and see.

You can use the *SBC instruction* (subtract with carry) like you use the ADC, but with one important difference. With the ADC you initialized with a CLC; with the SBC you must initialize using the SEC. Here is an example of subtracting two one-byte numbers:

```
SEC
LDA   #$9D
SBC   #$89
```

The result is $14 in the A-reg and the carry flag remaining *set*. The A-reg gives the difference while the C-flag tells you if a *borrow* took place to complete the subtraction.

Let's see an example of a borrow. Subtracting $80 from $7F requires a borrow because $80 is greater than $7F. So, the routine

```
SEC
LDA   #$7F
SBC   #$80
```

results in $FF in the A-reg and the C-flag cleared. For a single-byte subtraction, this means we had an *underflow* and the result was negative. The byte only holds positive values for us (at least for now), so the clearing of the C-flag tells us that the subtraction has no answer.

When subtracting with only positive numbers, the result can be meaningless whenever you try to subtract a larger number from a smaller number. If you don't range test the numbers first, the clearing of the C-flag tells you the subtraction underflowed. If it remains set, then the answer is correct.

If you want to add larger numbers together, you use the C-flag to include the carry from lower significant bytes to higher significant bytes. For instance, add two numbers in address format (lo/hi) together:

```
CLC
LDA ADDR1
ADC ADDR2        ;add low bytes
STA ADDR3
LDA ADDR1 + 1
ADC ADDR2 + 1    ;add high bytes
STA ADDR3 + 1
```

```
            BCS OFLOW        ;trap sum overflow
            continue, sum
            is OK
```

The two addresses are picked up from ADDR1 and ADDR2 in a mailbox, added to supply ADDR3 in the mailbox as the sum, and the overflow case trapped by the BCS. Between adding the low-order bytes and the high-order bytes, the C-flag is legitimately set or cleared to carry from the least significant byte to the most significant byte. Whatever it is, it is added together with ADDR2 + 1 to the contents of the A-reg.

So, in multiple-byte addition, use the CLC instruction once and only once, at the beginning of the first addition.

For longer numbers, you may use indexing. You could move your arguments into Page Zero for arithmetic operations and use Zero-page-X addressing. Or, you could set pointers to them and use indirect indexed addressing with the Y-reg. If you index, use the CLC just before entering the loop; don't clear it each time from within the loop! Note that you can't use the CPX, CPY, and CMP instructions in your loop; they force the C-flag and interfere with the carrying. Test your loop using the Z-flag or N-flag.

The address convention we just used has bytes ordered with increasing significance: the higher the byte address, the greater its significance. Some math packages will use decreasing significance, putting the most significant byte at the lowest address. With multiple-byte numbers, you may keep them in either order. Examples given here will be in increasing significance so that the ideas and some code can be used with address calculations as well.

If you wanted sixteen-byte precision addition using Page Zero, this is what the loop would look like: let ARG1, ARG2, and RESULT have their usual meanings with sixteen bytes reserved for each.

```
    ADD:     CLC                       ;initial C-flag
             LDX  #$F0                 ;minus sixteen
    ADD1:    LDA  ARG1 + 16,X          ;zero-page-X mode
             ADC  ARG2 + 16,X
             STA  RESULT + 16,X
             INX
             BNE  ADD1                 ;$FF is byte 15
             test for overflow,
             etc.
```

The CPX can't be used with X = 0, 1, 2, . . . , 15 as you'd like to do, because it forces the C-flag. So, use X = − 16, − 15, − 14, . . . , − 1 to count forward instead and when X = 0 you can leave the loop. The $F0 acts like − 16 with Zero-Page-X addressing because the high address byte is always forced to zero. This trick won't work with other indexing modes because the address calculation will carry you into the next page.

There are other ways to do multiple-byte addition with a loop, but this is probably the simplest.

Subtracting numbers larger than one byte works just like the addition of large numbers. Set the C-flag before the first subtraction, then don't force it until the subtraction is finished. When a borrow occurs, the C-flag will be zero for the subtraction in the next most significant byte, giving a result one less than if the C-flag had been set. This is how it completes the borrow.

The example of subtracting two addresses:

```
SEC
LDA  ADDR1
SBC  ADDR2        ;subtract low bytes
STA  ADDR3
LDA  ADDR1 + 1
SBC  ADDR2 + 1    ;subtract high byte
STA  ADDR3 + 1
                  BCC to trap under-
                  flow continue,
                  difference is correct
```

follows that of the addition. Note that ADDR2 is subtracted from ADDR1; keep the addresses in order as subtraction does not commute.

The long loop example will work the same way as well. Replace the CLC with an SEC; the ADC with an SBC; and use the BCC to grab the underflow wherever BCS grabs the overflow and you have it.

Arithmetic can be done with BCD notation as easily as with binary. At the beginning of the addition or subtraction routine, set the D-flag with a SED instruction. At the end of the routine, clear it with a CLD instruction. This causes any ADC or SDC instruction to work in BCD arithmetic instead of binary. Apart from setting and clearing the D-flag, you don't write the routines any differently.

For example, if you have an addition routine called ADD that works now in binary you can write a short routine to call it:

```
ADDBCD:  SED        ;set Decimal
         JSR ADD    ;same routine
         CLD        ;clear Decimal
         RTS
```

Then, calling ADDBCD will do the same thing as ADD, but in BCD.

You only change the ADC and SBC actions. This means you do all your loop counts in hex, just as before. Only the data numbers, the ones you add and subtract with the carry, are in BCD.

Make sure you always have a CLD to finish off any BCD routine. If the D-flag remains set in ordinary routines, the Apple can behave very strangely!

3.6.3 Logic

To do multiplication and division on a 6502, as well as other tasks requiring bit-by-bit access, you need the ability to use the logical instructions. These are: ROL, ROR, AND, ORA, ASL, LSR, EOR, and BIT. There are a few simple ways in which these instructions are almost always used. These ways examine and manipulate bits, do multiplication and division of binary numbers, and let you work with *logical bytes* of sizes other than eight bits: four bit BCD digits, for example.

There are four instructions used for bit picking: AND, ORA, EOR, and BIT. They have many addressing modes although only a few will be cited here. With them, you can isolate bits from a byte for examination, set any one, or clear any one. They use *Boolean logic* to do these things. The AND results in each bit in the A-reg being set only if both the original bit and the corresponding bit in the memory byte are set; otherwise, AND results in a zero bit. The ORA results in each bit in the A-reg being cleared only if the original A-reg bit and the memory bit were both clear to begin with; otherwise the bit is set. With the EOR, the result bit is set only if the original A-reg bit and the memory bit are different; otherwise, they are the same — both set or both clear — causing the bit to be cleared.

One common use of the EOR is complementing the A-reg value. Complementing sets each byte that was clear and clears each byte that

was set. For instance, the complement of $EF is $10 because the first has only bit 4 set while the second has only bit 4 clear. The instruction

EOR #$FF

does that. Any bit that was set is the same as the bit in $FF, so it gets turned off. And, any bit that was clear is different than the same bit in $FF, so it gets turned on.

The BIT instruction is a lot like the CMP, except it does not alter the A-reg at all; just the P-reg. More about that later.

To illustrate the actions of these instructions, the immediate mode of addressing is shown where possible. Just remember, you can use any one you want.

The ORA instruction turns bits on in the A-reg. To set any given bit, use a value, called a mask, that has all bits clear except the one to be turned on. For instance,

ORA #$80

sets bit 7 without affecting any other bit in the A-reg. To set other bits, see Table 3-11.

Table 3-11. Setting Bits in A-reg

Bit	Instruction
0	ORA #$01
1	ORA #$02
2	ORA #$04
3	ORA #$08
4	ORA #$10
5	ORA #$20
6	ORA #$40
7	ORA #$80

The AND instruction turns bits off in the A-reg. To clear any given bit, use a mask that has all bits set except the one to be cleared. To clear bit 7, for example, use

AND #$7F

Table 3-12. Clearing Bits in A-reg

Bit	Instruction
0	AND #$FE
1	AND #$FD
2	AND #$FB
3	AND #$F7
4	AND #$EF
5	AND #$DF
6	AND #$BF
7	AND #$7F

To clear any other bit, see Table 3-12.

The EOR — Exclusive-OR — flags *mismatched bits*. Often, it is used to make a *checksum* of a block of data. The A-reg is set to $FF, all bits set, before EORing all bytes in the block together. The final value is called the checksum and is kept to compare against future checksums of the same data. The altering of just one bit in the entire block of data will cause the checksum to change, revealing that an error exists. Both DOS and Monitor tape routines use such checksum calculations and tests.

The BIT instruction is unique to the 6502. Mainly, it tests a memory location's value without altering the A-reg. You can keep a mask to be ANDed with various memory locations in the A-reg and test each location without destroying your mask each time. The AND operation performed by the BIT instruction only changes the Z-flag:

```
LDA  #MASK
BIT  MEMORY1
BEQ  MATCH1
BIT  MEMORY2
BEQ  MATCH2
. . .
```

Very useful for status flags testing in peripheral chips; see Chapter Eight.

Two other flags altered by the BIT instruction are the N-flag and the V-flag — bit 7 and bit 6 of the P-reg. These bits are simply copied from memory. In use, suppose you had a character in the A-reg you don't want disturbed. Now, suppose you wanted to look at several

peripheral chips to see which one is active. The active one has bit 7 set in its status register, so you just look at the chips' status registers:

```
BIT   STAT1
BMI   IRQS1
BIT   STAT2
BMI   IRQ2
. . .
```

In each test, if the device is on, it sets the N-flag when addressed from a BIT instruction. The A-reg remains unaltered.

A common use of the BIT instruction is in clearing the keyboard strobe:

```
GET    LDA   $C000    keyboard character
       BPL   GET      until keypress
       BIT   $C010    clear strobe
       AND   $#7F     make positive ASCII
       RTS
```

When the BIT instruction executes, the keyboard character is in the A-reg. The BIT does a read at $C010 without actually loading any register. Apple's keyboard hardware needs the read instruction at that address, but you don't need any register changed. The routine ends after turning off bit 7 of the character with the AND instruction.

To test bits 6 and 7 of a byte, use the BIT instruction. Use BPL and BMI for bit 7; BVC and BVS for bit 6. To test any bit, use the A-reg with AND

```
       LDA   #$04      Mask bit 2
       AND   MEMORY
       BNE   BITON
       BEQ   BITOFF
```

To test a pattern of bits, make up that pattern in a mask:

```
       LDA   #$16      Mask bits 1, 2, and 4
       AND   MEMORY
       BEQ   BITSOFF
```

The BNE here only tells you that at least one of those three bits was set. If you need an exact match, use the CMP instruction; a BEQ will branch if they match exactly.

Binary numbers can be multiplied by using 6502 instructions. To multiply a single byte by two, only one instruction is needed — the ASL (see Fig. 3-15). Called Arithmetic Shift Left, the ASL increases the position of each bit in the byte by one. Bit zero, the least significant bit, is replaced by a zero value. Bit one is replaced by the former value of bit zero, bit two is replaced by the former value of bit one, and so on. Bit seven is moved into the C-flag. The result is to multiply the byte by two, by shifting each bit one position left.
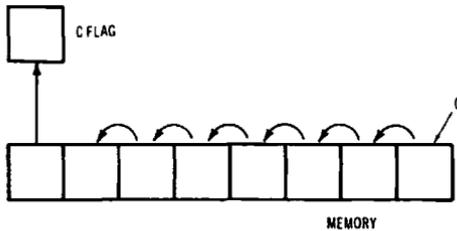
Fig. 3-15. The ASL instruction.

If the byte was $80 or greater, the high-order bit sets the C-flag, indicating an overflow from the multiplication. The resulting byte is always even, regardless of its original value.

The ASL operates on either memory or accumulator. Some assemblers want you to write ASL A for the accumulator mode; others want only ASL as the mnemonic with no explicit operand. Check your assembler's manual. In memory, ASL comes in Zero Page, Zero Page X, absolute, and indexed-X addressing modes. Aside from multiplication, you can use the ASL to examine bits one at a time from a byte, scanning left to right.

Another instruction will shift right instead of left. This is the LSR or Logical Shift Right (see Fig. 3-16) you can use to divide a byte by two or to pick up bits from a byte from right to left.

The LSR shifts each bit one position to the right. Bit seven is replaced by zero and the old contents of bit zero moves to the C-flag. Bit zero is replaced by bit one, bit one is replaced by bit two, and so on. In interpreting the result, if the C-flag is set then the original byte was odd, otherwise it was even. The resulting byte is always less than $80
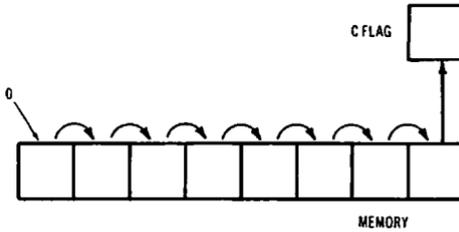
Fig. 3-16. The LSR instruction.

because bit 7 becomes zero. For the same reason, the N-flag is always zero after an LSR.

The shifts work with single bytes. LSR divides by two, results in zero bit 7 and old bit zero in the C-flag. ASL multiplies by two, results in zero bit 0 and old bit 7 in the C-flag.

If you have a large number to multiply or divide, or you want just to shift several bytes at a time to access their bits, you need another pair of instructions — the *rotates*. There are two: ROL for ROtate Left (Fig. 3-17) and ROR for ROtate Right (Fig. 3-18). They let you use the C-flag to carry the bit that you shifted out of a byte *into* the next byte. So, instead of forcing a zero bit into the other end, the C-flag content is used. A couple of examples should make this clear.
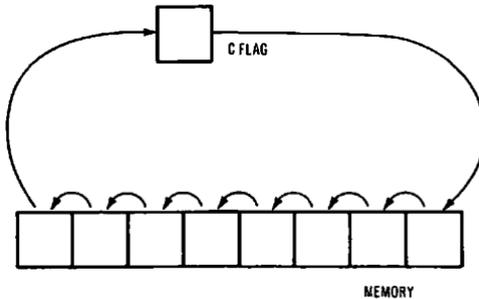
Fig. 3-17. The ROL instruction.

Suppose you rotate a byte containing 11000100 with the C-flag containing 1. Using the ROR instructions, the result would be 11100010 with C-flag = 0. The C-flag became bit 7, all bits were shifted right one position, and bit 0 became the C-flag. Another rotate right would give you 01110001 with C-flag = 0.

MEMORY
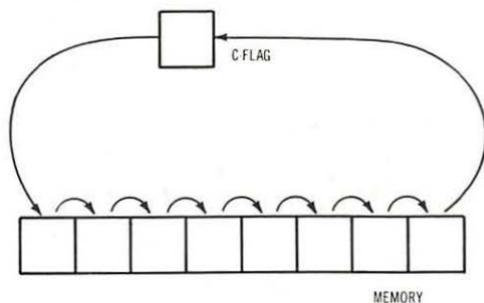
Fig. 3-18. The ROR instruction.

The rotate left reverses the action of rotating right. If byte was 00100001, C-flag of 1, then ROL would give you 01000011 with 0 in the C-flag.

Here's how you use the rotates in a loop to shift a field of several bytes one bit position at a time. The loop for shifting right in descending order of significance is

```
RIGHT:    LDX #0 – LENGTH
RIGHT1:   ROR FIELD + LENGTH,X
          DEX
          BNE RIGHT1
          RTS
```

where the bit to be rotated into FIELD is in the C-flag when called, and the bit from the other end is in the C-flag upon return. FIELD is the label for the bytes which must be in Page Zero. LENGTH is the number of bytes in FIELD. The ROR must be assembled as Zero-Page-X addressing mode. The X-reg points to the bytes because of the *wrap-around* address calculation that this mode makes. See multibyte addition above for a description of how this works.

Rotating the C-flag with a FIELD in the other direction, using ROL, is a little simpler.

```
LEFT:     LDX #LENGTH
LEFT1:    ROL FIELD – 1,X
          DEX
          BNE LEFT1
          RTS
```

The FIELD is shifted left in descending order of significance as well.
This method of rotation is used in multiplication and division
algorithms, in handling BCD digits, in binary/decimal conversion
routines, and in manipulating floating-point numbers.

3.6.4 Handling Numbers

Most numbers you deal with in programming are simple binary
natural numbers. They have fixed positional notation, they are in
binary, represented usually in hex notation, and they don't have any
sign ($+/-$); they are all positive.

Usually, binary natural numbers are small, taking only one or two
bytes to contain them. They are used for addresses and indexes in ma-
chine programs. For pointer manipulation, you need only to know
how these numbers work.

For applications, other numbers are needed. Integers, large and
small numbers are used. Sometimes, calculations must be made in
decimal. Three other systems are used for these purposes: integers,
BCD, and floating-point.

For binary natural numbers, you can use the increment and
decrement routines to bump Page Zero pointers. Then, if you need to
calculate an offset, you can add the addresses together like the addi-
tion examples show. Multiplication is needed to create your own table
lookup methods, so one is given in Example 3-1. By way of compari-
son, the divide routine in Example 3-2 works in a similar fashion. Play
with them first to be sure you know just what they can do for you.

The binary numbers described so far have all been *unsigned;* that is,
they are positive numbers only. If one bit in a number is reserved to
represent an algebraic sign — set for negative, clear for positive —
then you could use the number as an integer. You can, because the
6502 arithmetic supports signed number operations.

Numbers can be signed as base two integers. The sign bit is bit 7 in a
one-byte number. In a two-byte integer, the sign appears in bit 7 of the
most significant byte. This leaves fifteen bits for the size of the
integer. For one-byte integers, the size fits into seven bits. Integers
have sign and size; unsigned numbers have size only. To do this,
integers only have one-half the size of an unsigned number in the same
space. One-byte numbers can represent an unsigned number from 0 to
255 or it can represent an integer from -128 to $+127$. Similarly, two
bytes can hold unsigned numbers from 0 to 65,025 or integers from

Example 3-1.

```
SOURCE FILE: EXAMPLE 3.1
0000:              1  ******************************
0000:              2  * EXAMPLE 3.1                *
0000:              3  * INTEGER MULTIPLY ROUTINE   *
0000:              4  *                            *
0000:              5  * USES $50.55 IN PAGE ZERO   *
0000:              6  * LEAVES Y-REG UNCHANGED      *
0000:              7  * TO CALL:                   *
0000:              8  *    $50.51 <--- X VALUE      *
0000:              9  *    $52.53 <--- B CONSTANT   *
0000:             10  *    $54.55 <--- M MODULUS    *
0000:             11  *        JSR MULT             *
0000:             12  *    $$50.53 ---> Y RESULT    *
0000:             13  * WHERE                       *
0000:             14  *      Y = M*X + B            *
0000:             15  * ALL NUMBERS IN (LO,HI) ORDER*
0000:             16  * AND UNSIGNED.               *
0000:             17  ******************************
0000:             18  *
0000:             19  *
----- NEXT OBJECT FILE NAME IS EXAMPLE 3.1.OBJ0

1000:             20          ORG   $1000      FOR TEST
1000:             21  *
1000:             22  *
1000:A2 10        23  MULT    LDX   #16
1002:18           24          CLC
1003:26 52        25          ROL   $52
1005:26 53        26          ROL   $53        INITIALIZE
CARRY
1007:66 53        27  MULT1   ROR   $53
1009:66 52        28          ROR   $52        SHIFT BX IN
TO CARRY
100B:66 51        29          ROR   $51
100D:66 50        30          ROR   $50
100F:90 0D        31          BCC   MULT2
1011:             32  * A BIT FROM X-VALU IS DETECTED.
1011:             33  * ADD MODULUS AS THE PARTIAL
1011:             34  * PRODUCT TO B.
1011:18           35          CLC
1012:A5 52        36          LDA   $52
1014:65 54        37          ADC   $54
1016:85 52        38          STA   $52
1018:A5 53        39          LDA   $53
101A:65 55        40          ADC   $55
101C:85 53        41          STA   $53
101E:             42  * NEXT BIT SHIFT
101E:CA           43  MULT2   DEX
101F:10 E6        44          BPL   MULT1
1021:60           45          RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

−32,768 to +32,767. The two-byte integer is what you find in most BASICs like Apple's INTEGER and in Applesoft.

Example 3-2.

```
SOURCE FILE: EXAMPLE 3.2
0000:              1 *******************************
0000:              2 * EXAMPLE 3.2                 *
0000:              3 * INTEGER DIVIDE ROUTINE      *
0000:              4 *                             *
0000:              5 * USES $50.55 IN PAGE ZERO    *
0000:              6 * LEAVES Y-REG UNCHANGED      *
0000:              7 * TO CALL:                    *
0000:              8 *    $50.51 <--- NUMBER       *
0000:              9 *       $53 <--- DIVISOR      *
0000:             10 *       JSR DIVID             *
0000:             11 *    $50.51 <--- QUOTIENT     *
0000:             12 *       $52 <--- REMAINDER    *
0000:             13 * ALL NUMBERS IN (LO,HI) ORDER*
0000:             14 * AND UNSIGNED.               *
0000:             15 *******************************
0000:             16 *
0000:             17 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 3.2.OBJ0

1000:             18        ORG  $1000      FOR TEST
1000:             19 *
1000:             20 *
1000:A2 10        21 DIVID  LDX  #16
1002:A9 00        22        LDA  #0
1004:85 52        23        STA  $52
1006:18           24        CLC
1007:26 50        25 DIVID1 ROL  $50        NUMBER-LO
1009:26 51        26        ROL  $51        NUMBER-HI
100B:26 52        27        ROL  $52        REMAINDER
100D:CA           28        DEX
100E:30 0B        29        BMI  DIVID2     WHEN FINISH
ED
1010:A5 52        30        LDA  $52        RESIDUE
1012:38           31        SEC
1013:E5 53        32        SBC  $53        TRIAL SUBTR
ACT
1015:90 F0        33        BCC  DIVID1     IF IT BORRO
WS
1017:85 52        34        STA  $52        POST IF SUB
TRACT OK
1019:B0 EC        35        BCS  DIVID1     ALWAYS
101B:18           36 DIVID2 CLC
101C:66 52        37        ROR  $52        NORMALIZE R
EMAINDER
101E:60           38        RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

With integers, the N-flag has meaning. It flags the result of an instruction as having the sign bit, bit 7, set or cleared. If set, then the N-flag is also set and indicates a negative result. If clear, the N-flag is cleared to indicate a positive result. The BMI and BPL instructions then make sign testing easy.

When doing addition or subtraction, either can result in underflow or overflow. The single byte or the high-order byte must be tested for overflow or underflow as in the unsigned case, but the C-flag won't work. The problem with the C-flag is that it detects overflow from bit 7 only. With signed numbers, overflow occurs from bit 6; it overflows into sign bit 7 and destroys the sign. A special flag is provided so you can detect overflow from bit 6; it is the V-flag. Test the V-flag with a BVS branching to your error handler immediately after adding or subtracting signed numbers. You still use CLC before adding, and SEC before subtracting, but always test for the V-flag set to trap overflow or underflow, regardless of the operation.

The C-flag works the same with signed numbers as far as carrying arithmetic from low-order bytes to high-order bytes. When numbers are unsigned, the C-flag tells you if underflow or overflow occurred. When numbers are signed, the V-flag tells you; the C-flag does not. The branches on error are: BCS for unsigned addition, BCC for unsigned subtraction, and BVS for signed addition or subtraction.

Unsigned arithmetic can be binary or BCD; signed arithmetic is binary only, no BCD. Signed numbers are used in integer arithmetic in BASIC, store compactly, and use the V-flag for overflow detection. This is summarized for you in Table 3-13.

Table 3-13. Arithmetic Flags

Flag	Addition	Subtraction
	Must be cleared first Carries from byte to byte Unsigned overflow flagged by a 1	Must be set first Borrows from byte to byte Unsigned underflow flagged by a 0
C		
V	Signed overflow flagged by a 1	Signed underflow flagged by a 0

There are two areas of application where you will find BCD format numbers useful and even preferable to the usual binary format. One is digital hardware. Many devices come with BCD outputs that are converted to seven-segment displays. By picking up the four lines per digit from the gadget, you can interface to the Apple and work with its output using BCD software. Another place where you want to use BCD is when base 10 precision is needed. Business calculations done in BCD don't require adjusting for the errors from conversion back to decimal before display. Some numerical methods that mathematicians make may be easier and simpler to program in BCD because of the ease of display.

On the other hand, BCD requires more storage for a given size number. In the old days when numbers were kept in "electronic brains" where each bit needed a 12AT7 tube to hold it, this was important. Binary saved space and space was money, time, and heat dissipation. Clever efficient binary arithmetic was the result we inherited. But with cheap, plentiful memory in the Apple, the choice is yours if you want BCD instead.

BCD is formatted as two decimal digits per byte. Each byte is divided into two *nibbles* of four bits each. The lowest nibble is bit 0 to bit 3 and contains the least significant digit; the high nibble is bit 4 to bit 7 and contains the most significant digit. Several bytes grouped together hold long numbers, usually in descending order of significance. A dump of a multibyte BCD number can be read directly. For example,

<div align="center">

1F00: 23 18 40 00

</div>

dumping a BCD number at $1F00.1F03 is read as twenty-three million, one hundred eighty-four thousand. Simple.

Calculations in BCD are as easy as those in binary. Just use SED before any loop or sequence using ADC or SBC instructions that you want to work in decimal. Be sure to CLD immediately in the code where finished with decimal calculations.

Formatting is much easier than binary. The trick is to have a couple of routines that put the low nibble of the A-reg into a field of BCD and fetch a nibble from the field to the A-reg. For example, here is a routine to rotate the entire field by one nibble, four bits, with the low nibble of the A-reg (see Fig. 3-19). A left rotation, the routine puts the A-reg into the least significant digit and fetches the most significant digit to the A-reg.

Fig. 3-19. Left shift of field into A-reg.

```
        NLEFT:   ASL A        ;move digit
                 ASL A        ;from low nibble
                 ASL A        ;to high nibble
                 ASL A        ;in A-reg
                 LDY #4
        NLEFT1:  ROL A        ;rotate A-reg
                 JSR LEFT     ;rotate FIELD
                 DEY
                 BNE NLEFT1   ;repeat four times
                 ROL A        ;last one from C-flag!
                 RTS
```

The LEFT routine to rotate a field by one byte is used. The corresponding RIGHT routine can be used to do the same rotation of a nibble. This time, the A-reg adjustment is made after the rotation.

```
        NRIGHT:  CLC
                 LDY #4
        NRIGHT1: ROR A        ;rotate A-reg
                 JSR RIGHT    ;rotate FIELD
                 DEY
                 BNE NRIGHT1  ;repeat four times
                 ROR A        ;last one!
                 LSR A        ;move digit
                 LSR A        ;from high nibble
                 LSR A        ;to low nibble
                 LSR A        ;on A-reg
                 RTS
```

If you write your own NLEFT and NRIGHT routines, you can simply imbed the LEFT and RIGHT routines and avoid the JSR/RTS instructions.

When you work with BCD, you will find the conversion to display characters simple. See Example 3-3.

Working with both binary and BCD requires the use of conversion from binary to BCD. Example 3-4 does this.

Use the two examples just given to write a conversion from BCD to binary. Then you have all the little utilities you need to work with BCD.

Example 3-3.

```
SOURCE FILE: EXAMPLE 3.3
0000:              1 ********************************
0000:              2 * EXAMPLE 3.3                   *
0000:              3 * CONVERT BINARY TO DISPLAY     *
0000:              4 *                               *
0000:              5 * USES $50.53 OF PAGE ZERO      *
0000:              6 * CALLS DIVID OF EXAMPLE 3.2    *
0000:              7 * TO CALL:                      *
0000:              8 *    $50.51 <--- BINARY NUMBER  *
0000:              9 *       JSR BDISP               *
0000:             10 *    STRING.STRING+4 --->       *
0000:             11 *       A STRING OF FIVE CHARS  *
0000:             12 ********************************
0000:             13 *
0000:             14 *
1000:             15 DIVID   EQU $1000        TEST LOCATI
ON
0000:             16 *
0000:             17 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 3.3.OBJ0

1040:             18         ORG $1040        FOR TEST
1040:             19 *
1040:             20 *
1040:A9 0A        21 BDISP   LDA #10
1042:85 53        22         STA $53          DIVISOR
1044:A0 04        23         LDY #4
1046:20 00 10     24 BDISP1  JSR DIVID        DIVIDE BY T
EN
1049:A5 52        25         LDA $52          GET REMAIND
ER
104B:09 B0        26         ORA #$B0         TO MAKE DIG
IT
104D:99 54 10     27         STA STRING,Y
1050:88           28         DEY
1051:10 F3        29         BPL BDISP1
1053:60           30         RTS
1054:             31 *
1054:B0 B0 B0     32 STRING  ASC "00000"
1057:B0 B0

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

The alternative to integers for arithmetic data is *floating-point* representation. Floating-point is good for scientific and engineering applications; it provides a wide range of values without the need for a large number of bytes to hold it.

If you work with very large or very small numbers, you appreciate floating-point. A large number like 9.4605×10^{15}, which is the number of meters in a light-year, or 6.62620×10^{-34} (Plank's constant, MKS) could not be kept in a reasonably-sized chunk of memory. They would need too many zeros, zeros that don't tell you anything except where

Example 3-4.

```
SOURCE FILE: EXAMPLE 3.4
0000:              1 *******************************
0000:              2 * EXAMPLE 3.4                 *
0000:              3 * CONVERT BCD TO BINARY       *
0000:              4 *                             *
0000:              5 * Y-REG IS LEFT UNCHANGED     *
0000:              6 * TO CALL:                    *
0000:              7 *    A-REG <--- BCD 00 TO 99  *
0000:              8 *        JSR DECBIN           *
0000:              9 *    A-REG ---> BINARY $00-$63 *
0000:             10 *                             *
0000:             11 *******************************
0000:             12 *
0000:             13 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 3.4.OBJ0

1000:             14          ORG   $1000      FOR TEST
1000:             15 *
1000:A2 00        16 DECBIN   LDX   #0         TO COUNT TE
NS
1002:C9 10        17 DECB1    CMP   #$10
1004:90 06        18          BCC   DECB2      NO MORE TEN
S
1006:38           19          SEC
1007:E9 10        20          SBC   #$10       REMOVE TEN
1009:E8           21          INX
100A:D0 F6        22          BNE   DECB1      ALWAYS
100C:             23 *
100C:             24 * NUMBER OF TENS IN X-REG
100C:             25 *
100C:CA           26 DECB2    DEX
100D:30 05        27          BMI   DECB3      NO MORE TEN
S
100F:18           28          CLC
1010:69 0A        29          ADC   #$0A       ADD TEN
1012:D0 F8        30          BNE   DECB2      ALWAYS
1014:             31 *
1014:             32 * BINARY NUMBER IN A-REG
1014:             33 *
1014:60           34 DECB3    RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

the decimal point is located. The five or six *significant figures* need to be stored, and another number giving the power of ten can fit into one byte only. So, the fixed decimal point that uses positional notation to give the number's size is replaced by a byte giving the position. This byte is separate from the other figures of the number and is called the exponent of the number.

In floating-point form, the length of a light-year would be written as

$$0.94605 \text{ E } 16$$

and Plank's constant as

$$0.66262 \text{ E } -33$$

Each floating-point number has two parts: a *mantissa* that is always a proper fraction, between zero and one, and an *exponent* that is the power of ten that weighs the number. Applesoft displays its floating-point numbers this way when they are large or small. It is close to the way they are actually stored in the Apple.

The mantissa part of the floating-point number is *left-justified*; that is, it has no leading zeros. It is a fraction between zero and one. So, the leftmost digit (in base 10) is the tenths, then the hundredths, then the thousandths, and so on. Most packages keep the mantissa in binary, so the leftmost bit is the half, followed by the quarter, then the eighth, the sixteenth, the thirty-secondth, and so on, from left to right. Several bytes may be used and one bit or entire byte must be designated for the algebraic sign. Often, a left-justified number like a floating-point mantissa is called *normalized*. The routines that do the normalization use the rotates, especially the ROL instruction. As long as the number of shifts needed is counted, the exponent of the number can be adjusted to preserve its value.

The exponent part is best kept as a signed number in one byte. This gives possible values of -128 to $+127$ with the 6502 arithmetic made easy. Multiplication and division of exponents are easy — add exponents when multiplying and subtract them when dividing. Normalizing the mantissa means decrementing the exponent for each left shift, incrementing it for each right shift. With the exponent easy to maintain, operations on floating-point numbers are almost as fast as those on fixed-point numbers.

Floating-point numbers can be kept in either binary or BCD form. If BCD, the exponent is the power of ten and is itself usually in binary. the mantissa in BCD is usually long to accommodate many figures, sixteen or thirty-two being common. BCD is popular for business to avoid roundoffs when converting to and from binary. Typically, binary floating-point packages like Applesoft have four bytes used for

a mantissa and one byte for an exponent, the entire number occupying five bytes.

Floating-point is mainly used for scientific calculations, and it is the main format for Applesoft numbers. It is possible to have BCD floating-point as well; the best usage of each format is given in Table 3-14.

Table 3-14. Number Formats

	Fixed Point	Floating Point
Binary	Internal program	Most scientific
BCD	Instrumentation Accounting	Precision scientific

CHAPTER FOUR

Applesoft BASIC

4.1 THE LANGUAGE

Here are the Applesoft statements and functions for your reference. These are verb keywords; their objects are called their *arguments*. The kind of argument used in each case varies, so several forms of the same verb can be used. Lower case italic is used for stating the argument type; where an example is more appropriate, only upper case is used. To locate a verb for a specific job, use Table 4-1. All verbs are given in alphabetical order for easy lookup.

ABS*(expr)* is a function that returns the absolute, positive value of the expression.

ASC*(string)* is a function that returns the positive ASCII code number of the first character in the string.

ATN*(expr)* is a function that returns the arctangent of the expression. The arctangent angle is in radians.

CALL*(expr)* statement will execute a machine-language routine. The address of the routine must be the value of the expression: for example, CALL − 151.

CHR$*(expr)* is a function to get a character code to become a string. Argument must be the positive ASCII code and the function returns the single character string. This is the inverse function of ASC.

CLEAR is a statement that clears all variables and the stack.

COLOR = *(expr)* is a statement to set the LORES display color. Value should be from zero to fifteen.

Table 4-1. Applesoft Command Sets

Program Flow	Input/Output	Screens	Variable Control	Math, String Functions	Assignment Symbols	Edit/Debug
&	GET...	FLASH	CLEAR	ABS(	LET	CtrlC,
CALL...	IN#...	HOME	DTA...	ASC(	AND	CtrlX and reset
DEF FN...	INPUT...	HTAB...	DIM...	ATN(	OR	CONT
END...	LOAD	INVERSE	FRE(	COS(	NOT	DEL...
FOR...−...	PDL(	NORMAL	READ...	CHR$(	( )	HIMEM:...
TO... STEP...	PEEK(	POS(	RESTORE	EXP(	=	LIST...
GOSUB...	POKE...	TEXT		INT(	+	LOMEM:...
GOTO...	PR#...	VTAB...		LEFT$(	−	NEW
IF...GOTO...	PRINT...	COLOR = ...		LEN(	*	NOTRACE
IF...THEN...	or ?...	GR		LOG(	/	TRACE
ELSE...	RECALL...	HLIN...		MID$(	∧	RUN...
NEXT...	SAVE	PLOT...		RIGHT(		
ONERR GOTO..	SH LOAD	SCRN(		RND(		
POP	SPC(*	VLIN...		SGN(		
REM...	STORE...	DRAW...		SIN(		
RESUME	TAB(*	HCOLOR...		SQR(		
RETURN	WAIT(	HGR		STR(		
SPEED...		HGR2		TAN(		
STOP		HPLOT...		VAL(		
USR(		ROT = ...				
		SCALE = ...				
		XDRAW...				

* Used only in PRINT

CONT is a statement used in immediate mode to continue the execution of a program that was STOPped. It works after ctrl/C and END have halted the program as well.

COS*(expr)* is a function that returns the cosine of the expression. The value of the expression must be in radians.

DATA *value, value, value, . . .* is a statement used to create a list of values for use by the READ statement. One or several values may be listed, as needed. They may be any type, but must be matched by appropriate variables in the corresponding READ statement. Strings containing alphabetics only need not be in quotes, but it is safest to quote all strings as a habit.

DEF FN *fpvar(fpvar)* = *expr* is a statement to define a function. Examples of useful function definitions like DEF FN AD(X) = 256*PEEK(X + 1) + PEEK(X) appear in Chapter One. The variable called X in this example is a dummy variable, and so X remains available for use; it is not *consumed* by the function definition.

DEL *line,line* is a statement that deletes program lines. Both the beginning line named and the ending line named are deleted,

DIM *var size, var size, . . .* is a statement to create *dimensioned variables.* One or more may be declared. The variable named may be a floating-point number, an integer, or a string. The size may have any order; each order may have any dimension. For example, DIM TH(5,9,3) defines TH as a floating-point array of order three, dimensions 6, 10, and 4. Note that the array will be addressed as TH(i,j,k) where i is a value from zero (not one) to five, j is a value from zero to nine, and k is a value from zero to three.

DRAW *expr* AT *expr,expr* is a statement to draw a shape at a given point on the HIRES screen. The AT *expr, expr* is optional; not using it will cause the current plotting position to be used instead. See Chapter Six for details.

END halts program execution. Unlike STOP, it doesn't display any message. By convention, one END statement at line 32767 is used for normal program termination.

EXP*(expr)* is a function that returns the value of e raised to the power given. Note e = 2.7182818, as the base of natural logarithms. This is the inverse function of LOG.

FLASH is a statement to make further PRINTed characters flash on the screen. Since the flash codes are used differently in lowercase displays like the IIe 80-column card, this won't always work the way it should. Undo it with NORMAL.

FN *fpvar(expr)* is a program-defined function to evaluate the expression in its own way. See DEF FN. The argument passed as *expr* will replace the dummy variable in the DEF FN statement.

FOR *fpvar* = *expr* TO *expr* STEP *expr* is a statement to begin a loop. Loop counter *fpvar* must be floating-point; don't use an integer variable. Loop ends whenever *fpvar* value is outside the range of expressions. The STEP is optional, default is + 1. Each FOR must have a NEXT to end the loop.

FRE(0) is a function that returns the amount of free memory, in bytes, remaining to the program. In doing so, it recovers any old strings that have been reassigned and makes their space available again. This *garbage collection* should be done in loops that reassign strings often.

GET *stringvar* is a statement that accepts single-character input without a CR character. Often used for single keystroke responses in menus, cursor movement routines, etc.

GOSUB *line* is a statement to execute a subroutine from within a program.

GOTO *line* is a statement that causes execution to continue at the line given.

GR is a statement to switch the display to LORES graphics from TEXT. Screen is cleared to black and text display remains at bottom four lines. See Chapter Six.

HCOLOR = *expr* is a statement to set the HIRES plotting color. See Chapter Six.

HGR is a statement to switch the display to HIRES graphics. Screen is cleared to black and four lines of text display remain at the bottom. See Chapter Six.

HGR2 works just like HGR, but for HIRES2 instead of HIRES1.

HIMEM:*expr* is a statement that sets the highest memory location available to the BASIC program. See Section 4.2.

HLIN *expr1,expr2* AT *expr3* is a statement to draw a horizontal line in LORES graphics. See Chapter Six.

HOME clears all text within the display window and moves the cursor to the upper left of the window.

HPLOT *expr,expr* is a statement to plot a single point on the HIRES screen. See Chapter Six. May be modified to plot lines by appending: TO *expr,expr* for the second end point. May be modified by more TO extensions to make polygons.

HTAB *expr* is a statement to move the text cursor horizontally, to any column number, 1 to 40 (or 80).

IF*expr* THEN*statement* is a statement that evaluates an expression. If the result is not zero (true) then the given statement is executed. If the result is zero (false) then the following statement is executed. If the given statement does not say otherwise, the following statement will be executed normally after the given statement. If an ELSE is used, the same rule applies; ELSE only invokes one statement as well. For example, IF A = B THEN A$ = "EQUAL" ELSE A$ = "UNEQUAL":A = 5 will set A to the value 5 regardless of the IF.

IN#*slot* sets the current input device to the slot numbered: 1 to 7. See Chapter Six.

INPUT *string;var,var,* . . . is a statement to input variables from the current device. For keyboard use especially, the optional *string;* will be used as a prompt message on the screen. If more than one variable is to be INPUT, separate with commas as shown.

INT*(expr)* is a function that returns the closest integer value less than or equal to the expression. For example, INT(− 5.9) gives 6 and INT(5.9) gives 5.

INVERSE is a statement to display further characters on the text screen as black-on-white. It won't work with active 80-column displays; to cancel its effect, use NORMAL to get white-on-black display to return.

LEFT$*(string,expr)* is a function that returns a string consisting of the *expr* leftmost characters of *string*.

LEN*(string)* is a function that returns the number of characters in the *string* argument.

LET *var = expr/string* is a statement to assign a value to a variable. It may be either string or numeric. A string variable must be assigned with a string; expressions will be converted from floating-point or integer to integer or floating-point values, as the numeric variable requires. Use the ASC and CHR$ functions if you need them to convert between numbers and strings. The LET verb is optional; most assignment statements are made without it.

LIST *line,line* is a statement to list the current program to the current output device. Usually, to display on the text screen. When optional line numbers are given, only that range is LISTed; you can use a (-) instead of a comma (,). A single line may be LISTed; for example, LIST 20100.

LOAD is a statement to read a BASIC program from tape. For disk, a filename must be given; then DOS will intercept the statement as a disk command.

LOG*(expr)* is a function of the natural logarithm (base e) of the expression. This is the inverse function of EXP.

LOMEM:*expr* is a statement to set the lowest address of memory available to BASIC for variables storage.

MID$*(string,expr,expr)* is a function that returns a string of characters beginning with the one in the position given in the first expression; for example, MID$("HELLO",3) returns "LLO." If the second expression is given, it sets the length of the returned string. For example, MID$("HELLO",3,2) returns LL.

NEW is a statement that clears the current program from memory. Use NEW before writing a new program.

NEXT *fpvar,fpvar*, . . . is a statement used to mark the bottom of one or more FOR loops. With no argument, it ends the last (innermost) loop. Use more than one argument to end several loops in one statement; be careful to list them in order from last (innermost loop variable) to first (outermost loop variable).

NORMAL removes the effects of INVERSE and FLASH statements. Future output characters are displayed in normal white-on-black form on the text screen.

NOTRACE turns off the TRACE feature that displays line numbers during execution.

ON *expr* GOSUB *line,line,line,* . . . is a statement to select one of several subroutines according to the value of an expression. The integer value of *expr* selects the first, second, third, etc., line number for the GOSUB. If none are selected, control simply passes to the next statement.

ON *expr* GOTO *line,line,line,* . . . is a statement like the ON . . GOSUB . . in that it selects the line number in exactly the same manner.

ONERR GOTO *line* is a statement that sets Applesoft's error trap to execute at *line* instead of displaying an error message. See details in Chapter One.

PDL*(expr)* is a function that returns the position of a games paddle or joystick. The argument selects one of four paddles: 0, 1, 2, or 3. A joystick normally uses 0 and 1.

PEEK*(expr)* is a function that returns the contents of the memory location whose address is given as the argument.

PLOT *expr1,expr2* is a statement that plots a single LORES pixel. Arguments give the column as *expr1* and the row as *expr2*.

POKE *expr1,expr2* is a statement that writes to a memory location

given by the address *expr1*. The value written is that of *expr2* which must be from zero to 255.

POP is a statement that removes the last return address from the stack. It acts just like a RETURN statement, except control is *not* transferred to the return point; instead, it falls through to the next statement after the POP.

POS(0) is a function that returns the column number of the cursor. The argument is ignored.

PR#*slot* is a statement that sets up the peripheral in the named slot, 1 to 7, as the current output device.

PRINT *list* is a statement to write to the current output device. The *list* contains one or more expressions — numeric and string. Comma or semicolon delimiters may be used. If no list is given, the PRINT generates a CR character anyway.

READ *var,var,var* . . . is a statement to get values in DATA statements assigned to variables. Data type must be compatible, either string or numeric. Syntax permits one or several variables to be named in a READ.

RECALL *arrayname* is a statement that reads data from a tape into a named (DIMensioned) array.

REM is the remark statement. No execution is done.

RESTORE is a statement that restores the READ pointer back to the beginning of the first DATA statement, allowing previously READ data to be re-READ.

RESUME is a statement that ends an error handling routine so control is returned to the statement that caused the error. That statement will re-execute. See ONERR GOTO.

RETURN is a statement that transfers control to the statement following the last GOSUB; it ends a subroutine.

RIGHT$*(string,expr)* is a function that returns the rightmost characters of *string*. The length of this returned substring is the value of *expr*.

RND*(expr)* returns a pseudo-random number, from zero to one. The sign of the argument gives different results: positive arguments generate new random numbers. Zero argument gives the same random number it gave last time. Negative arguments always give a specific result, so a sequence of numbers are *seeded.*

SQR*(expr)* is a function that returns the square root of its argument.

STOP is a statement to stop program execution. Used in debugging, it prints the line number of the statement.

STORE *arrayname* is a statement to save data to tape from the named array. See also RECALL.

STR$(*expr*) is a function that returns a string representing the value of its argument. Same function that is implied in PRINT statements.

TAB(*expr*) is a statement modifier for the PRINT statement. It advances the cursor to the column number given by the argument.

TAN(*expr*) is a function that returns the trigonometric tangent of the argument which must be in radians.

TEXT is a statement that resets the display to the normal, 40-column TEXT screen. The window is reset to full size (24 × 40) and the cursor positioned at lower left.

TRACE is a command that causes each line number to be displayed as their statements are executed. Used for debugging; see also NOTRACE.

USR(*expr*) is a user-defined function that executes the machine language routine setup by a JMP instruction at $0A.0C in Page Zero. The argument is passed in FAC ($9D.A2) and the result returned in FAC as well.

VAL(*string*) is a function that returns the value of the number given as a string. This is the function implied in the INPUT statement. For example, VAL("294.5") gives a numeric value of 294.5.

VLIN *expr1, expr2* AT *expr3* is a statement that draws a vertical line in LORES graphics. See Chapter Six.

VTAB *expr* is a statement to position the text cursor to the row number given by *expr*. Values must be one to 24.

WAIT *expr1,expr2,expr3* is a statement used to wait until something happens at the address given by *expr1*. The WAIT is completed when any bit in *expr2* is also on at the location. However, if the optional *expr3* is given, it tests for either on or off, according to *expr3* mask bits. See discussion in Chapter One.

XDRAW *expr1* AT *expr2,expr3* is a statement to draw a shape. See Chapter Six for details.

4.2 THE STRUCTURE

4.2.1 Memory Usage

In both the Apple II Plus and IIe models, Applesoft resides in ROM at $D000.F7FF, between the input/output addresses and the Monitor.

It is just a collection of routines and constants whose role is the ultimate execution of specific statements and functions. It does this by using both its own routines and several standard ones in the Monitor.

When you type a command, it is read from the keyboard input buffer at $0200 where it was placed by the GETLN routine at $FD6A. Applesoft then uses its own routines to dispose of the line. If the command begins with a line number, then it won't execute any statements; instead it creates a program line and stores it in RAM. Program lines are kept in sequence to make up what is called the program *text* in memory. Program text normally begins at location $0801 and continues as unbroken memory. This text is not the line you typed in; rather, it has been encoded in a short form that Applesoft can read. The LIST command causes Applesoft to read the program text and output it in decoded form to make it human-readable.

Whenever you RUN a program, Applesoft stops taking commands from the keyboard buffer and takes them from the current program — the program text. By using a Page Zero pointer, it moves through your program, reading and interpreting the program text. Remember that your BASIC program is the data that Applesoft reads.

Then, while your program is RUNning, it needs space for variables. Whenever a variable is used for the first time or an array is DIMensioned, Applesoft adds it to the end of the program text together with its allocated name. So, as the program runs, Applesoft is building variables in RAM following the program text, lengthening the entire program upwards in memory.

This variables storage space always increases during the program RUN. It never decreases; Applesoft cannot recover space from old variables. You must reuse the same variable in different parts of your program if you have to conserve memory.

One consequence of using a lot of variable space is the encroachment of HIRES1 at $2000.3FFF during the run. When this happens, the variables overwrite screen graphics and you may see it on the screen. Then, when graphics are drawn, they clobber the variables! To get around this, the LOMEM: statement must be used before any variables are referenced in the program. Instead of starting at the next location following program text, Applesoft then will start variables storage at the LOMEM address. Simply set LOMEM to $4000 (16384), just past HIRES1.

Because variable space cannot be recovered and reused, the string variables don't hold the strings themselves. Instead, they point to the strings kept in other parts of memory: in between quotes in program

text, in DATA statements, and in the highest RAM available. Whenever a string is created during execution, the new string is stored by Applesoft in memory, below DOS at $9600. This upper limit can be changed before any strings are referenced in the program by the HIMEM: statement. But, the string variables themselves, pointing to the actual strings, are in variables storage. So, the string space at the top of memory can be managed to recover the space occupied by old strings no longer referenced by string variables.

While variable space cannot be recovered once a variable is brought into existence, the created strings that the string variables point to may be recovered. Each time a new string is created, the string storage enlarges at the top of memory. When it runs out of space, Applesoft does a *garbage collection* to repack the valid strings at the top and thereby free up new memory space. By weeding out dead strings regularly, you can avoid these strings from encroaching down into HIRES screens. Putting the statement

$$X = FRE(0)$$

in your program will do this for you. Put it in your main loop and any loop that uses strings. It will force garbage collection regularly instead of waiting until the strings grow down to the top of the program variables and arrays.

To summarize, Applesoft executes your BASIC program by stepping through program text, usually from $0801. This encoded text is followed in memory by variables and arrays that Applesoft builds as they are encountered. Strings are created from the top of memory down, below $9600. In some programs, the HIRES screens in the $2000.5FFF area must be protected against variables and strings, using LOMEM: and FRE(0) features.

Applesoft RAM usage in the normal case is shown in Fig. 4-1.

If you dump the block of Page Zero $67.74 you will see seven pointers that Applesoft sets up and uses to maintain your BASIC programs. They point to the boundaries in RAM where your program resides, where variables are maintained, and where strings are stored. Knowing what these pointers do and how to set them yourself enables you to control your program's memory map. You can tell Applesoft where to load, where to keep variables, and where to store strings. And you can examine these pointers to see what Applesoft is doing; see memory contention problems before they occur. See Fig. 4-2.
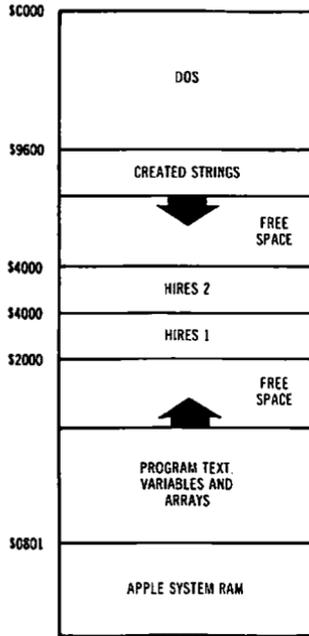
Fig. 4-1. Applesoft RAM usage.
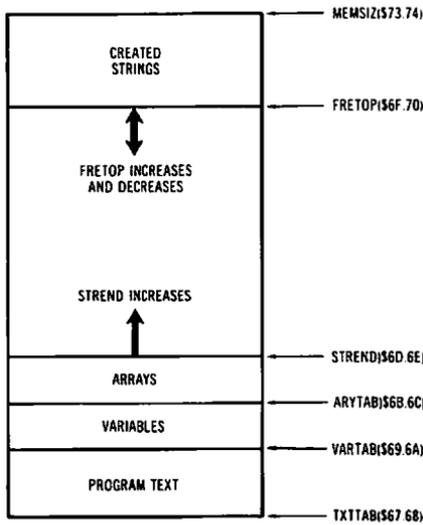
Fig. 4-2. Applesoft memory pointers.

TXTTAB at $67.68 points to the start of program text. Usually this is $801 but you can change it to any other address before loading a program. It won't work properly on a program already loaded. The low byte in $67 is best left as $01; just change $68 to the page number of your new program area. For example, you could POKE 104,96 to set it to $6001. Applesoft expects the contents of the byte located immediately before the text to be zero, so a POKE 24576,0 would set $6000 to zero. A load of an Applesoft program will put program text in memory beginning at $6001, above the HIRES pages in this case.

VARTAB at $69.6A points normally to one or two bytes beyond the end of program text. It points to the beginning of variable storage. When a program runs, Applesoft builds variables from the location that VARTAB points to, and always puts the simple variables ahead of array variables. You normally change VARTAB indirectly using the LOMEM: command in Applesoft. Within a program, use LOMEM: to set VARTAB for you before any variable references occur. For example, LOMEM:16384 sets VARTAB to $4000 to protect HIRES1, and LOMEM:24576 sets it to $6000, protecting the entire HIRES screen area from encroachment by variables.

ARYTAB at $6B.6C points to a spot within the variable storage where the arrays begin. This is convenient for Applesoft, so it can begin looking for DIMensioned variables here without having to search simple ones first.

STREND at $6D.6E marks the end of variables storage. It marks the beginning of the *free space* between the end of arrays and the bottom of string storage. You can check this location to see if variables do indeed contend with a HIRES screen in graphics programs.

FRETOP at 6F.70 points to the bottom of string storage. It starts out pointing to the top of memory like MEMSIZ below, and points below the strings as each is added. It releases space by being reset to a higher address during garbage collection. You can read it to see if your strings are crowding HIRES screen memory, or read the difference between FRETOP and STREND after a garbage collection in the variable returned by the FRE(0) function.

FRESPC at $71.72 is a pointer used by the string handling routines — there is no need to refer to it.

MEMSIZ at $73.74 is the pointer to the highest RAM address available to Applesoft, less one. Normally it is set to $9600 with DOS. Without DOS it would be set to $C000. Sometimes machine language routines are put just below $9600 and MEMSIZ changed to point to a

lower address below the routine. This must be done before any strings are assigned in the program, and is best accomplished with the HIMEM: command. For instance, HIMEM: 36864 sets MEMSIZ to $9000 and leaves the $9000.95FF chunk of memory free for the machine language routine, safe from Applesoft. Be careful in this area because the MAXFILES command to DOS will change MEMSIZ as well. Anything besides MAXFILES 3 will change MEMSIZ to something besides $9600.

You can change TXTTAB only before loading a program. VARTAB is best changed by a LOMEM: command; MEMSIZ by a HIMEM: one. The other pointers may be watched for potential trouble.

Here are some common ways of fitting machine-language programs into memory with an Applesoft BASIC program. The one you choose will depend on the program size, the way it uses variables, and how large your machine language program is. When the memory map has been chosen, you just adjust the pointers accordingly to realize it; some can be done from within the BASIC program itself.

The most common method is to make room at the top of memory using the HIMEM: statement. The result is shown in Fig. 4-3. Before any strings are referenced, your program simply states HIMEM:36864
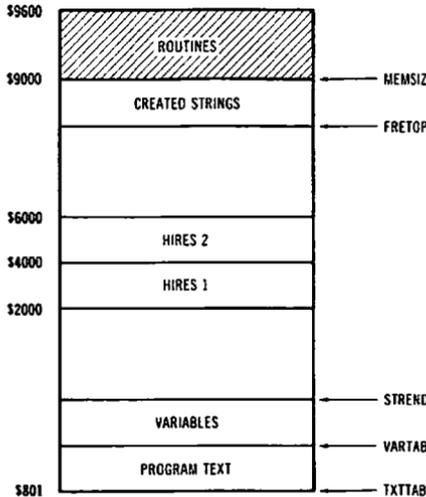
Fig. 4-3. Applesoft and routine, first method.

to set MEMSIZ down to $9000. This protects the $9000.95FF chunk of memory against Applesoft for you. It works; it's simple. But, it won't protect against a MAXFILES command to DOS, and it won't solve the memory contention problem when HGR or HGR2 clobbers variables or strings. So, use it for programs that don't use HIRES, and don't use MAXFILES.

Suppose you want to use HIRES and you want to have a machine language routine as well. Then, you could use a LOMEM: statement to set VARTAB to point above the HIRES screen memory. A LOMEM:16384 sets VARTAB to $4000, protecting HIRES1; and, a LOMEM:24576 sets VARTAB to $6000, protecting HIRES1 and HIRES2. You can usually find space between the end of program text and $2000 to place your routines. Kep the routines as close to $2000 as practicable so as to leave room for additional program text. If both program text and routine can fit into $0801.1FFF, this is the simplest way to have your HIRES, BASIC program, and machine language routine coexisting, as in Fig. 4-4.

$9600		MEMSIZ
	CREATED STRINGS	
		FRETOP
		STREND
	VARIABLES	
$6000		VARTAB
	HIRES 2	
$4000	HIRES 1	
$2000	ROUTINES	
	PROGRAM TEXT	
$0801		TXTTAB
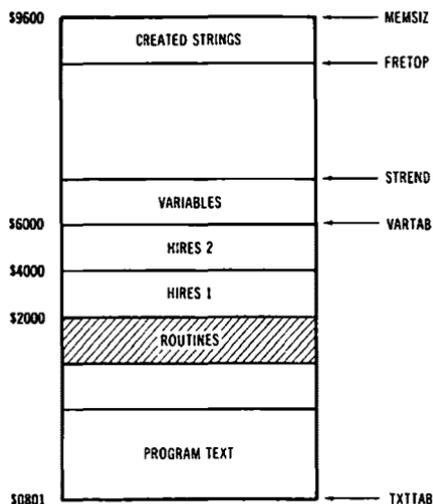
Fig. 4-4. Applesoft and routine, second method.

If the routine won't fit between the end of program text and HIRES1, you can find another place for it above the HIRES screen memory, as in Fig. 4-5. Make your routine start at $4000 (or $6000 if you use both screens) and note the next free location beyond the routine. Convert this location to decimal and use it in a LOMEM:

```
$9600 ┌─────────────────────┐ ◄──── MEMSIZ
      │   CREATED STRINGS   │
      │                     │ ◄──── FRETOP
      │                     │
      │                     │ ◄──── STREND
      │      VARIABLES      │
      ├─────────────────────┤ ◄──── VARTAB
      │////////ROUTINES/////│
$6000 ├─────────────────────┤
      │       HIRES 2       │
$4000 ├─────────────────────┤
      │       HIRES 1       │
$2000 ├─────────────────────┤
      │                     │
      │                     │
      │    PROGRAM TEXT     │
      │                     │
$0801 └─────────────────────┘ ◄──── TXTTAB
```
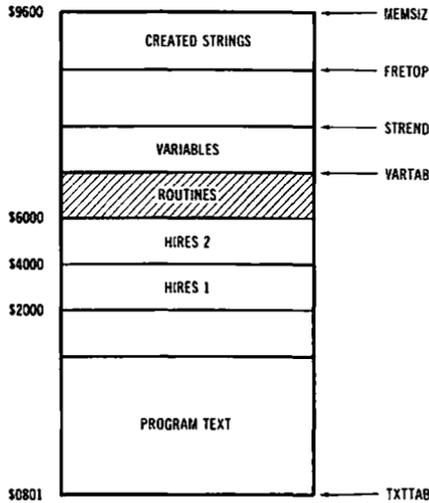
Fig. 4-5. Applesoft and routine, third method.

statement. This will set VARTAB to protect all memory from program text to the last address of your routine. Provided the routine is not very long, you can usually spare the room.

If you are faced with the other extreme of having a long machine language routine and a short BASIC program, you must then reverse their storage. Put your routine between $0800 and $2000; then put the program text above the HIRES screen(s). For instance, to create the arrangement shown in Fig. 4-6 use a POKE 104,64 and POKE 24576,0 before the BASIC program is loaded. Alternatively, POKE 104,96 and POKE 24576,0 produces the map in Fig. 4-7 where TXTTAB is set to $6001 (from $0801) instead of $4001. The location before the start of program text is always set to zero: $800 is normally zero, so $4000 or $6000 is set to zero as well.

The drawback to this method is that the program itself cannot do it. A separate program in machine language can do it, or use an EXEC file to do the POKEs and the RUN statements. For large graphics routines that use graphics and provide for user BASIC drivers, this is the way to go.

There are other ways. The method of writing machine language loaders to setup the maps as the routines are first loaded is given in Chapter Seven.
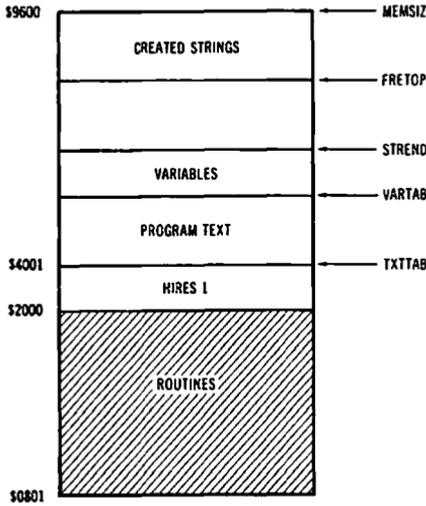
Fig. 4-6. Applesoft and routine, fourth method.
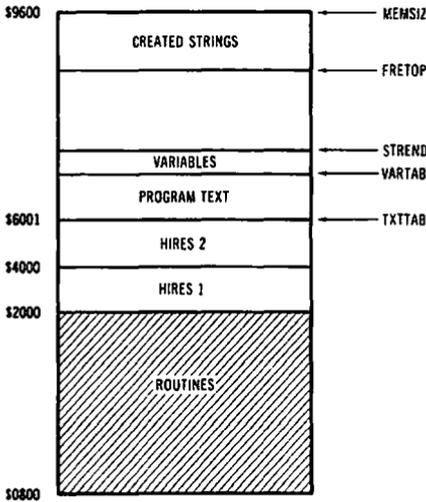
Fig. 4-7. Applesoft and routine, fifth method.

4.2.2 Data Storage

Data are stored in variables and arrays as one of the three types: floating-point, integer, and string. The user defined functions have

their names stored in the same manner as variables. First, here is how each variable type is kept in memory.

Floating-point numbers are stored as simple variables by Applesoft during the execution of your BASIC program. By dumping the memory between the addresses pointed to by VARTAB and ARYTAB you can examine them directly.

Each number is stored in seven bytes. The first two bytes contain the variable name in ASCII code. Each ASCII name byte has bit 7 clear. Floating-point (FP) variable names are the only ones like this: other variables have at least one of the bit 7s set in their names. Applesoft uses this scheme to tell the variable types apart. See Table 4-2.

Table 4-2. Type-Encoded Applesoft Variable Names

Variable Type	First Byte	Second Byte	Example Name
FP	Positive	Positive	A1 as 41,31
Integer	Negative	Negative	A1% as C1,B1
String	Positive	Negative	A1$ as 41,B1
Function	Negative	Positive	FN A1 as C1,31

The remaining five bytes of the variable after its name are its contents. For floating-point numbers, the contents consist of a *one-byte exponent* followed by a *four-byte mantissa.*

The exponent is in what is called *excess-$80* form. In this scheme, $80 is the code for a zero exponent; $81, $82, $83, etc., are positive exponents of 1, 2, 3, etc., respectively. An exponent of minus one would be encoded as $7F. Minus two as $7E. The exponent has $80 added to it when it is converted to excess-$80 form.

The mantissa has been normalized and appears in decreasing order of significance. Fig. 4-8 shows the format of Applesoft variables. Its four bytes are called HO, MOH, MO, LO in left-to-right order. The binary point is to the immediate left of HO. Because of normalization, the leftmost bit, bit 7 of HO, is always one. So, it doesn't carry any information. In variable storage, this bit is replaced by the mantissa's sign, which is zero for plus and one for minus. This replacement to increase the information content is called *packing.*

To summarize the FP variable, it consists of two bytes of name and five bytes of number. The name has both bytes with their bit 7s clear. The number consists of one byte of excess-$80 exponent followed by four bytes of packed mantissa. As a simple variable, its total size is seven bytes.

POS	POS	EXP	HO	MOH	MO	LO

NAME ⌐
EXCESS-$80 EXPONENT ⌐
PACKED MANTISSA ⌐

(A) Floating point number.

NEG	NEG	HI	LO	0	0	0

NAME ⌐
SIGNED VALUE ⌐

(B) Integer number.

POS	NEG	LEN	LO	HI	0	0

NAME ⌐
DESCRIPTOR (LENGTH & ADDRESS) ⌐

(C) String.

NEG	POS	LO	HI	LO	HI	B
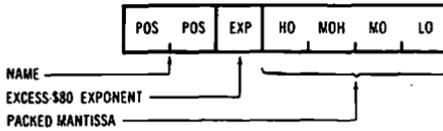
NAME ⌐
DEF'N ADDRESS ⌐
DUMMY ARG. ADDRESS ⌐
FIRST BYTE OF DEF'N ⌐
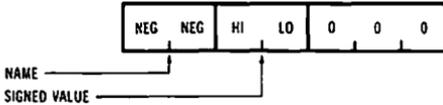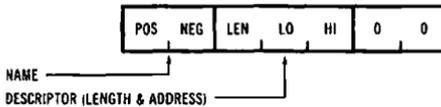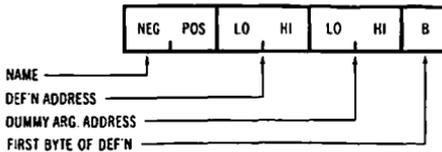
(D) User defined function.

Fig. 4-8. Format of Applesoft variables.

Integers also appear in simple variables as seven bytes. The name in the first two bytes has both bit 7s set to distinguish it as an integer. The contents of the integer consist of a two-byte signed number followed by three zeros.

An example shows how the name works. Suppose that the variable name TA were encountered by Applesoft as it executed your program. It would create a new FP variable as a floating-point number and the name code would be $54 followed by $41, the codes for T and A. But suppose the variable were named TA% instead. Applesoft then creates a new integer variable and gives it the name code $B4 and $A1 instead. The difference in names is that the bit 7s are both set for integer name but both clear for FP name.

The contents are the integer itself in the two bytes following the name. These two bytes are in decreasing order of significance: high-order byte followed by low-order byte. This is just opposite the format of addresses, so don't confuse the two. Integers are signed numbers with bit 7 of the high-order byte giving the sign: zero for plus, one for minus. This gives integers a domain of -32768 to $+32767$ in value.

The three remaining bytes of an integer variable are unused and Applesoft sets them to zero.

Strings are also built as simple variables by Applesoft executing BASIC program text. As a simple variable, a string variable consists of a name in two bytes followed by five bytes of content. The difference with strings, however, is that the content is not the string itself but a pointer to the string. The name, on the other hand, behaves like other variable names.

If a simple variable is a string, its name has the first byte with bit 7 clear and the second byte with bit 7 set. So, a string name like TA$ would be encoded as $54 followed by $A1.

The content of a string variable is a three-byte *descriptor* followed by two unused bytes set to zero. The descriptor gives the length of the string followed by its address — low byte then high byte. By copying the descriptor into Page Zero, Applesoft can access the string using indirect indexed addressing. With the length included, the descriptor gives all the information needed to handle the string it describes.

The string itself can be anywhere in memory. If assigned as a literal in the listing like

A$ = "LITERAL STRING"

it will be pointed to in the listing itself by the new descriptor. Similarly, a READ of a DATA statement causes the descriptor of the variable to point to the string within the DATA statement. If Applesoft creates a new string as with a concatenation like

A$ = A$ + "SUFFIX"

the new string is built in "free space" at the top of memory using FRESPC. Then the descriptor in the A$ variable is given its length and address. Regardless of where the string itself resides, it is assigned to its variable by putting its length, address-low, and address-high as the descriptor of that variable.

Functions are kept as simple variables by Applesoft as well. The two-byte name has the first byte with bit 7 set and the second byte with bit 7 clear. So, a name like FN TA encodes as $B4 then $41. This naming scheme completes the pattern used for FP numbers, integers and strings. The contents of a function variable is again different from other variable contents.

The function variable points to the function definition in BASIC program text where you made your DEF FN statement. It also points to the FP number within the variable being used as its dummy argument. For example, suppose you defined

DEF FN LO(Z) = Z − 256*INT(Z/256)

in your program. Applesoft encodes it in program text and when run creates the function variable LO. The contents of the variable consists of two bytes as the address of the first byte of the definition in program text. Then two bytes point to a second variable created at this time − the dummy variable FP number. The function points to this dummy variable *content,* not its name, which is undefined. The last byte in the contents of the function variable contains the first byte of the FN definition. So, the five bytes of a function variable are: address of first byte of the function definition, address of first byte of the dummy argument and the first byte itself of the function definition.

Arrays are built and maintained in memory above the simple variables, from ARYTAB to STREND. After a program runs, you can find the start address for the first array in ARYTAB.

Each array consists of a header followed by all its elements. The header is 2N + 5 bytes in length where N is the number of subscripts declared in the DIM statement. The contents of the header are listed in Table 4-3. Notice that you can have the same names for arrays as you

Table 4-3. Array Header

Number of Bytes	Contents
2	Array name: type-encoded
	FP (pos) (pos)
	Integer (neg) (neg)
	String (pos) (neg)
2	Array length, including entire header
1	N = number of subscripts, the order
2N	List of dimensions, last to first

do for simple variables because they are type-encoded the same way. The length of the entire array is the total number of bytes of both the header and all its entries. The number of subscripts is also known as the *order* of the array. The dimensions themselves are numbers one larger than the values given in the DIM statement in order to count the zeroth elements. And, these dimensions appear in the header in reverse order.

An example should make this clear. If you had an array of FP numbers called BX that you DIMensioned as

DIM BX(3,1,2)

it would be created in memory as shown in Table 4-4.

Table 4-4. Example Array Created by DIM BX(3,1,2)

Location	Contents	Description
00.01	42 58	Array name, BX
02.03	83 00	Array length = $83 = 131
04	3	Order of array
05.06	03 00	3rd dimension = 3
07.08	02 00	2nd dimension = 2
09.0A	04 00	1st dimension = 4
0B.0F	Zeros	Element BX(0,0,0)
10.14	Zeros	Element BX(1,0,0)
15.19	Zeros	Element BX(2,0,0)
1A.1E	Zeros	Element BX(3,0,0)
1F.23	Zeros	Element BX(0,1,0)
24.28	Zeros	Element BX(1,1,0)
29.2D	Zeros	Element BX(2,1,0)
2E.32	Zeros	Element BX(3,1,0)
33.37	Zeros	Element BX(0,0,1)
38.3C	Zeros	Element BX(1,0,1)
3D.41	Zeros	Element BX(2,0,1)
42.46	Zeros	Element BX(3,0,1)
47.4B	Zeros	Element BX(0,1,1)
4C.50	Zeros	Element BX(1,1,1)
51.55	Zeros	Element BX(2,1,1)
56.5A	Zeros	Element BX(3,1,1)
5B.5F	Zeros	Element BX(0,0,2)
60.64	Zeros	Element BX(1,0.2)
65.69	Zeros	Element BX(2,0,2)
6A.6E	Zeros	Element Bx(3,0,2)
6F.73	Zeros	Element BX(0,1,2)
74.78	Zeros	Element BX(1,1,2)
79.7D	Zeros	Element BX(2,1,2)
7E.82	Zeros	Element Bx(3,1,2)

The name BX is encoded as the two positive ASCII codes, 42 and 58, because the type is FP. The length is 131 bytes. The order is three because there are three subscripts given in the DIM statement. These dimensions are listed as three, two, and four. The three is the third, or last, dimension given as "2" in the DIM statement. The two is the second dimension given as a "1" in the DIM statement. And, the four is the first dimension given as "3" in the DIM statement. The total number of elements in the array is the product of its dimensions: here, $4 \times 2 \times 3 = 24$.

The 24 elements follow the header, beginning with the twelfth byte at location $0B. The DIM statement sets them all to zeros. There are five bytes each for FP numbers. Notice in particular that the innermost dimension varies the fastest and the outermost dimension varies the slowest. This is important when writing BASIC programs that you want to step quickly through arrays.

Unlike simple variables, array elements have lengths that depend on their type. Simple variables take five bytes regardless of type. Array elements take five bytes for FP type only. Integer type elements take only two bytes each; string type elements take only three.

4.2.3 Program Text

Program text is kept in memory starting at the target of TXTTAB, usually $0801. Each new line is entered in sequence. Together, all the lines of the text make up a linked list data structure.

Each record in the linked list contains a two-byte pointer to the next record in the list. The remainder of the record is the line and may be any length. The last byte of each line is always zero, however, so the end of the line can be recognized easily by routines that read it. These variable length records follow one another, in line number sequence in program text.

The pointer beginning each record is in low-byte/high-byte order. It contains the address of the first byte of the next record, which is its low-byte pointer. At the end of the Program text file is a *null record*. This record has two zero bytes instead of a pointer and tells the searcher that there are no more records — no more program lines.

Within each record, the pointer is followed by the line number in two bytes. The remaining bytes are the text of the line in ASCII code. Note that the seven-bit is used. If there is more than one statement in the line, they are separated by colons, ASCII code $3A. The last state-

ment of any line is terminated by a zero byte. You can see that some bytes are greater than $7F. These bytes are called *tokens*. Instead of storing the ASCII code for commands as they are typed, the commands are kept with tokens of only one byte each. For instance, $84 is the code for INPUT and $DA for SQR. This scheme saves some space.

Use the Applesoft Token Table to read tokens from BASIC program dumps.

Look at this BASIC program and its dump.

```
10   TEXT:HOME:VTAB 20
20   PRINT"HELLO, WORLD"
30   DIM SE$(4,2)
40   SE$(1,1) = "LITERAL"
50   END
```

```
801:   0D 08 0A 00 89 3A 97 3a A2 32 30 00
80D:   20 08 14 00 BA 22 48 45 4C 4C 4F 20 57 4F 52
       4C 44 22 00
820:   2E 08 1E 00 86 53 45 24 28 34 2C 32 29 00
82E:   46 08 28 00 53 45 24 28 31 2C 31 29 3D 22 4C
       49 54 45 52 41 4C 22 00
846:   4C 08 32 00 80 00
84C:   00 00
```

The first record at $801 begins with a pointer to the next record at $80D; this takes the first two bytes. Then the line follows: line ten is indicated in the following two bytes. The three statements of the line have their commands tokenized as 89 for TEXT, 97 for HOME and A2 for VTAB. They are separated by colons, ASCII code $3A. The "20" in the third statement comes out as $32 and $30. Finally, a zero byte terminates the line.

The second record begins at $80D and is pointed to by the pointer at the beginning of the first record. It in turn points to the third record at $820. The line begins with the line number twenty, $14 and $00, and ends with a zero.

The remaining records work the same way. The last record at $84C is pointed to by the record of the last line at $846. The zero value of its pointer marks the end of the program text file.

There are three commands that can clobber your BASIC program

— FP, NEW, and CLEAR. The CLEAR command removes all variables created during program execution, while the FP and NEW commands remove the program text itself in addition to any variables. The FP command is executed by DOS and resets Applesoft's pointers completely, including MEMSIZ and TXTTAB. The NEW command does not reset MEMSIZ and TXTTAB but it resets the others so as to remove your program.

After an FP command, you must restate any HIMEM: or LOMEM: commands and adjust TXTTAB if you want any memory map besides the default. A NEW command leaves your memory map intact, but you cannot LIST any program that was current before the NEW. For instance, if NEW was given with the above program in memory, a dump of memory after the NEW command would reveal

```
800: 00 00 00 0A 00 89 3A 97
808: 3A A2 32 30 00 20 08 14
810: 00 BA 22 48 45 4C 4C 4F
          etc.
```

The program is still there. All that the NEW command did to the program text was to replace the link in the first record with zeros. All you have to do to recover it is find the address of the second record and restore the first link.

The end of line token (zero) for the first line is at $80C. The following two bytes link to $820, a reasonable address. So, the next instruction starts there, at $80D. This is the first link that NEW clobbered:

```
801: 0D 08
```

By replacing the link at 801, you can recover any program that was accidentally wiped out with the NEW command.

4.3 INTERFACING TO ML ROUTINES

4.3.1 Three Methods

Applesoft provides you with three different methods of invoking machine language routines from your BASIC programs. The one you choose in any situation depends on how complicated your call has to be.

If you have a short routine to call, and if you don't have any parameters to pass between your BASIC and ML, then the CALL statement is the way to go. If the routine is short enough, it may fit at $0300.03CF so you won't have to use the HIMEM: or LOMEM: statements. A simple CALL 768 is all that is required. It is possible to follow the CALL with parameters, but this is rarely done. The CALL is used for simple, short routines, usually one per BASIC program.

The problem with the CALL method is its use of a fixed address. If you have a rather large routine, especially one with parameters, you cannot relocate it easily. Instead, you must locate all the CALL statements and change their addresses. In a collection of BASIC programs where each has several CALLs to various routines, this task becomes quite difficult, if not impossible. So, Applesoft has two other methods, each of which makes maintenance easier.

One method is the USR function, which is invoked by

result = USR(*expression*)

where *result* is the returned FP value and *expression* is the argument of your function.

You can create your own single-argument function with the USR. Put the jump instruction ($4C, addr-lo, addr-hi) in memory at $000A.000C by POKEs at the initialization of your BASIC program. Then, whenever USR is encountered, Applesoft will jump to $000A and find your routine address to execute. The expression you pass in the argument will be waiting for your ML routine in the FP accumulator in Page Zero, FAC. Then your routine can process the argument using the floating-point routines directly and leave the result you want in FAC. An RTS will return to Applesoft and deliver your result as your BASIC program continues its execution.

With only one address reference in the entire BASIC program, you can easily change it if you relocate your ML routine elsewhere in memory. And, USR gives you a simple parameter passing mechanism, at least for one FP parameter.

Then there is the ampersand method. Like USR, it provides an address where you put a jump instruction to your ML routine. But it doesn't pass any parameters for you; you have to build the pass logic yourself. However, this turns out to be easy with Applesoft's routines.

Using the ampersand, you put $4C, addr-lo, addr-hi at locations $03F5.03F7. Then you use "&" as the command in your BASIC pro-

gram whenever you want to invoke your routine. The ampersand gives you the jump vector flexibility to locate your routine that you have with the USR method, but without being tied down to one parameter. It is the most flexible method, both in terms of locating the routine and passing parameters.

4.3.2 The Ampersand Method

To pass parameters successfully, you must use the Applesoft interpreter directly in your ML routine. In particular, you need the little routine in Applesoft that actually points to and reads the character stream from your BASIC program. By knowing this routine and calling it directly yourself, you can work with your parameter list directly from your ML routines. See Table 4-5.

The routine that fetches characters from the BASIC Program text is called CHRGET. Applesoft puts this routine in Page Zero and always calls it there at $00B1. You can disassemble it there and have a look.

The pointer to the current character in Program text is at $B8.B9 in Page Zero and is called TXTPTR. It is imbedded in CHRGET as the absolute address of a LDA instruction:

$00B7:   AD lo hi

CHRGET increments TXTPTR by one before the LDA instruction so that it keeps advancing TXTPTR as it is called. You can call the routine at $00B7 to get the current character again without advancing TXTPTR. This is often done, and $00B7 is called CHRGOT.

Once the character is fetched, the routine tests it to see if it is a numeral character, 0 to 9, and to see if it is the end of a statement — : or zero. On return, a numeral sets the C-flag and an end of statement sets the Z-flag.

Applesoft sets TXTPTR to TXTTAB when it does a RUN command to begin looking at the Program text. In command mode, TXTPTR looks at the input buffer at $0200 where your direct commands are entered. That's why you see a Page Two address in TXTPTR when you disassemble CHRGET.

When Applesoft jumps to your routine, TXTPTR points to the next character following the jump command. This is because Applesoft routines normally finish their tasks by jumping to CHRGET and finding the delimiting character of their task. With that delimiter still in

Table 4-5. Applesoft Parameter Passing Routines

		Program Text Syntax:
CHRGET	$00B1	advance TXTPTR, get character INTO A-reg
CHRGOT	$00B7	re-get character
DATA	$D995	advance TXTPTR to end of statement
SNERR	$DEC9	bomb program with "SYNTAX ERROR"
ISLETC	$E07D	edit A-reg "A" to "Z"
CHKCOM	$DEBE	gobble comma
CHKOPN	$DEBB	gobble "("
CHKCLS	$DEB8	gobble ")"
		Passing by Value:
GETBYT	$E6F8	get expression to X-reg
FRMNUM	$DD67	get expression to FAC
GETADR	$E752	fix FAC to LINNUM
		Passing by Reference:
PTRGET	$DFE3	find named variable, addr VARPNT
GETARYPT	$F7D9	find array, address name LOWTR
MOVFM	$EAF9	unpack (Y,A) to FAC
MOVMF	$EB2B	pack FAC to (Y,A)
CONUPK	$E9E3	unpack (Y,A) to ARG
MOVAF	$EB63	move FAC to ARG
MOVFA	$EB53	move ARG to FAC
STRINI	$E3D5	create new string space
MOVSTR	$E5E2	move string into new space
		Page Zero Data and Pointers:
LINNUM	$50.51	unsigned integer, lo-hi format
FAC	$9D.A2	FP accumulator
ARG	$A5.AA	FP argument
VARPNT	$83.84	pointer to variable value
LOWTR	$9B.9C	pointer to array variable (name)
DSCTMP	$9D.9F	string descriptor: length, lo, hi
TXTPTR	$B8.B9	interpreter pointer, in CHRGET/GOT

the A-reg, it turns to its next task. If your routine is that next task, then you have that next character in the A-reg waiting for you when you get control.

To read your parameter(s) from BASIC, you fetch the characters immediately following the calling command and interpret them, perhaps using Applesoft's routines to help you. Such a call might look like

&(A$, 4-5*DR, T)

When control passes to your routine pointed to by the JMP at $3F5, the "(" is in the A-reg. By JSR CHRGET you advance TXTPTR by one to the A, fetching it to the A-reg. And so on. If you want to ignore the remaining parameters and skip to the end of statement, then a routine called DATA will do that. Just JMP DATA and you will return control back to normal Applesoft execution with TXTPTR at the end of your call statement. It's good practice to end your routines with a JMP DATA in all cases that continue BASIC execution.

One task you must perform when reading a parameter list, whether with CHRGET or routines that use CHRGET, is syntax checking. Parameter lists are usually enclosed in brackets and with the parameters themselves separated by commas. You must test to see that the right parameters are in the right place, and that brackets and commas are where they are expected. You may include a command word as a parameter, and interpret it in your routine, rejecting it if it is meaningless to you. Whenever you have to reject the parameter list, you can JMP SNERR to exit your routine. This will halt execution of the BASIC program and print

SYNTAX ERROR

to the screen.

So, you have two possible exits for ML routines run under Applesoft — JMP DATA and JMP SNERR. Use DATA for normal continuation of the BASIC program; SNERR to exit the program. Do all your parameter reading and syntax checking in the ML mainline and end it with a single JMP DATA instruction. Use JMP SNERR to trap errors in the mainline. This way, all your JSRed routines have clean parameters and no Applesoft text to deal with.

To do the syntax checking for brackets and commas, use

CHKCOM at $DEBE for commas
CHKOPN at $DEBB for "("
CHKCLS At $DEB8 for ")"

When called, each will test the current character by using CHRGOT. If it doesn't match, then it bombs your program by jumping to SNERR. If the character is all right, it exits by doing a CHRGET. This leaves TXTPTR pointing to the first character of the next parameter or to the end of statement. Just right for the parameter routines. So, a typical ML mainline would be

```
AMPER:   JSR  CHKOPN  ; gobble "("
         JSR  GETBYT  : get expression to X-reg
         JSR  FIRST   ; Parm 1 in X-reg
         JSR  CHKCOM  ; gobble comma
         JSR  GETBYT  ; get expression to X-reg
         JSR  SECOND  : Parm 2 in X-reg
         JSR  CHKCLS  ; gobble ")"
         JMP  DATA    ; continue BASIC
```

where GETBYT is typical of Applesoft parameter routines and your routines FIRST and SECOND are called with each of the two parameters in the X-reg. The SNERR calls are all within the syntax and parameter routines, making a simple, sequential mainline. The call sequence is

$$\&(\ parm1,\ parm2\ )$$

where *parm1* and *parm2* are expressions of values from 0 to 255.

Let's look at another example. This one has only one parameter, a single letter — "A" to "Z".

```
AMPER:    JSR  CHKOPN  ; gobble "("
          JSR  isletc  ; edit "A" to "Z"
          BCS  AMPER1  ; letter?
          JMP  SNERR   : no. . .bomb BASIC
AMPER1:   JSR  GOTCHA  ; yes. . .interpret it
          JSR  CHKCLS  ; gobble ")"
          JMP  DATA    ; that's all, folks!
```

The ISLETC range tests the A-reg and sets the C-flag if it is A to Z. The GOTCHA routine then has the character in the A-reg as its parameter.

The mainline of your ML routine then has the job of dealing with parameters. It begins with the first character in the A-reg with TXTPTR pointing to its location in the BASIC program. Using syntax-checking routines, parameter routines, and your application routines, it reads and assigns each parameter in turn. At the end, it normally exits with a JMP DATA; abnormal exits are through the SNERR routine. Using this strategy, you must design your ML routine to assign each parameter passed to them in the same order as in the parameter list. If any parameters are returned to BASIC, they

must be at the end of the list in order of availability for return. Normally, the mainline passes through the list, fetching each parameter once and only once. By careful design, perhaps using temporary storage, this will be enough.

There are two ways of passing parameters using Applesoft's routines — by value or by reference. The simplest and easiest of these is passing by value.

A string can be passed by value by putting the string in the parameter list, literally. For example,

&("THIS IS A STRING VALUE")

passes the string between the quotes to the ampersand routine. The quotes must be used to prevent Applesoft *tokenizing* it. Applesoft ignores text between quotes but replaces any substring it recognizes as a BASIC command with a token. Use quotes. In fact, for a single parameter like this,

&"THIS IS A STRING VALUE"

the parameter can be used without delimiters like brackets. The ML mainline is

```
AMPER:    JSR  CHRGET    ;next character
          JSR  ISLETC    ;letter "A" to 'Z'
          BCC  AMPER1    ;letter?
          JSR  GOTCHA    ; yes. . process it
          JMP  AMPER             then get another
AMPER1:   JMP  DATA      ; no. . .exit, all done
```

Such a routine might, for instance, interpret English-like commands. Of course, you can use your own edit instead of ISLETC.

To pass numbers by value, Applesoft has a couple of very useful routines to read expressions. You must pass legal Applesoft expressions, and each of these special routines will leave you with the calculated value at a known memory location.

One of these expression reducers is GETBYT, shown earlier. It must have TXTPTR pointing to the first character of the expression, and it returns the expression to you in the X-reg with TXTPTR pointing to the first character following the expression. If the expres-

sion doesn't make sense, then SNERR is invoked. If the result is not within zero to 255, it can't fit into the X-reg and an ILLEGAL QUANTITY error results. As far as your ML routine is concerned, the value is in the X-reg following the JSR GETBYT and TXTPTR is properly positioned at the expected delimiter — comma or ")".

Larger numeric values can be calculated from parameter expressions by the FRMNUM routine. It works just like GETBYT except it leaves the number in FAC — the floating-point accumulater in Page Zero. You can use the Applesoft floating-point package to work with the value at this point, or you can reduce the FP number to an integer if you wish. A routine called GETADR *fixes* the contents of FAC by changing it to an integer value in address format at a Page Zero location called LINNUM. So, the sequence

<div align="center">

JSR   FRMNUM
JSR   GETADR

</div>

results in a parameter expression evaluated into an integer at LINNUM ($50.51) in address form, low byte, high byte. Just a JSR FRMNUM will reduce the expression to the FAC only. The FRMNUM routine is the general way to get a numeric value, but you'll find GETBYT and GETADR very useful.

If you want to return a parameter from your ML routines to BASIC, then you must pass it by reference; you cannot return a parameter by value without inviting trouble. Passing by reference can be done either way: from ML to BASIC or from BASIC to ML. In the BASIC parameter list, you state the variable name and that declares it as a parameter.

By reference, you can't give an expression to pass to the ML routine; only a variable name. Your ML routine must first find the variable in memory by reading the name from the parameter list then searching for it. Then, with the address of the variable in Page Zero, it can read or write to the variable, as you wish. Since this is exactly what Applesoft itself does to reference variables, you can just use its PTRGET routine to lookup your referenced parameters.

A referenced parameter is a variable and PTRGET reads the variable name pointed to by TXTPTR. It then looks it up in the variable storage area and returns its address. If it can't find it, then it creates a new variable with that name; in either case it returns with the address. TXTPTR is returned, pointing to the next character following the

variable name in program text, as expected. The address of the variable itself is in Page Zero at VARPNT ($83.84) and in the registers: high byte in Y-reg, low byte in A-reg. Almost all referenced parameters are found using PTRGET.

If you wanted to reference an entire array of variables and not just a single entry, then another routine called GETARYPT will do that. It fetches the address of the beginning of the array, where its name is encoded, in the array storage area. The address is in LOWTR ($9B.9C) when it returns to you, and you'll have to calculate your own way through the entries. GETARYPT is for entire arrays only, such as you would access if you wrote sorts or matrix arithmetic.

Once you have your variable pointer in Page Zero, the way you fetch and replace the variable depends on whether you are working with a number or with a string. The key is the pointer in VARPNT and that was found by PTRGET.

Immediately after a JSR PTRGET, the address of the pointer is in the Y-reg and the A-reg. If you then JSR MOVMF, the FP variable will be moved to the FAC in Page Zero. This is how you fetch an FP variable:

```
JSR  PTRGET  ; reference pointer
JSR  MOVFM   ; variable to FAC
```

If you are using the floating-point package (see Table 4-6) and want a second variable in ARG then the call is

```
JSR  PTRGET  ; reference pointer
JSR  CONUPK  ; variable to ARG
```

An FP number can be returned the same way. Fetch the pointer and use a move routine:

```
JSR  PTRGET  ; reference pointer
JSR  MOVMF   ; FAC to variable
```

All three of these routines — MOVFM, CONUPK, MOVMF — expect the memory address in the Y- and A-registers where PTRGET puts it. And the routines do the packing and unpacking as necessary.

With the routines to work between the FP registers and the variables in memory, you have all you need to pass FP numbers. Integers are al-

Table 4-6. Applesoft Floating-Point Math

Registers:		
FAC	$9D.A2	;FP accumulator, unpacked
ARG	$A5.AA	;FP argument for binary function
TEMP1	$93.97	;Packed format
TEMP2	$98.9C	;Packed format
TEMP3	$8A.8E	;Packed format
RND	$C9.CD	;Packed format, random number

Other Applesoft routines use this Page Zero space differently when not doing floating-point math. See memory map in Chapter Two.

Moves:		
MOVFM	$EAF9	;Unpacks (Y,A) to FAC
CONUPK	$E9E3	;Unpacks (Y,A) to ARG
MOVMF	$EB2B	;Packs FAC to (Y,A)
MOVAF	$EB63	;Copy FAC to ARG
MOVFA	$EB53	;Copy ARG to FAC
GETADR	$E752	;Fix FAC to LINNUM (unsigned)
GIVAYF	$E2F2	;Float (signed) A,Y to FAC
FOUT	$ED34	;String FAC to FBUFFR (STR$ function)
STROUT	$DB3A	;Print string at (Y,A)

Unary Functions:		
SGN	$EB90	;Sign(1,0, − 1) of FAC
ABS	$EBAF	;Absolute value of FAC
INT	$EC23	;Next largest integer
SQR	$EE8D	;Square root of FAC
LOG	$E941	;Natural logarithm(base e) of FAC
EXP	$EF09	;Exponent (base e) of FAC
RND	$EFAE	;Random number to FAC
COS	$EFEA	;Cosine (FAC in radians) to FAC
SIN	$EFF1	;Sine (FAC in radians) to FAC
TAN	$F03A	;Tangent (FAC in radians) to FAC
ATN	$F09E	;Arctan (FAC in radians) to FAC

Binary Functions:		
FMULTT	$E982	;ARG * FAC to FAC
FDIVT	$EA69	;ARG / FAC to FAC
FADDT	$E7C1	;ARG + FAC to FAC
FSUBT	$E7AA	;ARG − FAC to FAC
FPWRT	$EE97	;ARG exp FAC to FAC (ARG to the FAC power)

You should do the JSR MOVFM just before a binary function JSR. Otherwise, do a LDA FAC before the JSR.

<div style="border:1px solid">

Constants:

RND	$00C9	;Random number
¼	$F070	;
½	$EE64	
– ½	$E937	
1	E913	
10	$EA50	
SQR(½)	$E92D	
SQR(2)	$E932	
LOG(2)	$E93C	;Base ten
LOG(2)	$EEDB	;Base e
PI/2	$F063	
2*PI	$F06B	
– 32768	$E0Fe	

Use addr-hi in Y-reg and addr-lo in A-reg to fetch a constant with MOVFM or CONUPK.

Compares:

FCOMP	$EBB2	;Compare FAC with (Y,A)
		;Result in A = reg:
		1 if (Y,A) > FAC
		0 if (Y,A) = FAC
		$FF if (Y,A) < FAC
SIGN	$EB82	;Sets A-reg according to FACSGN
		1 if FAC > 0
		0 if FAC = 0
		$FF if FAC < 0
COMPARE	$DF6A	;Compare ARG with FAC according
		;to the code at $0016. FAC is
		;set to TRUE (1) or FALSE (0) on
		;return:

Set $0016	FAC is TRUE if
1	ARG > FAC
2	ARG = FAC
3	ARG < FAC
4	ARG ≥ FAC
5	ARG ≠ FAC
6	ARG ≤ FAC

</div>

most never passed by reference. Strings have several routines to manage them from variables, and the ones most often used are STRINI and MOVSTR.

If you want to read a referenced string, you must first fetch its descriptor from the string variable to Page Zero. The descriptor then

points to the string itself and you can read using indirect indexed addressing.

```
JSR   PTRGET        ; reference pointer
LDY   #0
LDA   (VARPNT),Y    ;string length
STA   DSCTMP        ; descriptor
INY
LDA   (VARPNT),Y    ; string addr-lo
STA   DSCTMP+1
INY
LDA   (VARPNT),Y    ; string addr-hi
STA   DSCTMP+2
```

The string can be read by LDA (DSCTMP + 1),Y where Y varies from zero to one less than the length in DSCTMP. The three bytes at DSCTMP are in Page Zero.

If you want to write a referenced string, it gets trickier. The way Applesoft does it is to create new string storage with the FRETOP and FRESPC pointers. Then the new string is put into the location pointed to by FRESPC and its new descriptor replaces its old one in the variable's descriptor. Wow! Let's take that step-by-step.

The routine that creates the new space is called STRINI. Just give it the length you need in the A-reg and it works with the pointers and returns you the new descriptor in DSCTMP. Then you copy the DSCTMP descriptor to the variable pointed to by VARPNT. Finally, use the DSCTMP descriptor as the length and destination address to copy your result string. It goes like this.

```
JSR   PTRGET        ; reference pointer
LDA   #LENGTH       ; of your new string
JSR   STRINI        ; make room up there
LDY   #0
LDA   DSCTMP        ; copy new descriptor
STA   (VARPNT),Y    ; to variable
INY
LDA   DSCTMP+1
STA   (VARPNT),Y
INY
LDA   DSCTMP+2
```

```
        STA   (VARPNT),Y
        LDA   #LENGTH     ; length of new string
        LDY   # STRING    ; string addr-hi
        LDX   # STRING    ; string addr-lo
        JSR   MOVSTR      ; copy string to (FRESPC)
```

The STRINI routine prepared your way for the copy by leaving FRESPC pointing to the new space in string storage. There are shortcuts you can make, especially with fixed length strings, but you need experience to get away with them. This procedure will work safely and satisfy most of your needs. Usually, STRING will be a work buffer you set up; use Page Two if you aren't doing any conflicting inputting at the same time.

When passing by reference, the key routine is PTRGET and the variable pointer is VARPNT. From there, you can work with numbers or strings according to the parameter. Passing by reference finds its greatest use in returning parameters from ML to BASIC.

CHAPTER FIVE

Integer BASIC

5.1 THE LANGUAGE

A summary of Integer BASIC statements is in Section 1.2 as well as this section. The commands and statements on pages 266 through 275 are the descriptions that appeared in the original *Apple II Reference Manual* (1978). Thanks to Apple Computer Inc., for permission to reproduce them.

Like Applesoft, Integer BASIC resides in the ROM memory area $D000.F7FF, between the hardware and the Monitor. The BASIC itself requires only 5K and begins at $E000. The three ROM chips in $E0, $E8, and $F0 sockets contain several utilities in addition to Integer BASIC — the Miniassembler and Floating-Point Utility Routines are useful to the Integer programmer. The $D8 socket is filled by the Programmer's Aid #1, which is a package of utilities including HIRES graphics. Some of the early Apple IIs may not have this chip retrofitted, but an Apple dealer can supply the chip along with a manual.

The x memory available to an Integer BASIC program is delimited by the HIMEM and LOMEM pointers in Page Zero (see Fig. 5-1). In a 48K system with DOS, this gives the range $800.95FF by LOMEM pointing to $800 and HIMEM pointing to $9600. When you type in a program by numbering statement lines, it is kept tokenized as program text below the HIMEM address. The program is in ascending sequence and a pointer called PP (point to program) gives the address

BASIC COMMANDS

Commands are executed immediately; they do not require line numbers. Most Statements (see Basic Statements Section) may also be used as commands. Remember to press Return key after each command so that Apple knows that you have finished that line. Multiple commands (as opposed to statements) on same line separated by a " : " are NOT allowed.

COMMAND NAME

AUTO *num*	Sets automatic line numbering mode. Starts at line number *num* and increments line numbers by 10. To exit AUTO mode, type a control X*, then type the letters "MAN" and press the return key.
AUTO *num1, num2*	Same as above execpt increments line numbers by number *num2*.
CLR	Clears current BASIC variables; undimensions arrays. Program is unchanged.
CON	Continues program execution after a stop from a control C*. Does not change variables.
DEL *num1*	Deletes line number *num1*.
DEL *num1, num2*	Deletes program from line number *num1* through line number *num2*.
DSP *var*	Sets debug mode that will display variable *var* every-time that it is changed along with the line number that caused the change. (NOTE: RUN command clears DSP mode so that DSP command is effective only if program is continued by a CON or GOTO command.)
HIMEM: *expr*	Sets highest memory location for use by BASIC at location specified by expression *expr* in decimal. HIMEM: may not be increased without destroying program. HIMEM: is automatically set at maximum RAM memory when BASIC is entered by a control B*.
GOTO *expr*	Causes immediate jump to line number specified by expression *expr*.
GR	Sets mixed color graphics display mode. Clears screen to black. Resets scrolling window. Displays 4Øx4Ø squares in 15 colors on top of screen and 4 lines of text at bottom.
LIST	Lists entire program on screen.
LIST *num1*	Lists program line number *num1*.
LIST *num1, num2*	Lists program line number *num1* through line number *num2*.

LOAD *expr.*	Reads (Loads) a BASIC program from cassette tape. Start tape recorder before hitting return key. Two beeps and a " > " indicate a good load. "ERR" or "MEM" FULL ERR" message indicates a bad tape or poor recorder performance.
LOMEM: *expr*	Similar to HIMEM: except sets lowest memory location available to BASIC. Automatically set at 2048 when BASIC is entered with a control B*. Moving LOMEM: destroys current variable values.
MAN	Clears AUTO line numbering mode to all manual line numbering after a control C* or control X*.
NEW	Clears (Scratches) current BASIC program.
NO DSP *var*	Clears DSP mode for variable *var*.
NO TRACE	Clears TRACE mode.
RUN	Clears variables to zero, undimensions all arrays and executes program starting at lowest statement line number.
RUN *expr*	Clears variables and executes program starting at line number specified by expression *expr*.
SAVE	Stores (saves) a BASIC program on a cassette tape. Start tape recorder in record mode prior to hitting return key.
TEXT	Sets all text mode. Screen is formated to display alpha-numeric characters on 24 lines of 40 characters each. TEXT resets scrolling window to maximum.
TRACE˙	Sets debug mode that displays line number of each statement as it is executed.

* Control characters such as control X or control C are typed by holding down the CTRL key while typing the specified letter. This is similiar to how one holds down the shift key to type capital letters. Control characters are NOT displayed on the screen but are accepted by the computer. For example, type several control G's. We will also use a superscript C to indicate a control character as in X^C.

BASIC Operators

Symbol	Sample Statement	Explanation

Prefix Operators

( )	1∅ X= 4*(5 + X)	Expressions within parenthesis ( ) are always evaluated first.
+	2∅ X= 1+4*5	Optional; +1 times following expression.
-	3∅ ALPHA = -(BETA +2)	Negation of following expression.
NOT	4∅ IF A NOT B THEN 2∅∅	Logical Negation of following expression; ∅ if expression is true (non-zero), 1 if expression is false (zero).

Arithmetic Operators

↑	6∅ Y = X↑3	Exponentiate as in X^3. NOTE: ↑ is shifted letter N.
*	7∅ LET DOTS=A*B*N2	Multiplication. NOTE: Implied multiplication such as (2 + 3)(4) is not allowed thus N2 in example is a variable not N * 2.
/	8∅ PRINT GAMMA/S	Divide
MOD	9∅ X = 12 MOD 7 1∅∅ X = X MOD(Y+2)	Modulo: Remainder after division of first expression by second expression.
+	11∅ P = L + G	Add
-	12∅ XY4 = H-D	Substract
=	13∅ HEIGHT=15 14∅ LET SIZE=7*5 15∅ A(8) = 2 155 ALPHA$ = "PLEASE"	Assignment operator; assigns a value to a variable. LET is optional

Relational and Logical Operators

The numeric values used in logical evaluation are "true" if non-zero, "false" if zero.

Symbol	Sample Statement	Explanation
=	16Ø IF D = E THEN 5ØØ	Expression "equals" expression.
=	17Ø IF A$(1,1)= "Y" THEN 5ØØ	String variable "equals" string variable.
# or < >	18Ø IF ALPHA #X*Y THEN 5ØØ	Expression "does not equal" expression.
#	19Ø IF A$ # "NO" THEN 5ØØ	String variable "does not equal" string variable. NOTE: If strings are not the same length, they are considered un-equal. < > not allowed with strings.
>	2ØØ IF A>B THEN GO TO 5Ø	Expression "is greater than" expression.
<	21Ø IF A+1<B-5 THEN 1ØØ	Expression "is less than" expression.
>=	22Ø IF A>=B THEN 1ØØ	Expression "is greater than or equal to" expression.
<=	23Ø IF A+1<=B-6 THEN 2ØØ	Expression "is less than or equal to" expression.
AND	24Ø IF A>B AND C<D THEN 2ØØ	Expression 1 "and" expression 2 must both be "true" for statements to be true.
OR	25Ø IF ALPHA OR BETA+1 THEN 2ØØ	If either expression 1 or expression 2 is "true", statement is "true".

BASIC FUNCTIONS

Functions return a numeric result. They may be used as expressions or as part
of expressions. PRINT is used for examples only, other statements may
be used. Expressions following function name must be enclosed between two
parenthesis signs.

FUNCTION NAME

ABS *(expr)*	300 PRINT ABS(X)	Gives absolute value of the expression *expr*.
ASC *(str$)*	310 PRINT ASC("BACK") 320 PRINT ASC(B$) 330 PRINT ASC(B$(4,4)) 335 PRINT ASC(B$(Y))	Gives decimal ASCII value of designated string variable *str$*. If more than one character is in designated string or sub-string, it gives decimal ASCII value of first character.
LEN *(str$)*	340 PRINT LEN(B$)	Gives current length of designated string variable *str$*; i.e., number of characters.
PDL *(expr)*	350 PRINT PDL(X)	Gives number between 0 and 255 corres- ponding to paddle position on game paddle number designated by expression *expr* and must be legal paddle (0,1,2,or 3) or else 255 is returned.
PEEK *(expr)*	360 PRINT PEEK(X)	Gives the decimal value of number stored of decimal memory location specified by expression *expr*. For MEMORY locations above 32676, use negative number; i.e., HEX location FFF0 is -16
RND *(expr)*	370 PRINT RND(X)	Gives random number between 0 and (expression *expr* -1) if expression *expr* is positive; if minus, it gives random number between 0 and (expression *expr* +1).
SCRN(*expr1,* *expr2)*	380 PRINT SCRN (X1,Y1)	Gives color (number between 0 and 15) of screen at horizontal location designated by expression *expr1* and vertical location designated by expression *expr2* Range of expression *expr1* is 0 to 39. Range of expression *expr2* is 0 to 39 if in standard mixed colorgraphics display mode as set by GR command or 0 to 47 if in all color mode set by POKE -16304 ,0: POKE - 16302,0.
SGN *(expr)*	390 PRINT SGN(X)	Gives sign (not sine) of expression *expr* i.e., -1 if expression *expr* is negative, zero if zero and +1 if *expr* is positive.

BASIC STATEMENTS

Each BASIC statement must have a line number between Ø and 32767. Variable
names must start with an alpha character and may be any number of alpha-
numeric characters up to 1ØØ. Variable names may not contain buried any
of the following words: AND, AT, MOD, OR, STEP, or THEN. Variable names may
not begin with the letters END, LET, or REM. String variables names must end
with a $ (dollar sign). Multiple statements may appear under the same line number
if separated by a : (colon) as long as the total number of characters in the line
(including spaces) is less than approximately 15Ø characters
Most statements may also be used as commands. BASIC statements are executed
by RUN or GOTO commands.

NAME

CALL *expr*	1Ø CALL-936	Causes execution of a machine level language subroutine at <u>decimal</u> memory location specified by expression *expr* Locations above 32767 are specified using negative numbers; i.e., location in example 1Ø is hexidecimal number $FC53
COLOR= *expr*	3Ø COLOR=12	In standard resolution color (GR) graphics mode, this command sets screen TV color to value in expression *expr* in the range Ø to 15 as described in Table A. Actually expression *expr* may be in the range Ø to 255 without error message since it is implemented as if it were expression *expr* MOD 16.
DIM *var1 (expr1)* *str$ (expr2)* *var2 (expr3)*	50 DIM A(2Ø),B(1Ø) 6Ø DIM B$(3Ø) 7Ø DIM C (2) Illegal: 8Ø DIM A(3Ø) Legal: 85 DIM C(1ØØØ)	The DIM statement causes APPLE II to reserve memory for the specified variables. For number arrays APPLE reserves approximately 2 times *expr* bytes of memory limited by available memory. For string arrays - *str$ (expr)* must be in the range of 1 to 255. Last defined variable may be redimensioned at any time; thus, example in line is illegal but 85 is allowed.
DSP *var*	Legal: 9Ø DSP AX: DSP L Illegal: 1ØØ DSP AX,B 1Ø2 DSP AB$ 1Ø4 DSP A(5) Legal: 1Ø5 A=A(5): DSP A	Sets debug mode that DSP variable *var* each time it changes and the line number where the change occured.

NAME	EXAMPLE	DESCRIPTION
END	11Ø END	Stops program execution. Sends carriage return and "> " BASIC prompt) to screen.
FOR *var*= *expr1* TO*expr2* STEP*expr3*	11Ø FOR L=Ø to 39 12Ø FOR X=Y1 TO Y3 13Ø FOR I=39 TO 1 15Ø GOSUB 1ØØ *J2	Begins FOR...NEXT loop, initializes variable *var* to value of expression *expr1* then increments it by amount in expression *expr 3* each time the corresponding "NEXT" statement is encountered, until value of expression *expr 2* is reached. If STEP *expr3* is omitted, a STEP of +1 is assumed. Negative numbers are allowed.
GOSUB *expr*	14Ø GOSUB 5ØØ	Causes branch to BASIC subroutine starting at legal line number specified by expression *expr* Subroutines may be nested up to 16 levels.
GOTO *expr*	16Ø GOTO 2ØØ 17Ø GOTO ALPHA+1ØØ	Causes immediate jump to legal line number specified by expression *expr*.
GR	18Ø GR 19Ø GR: POKE -163Ø2,Ø	Sets mixed standard resolution color graphics mode. Initializes COLOR = Ø (Black) for top 4Øx4Ø of screen and sets scrolling window to lines 21 through 24 by 4Ø characters for four lines of text at bottom of screen. Example 19Ø sets all color mode (4Øx48 field) with no text at bottom of screen.
HLIN *expr1*, *expr2*AT*xpr3*	2ØØ HLIN Ø,39 AT 2Ø 21Ø HLIN Z,Z+6 AT I	In standard resolution color graphics mode, this command draws a horizontal line of a predefined color (set by COLOR=) starting at horizontal position defined by expression *expr1* and ending at position *expr2* at vertical position defined by expression *expr3* . *expr1* and*expr2* must be in the range of Ø to 39 and *expr1* < = *expr2* . *expr3* be in the range of Ø to 39 (or Ø to 47 if not in mixed mode).
Note:		HLIN Ø, 19 AT Ø is a horizontal line at the top of the screen extending from left corner to center of screen and HLIN 2Ø,39 AT 39 is a horizontal line at the bottom of the screen extending from center to right corner.

<u>IF</u> *expression*
<u>THEN</u> *statement*
	220 IF A > B THEN
	      PRINT A
	230 IF X=0 THEN C=1
	240 IF A#10 THEN
	      GOSUB 200
	250 IF A$(1,1)# "Y"
	      THEN 100
	Illegal:
	260 IF L > 5 THEN 50:
	      ELSE 60
	Legal:
	270 IF L > 5 THEN 50
	      GO TO 60
		If *expression* is true (non-zero) then
		execute *statement*; if false do not
		execute *statement*. If *statement*
		is an expression, then a GOTO *expr*
		type of statement is assumed to be implied.
		The "ELSE" in example 260 is illegal but
		may be implemented as shown in example 270.

<u>INPUT</u> *var1,*
  var2, str$
	280 INPUT X,Y,Z(3)
	290 INPUT "AMT",
	      DLLR
	300 INPUT "Y or N?", A$
		Enters data into memory from I/O
		device. If number input is expected,
		APPLE wil output "?"; if string input is
		expected no "?" will be outputed. Multiple
		numeric inputs to same statement may be
		separated by a comma or a carriage return.
		String inputs must be separated by a
		carriage return only. One pair of " " may
		be used immediately after INPUT to output
		prompting text enclosed within the quotation
		marks to the screen.

<u>IN#</u> *expr*	310 IN# 6 320 IN# Y+2 330 IN# 0	Transfers source of data for subsequent INPUT statements to peripheral I/O slot (1-7) as specified as by expression *expr*. Slot 0 is not addressable from BASIC. IN#0 (Example 330) is used to return data source from peripherial I/O to keyboard connector.
<u>LET</u>	340 LET X=5	Assignment operator. "LET" is optional

<u>LIST</u> *num1,*
  num2
	350 IF X > 6 THEN
	      LIST 50
		Causes program from line number *num1*
		through line number *num2* to be displayed
		on screen.

<u>NEXT</u> *var1,* *var2*	360 NEXT I 370 NEXT J,K	Increments corresponding "FOR" variable and loops back to statement following "FOR" until variable exceeds limit.
<u>NO DSP</u> *var*	380 NO DSP I	Turns-off DSP debug mode for variable
<u>NO TRACE</u>	390 NO TRACE	Turns-off TRACE debug mode

PLOT . *expr1, expr2*	400 PLOT 15, 25 400 PLT XV,YV	In standard resolution color graphics, this command plots a small square of a predefined color (set by COLOR=) at horizontal location specified by expression *expr1* in range 0 to 39 and vertical location specified by expression *expr2* in range 0 to 39 (or 0 to 47 if in all graphics mode) NOTE: PLOT 0 0 is upper left and PLOT 39, 39 (or PLOT 39, 47) is lower right corner.
POKE *expr1, expr2*	420 POKE 20, 40 430 POKE 7*256, XMOD255	Stores <u>decimal</u> number defined by expression *expr2* in range of 0 255 at <u>decimal</u> memory location specified by expression *expr1* Locations above 32767 are specified by negative numbers.
POP	440 POP	"POPS" nested GOSUB return stack address by one.
PRINT *var1, var, str$*	450 PRINT L1 460 PRINT L1, X2 470 PRINT "AMT=";DX 480 PRINT A$;B$; 490 PRINT 492 PRINT "HELLO" 494 PRINT 2+3	Outputs data specified by variable *var* or string variable *str$* starting at current cursor location. If there is not trailing "," or ";" (Ex 450) a carriage return will be generated. Commas (Ex. 460) outputs data in 5 left justified columns. Semi-colon (Ex. 470) inhibits print of any spaces. Text imbedded in " " will be printed and may appear multiple times.
PR# *expr*	500 PR# 7	Like IN#, transfers output to I/O slot defined by expression *expr* PR# 0 is video output not I/O slot 0.
REM	510 REM REMARK	No action. All characters after REM are treated as a remark until terminated by a carriage return.
RETURN	520 RETURN 530 IFX= 5 THEN RETURN	Causes branch to statement following last GOSUB; i.e., RETURN ends a subroutine. Do not confuse "RETURN" statement with Return <u>key</u> on keyboard.

TAB *expr*	530 TAB 24 540 TAB I+24 550 IF A#B THEN TAB 20	Moves cursor to absolute horizontal position specified by expression *expr* in the range of 1 to 40. Position is left to right
TEXT	550 TEXT 560 TEXT: CALL-936	Sets all text mode. Resets scrolling window to 24 lines by 40 characters. Example 560 also clears screen and homes cursor to upper left corner
TRACE	570 TRACE 580 IFN > 32000 THEN TRACE	Sets debug mode that displays each line number as it is executed.
VLIN *expr1, expr2* AT *expr3*	590 VLIN 0, 39AT15 600 VLIN Z,Z+6ATY	Similar to HLIN except draws vertical line starting at *expr1* and ending at *expr2* at horizontal position *expr3*.
VTAB *expr*	610 VTAB 18 620 VTAB Z+2	Similar to TAB. Moves cursor to absolute vertical position specified by expression *expr* in the range 1 to 24. VTAB 1 is top line on screen; VTAB24 is bottom.

PROGRAM TEXT	── HIMEM AT $4C.4D
	── PP AT $CA.CB
VARIABLES STORAGE	── PV AT $CC.CD
	── LOMEM AT $4A.4B
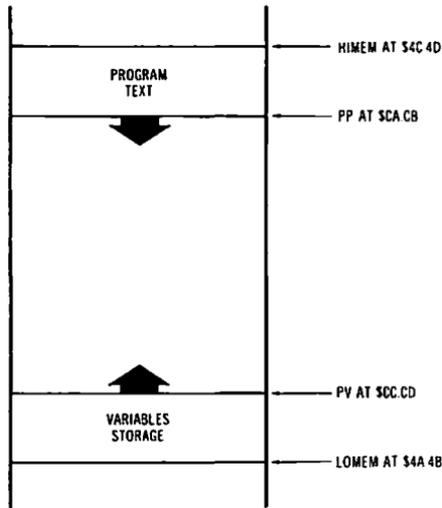
Fig. 5-1. Map of Integer BASIC program.

of the first instruction. So, program text resides between the addresses given by PP and HIMEM at the top end of user RAM.

When you RUN an Integer BASIC program, it builds its variables starting from the LOMEM address. A pointer called PV (point to variables) marks the end of the variable storage area. If adding a variable should ask PV to become bigger than PP you will get an "***MEM FULL ERROR." The variables include both numbers and strings between the addresses given by LOMEM and PV at the bottom end of user RAM.

Compare the memory maps of Integer and Applesoft. Both are resident in firmware in the $D000.F7FF region with the cold entry at $E000 and the warm entry at $E003. Both delimit RAM for BASIC use with a pointer pair: Applesoft with TXTTAB and FRETOP; Integer with LOMEM and HIMEM. Both tokenize their commands when building program text to save space and execution time. Both build variables during BASIC execution. And in both, the LOMEM: command changes the beginning of variable storage so that you can protect the HIRES screen(s) from encroachment. However, the maps are different and you use them differently to achieve coexistence with graphics and ML routines.

The big difference is where program text resides. Integer puts it at the top of memory, right where Applesoft kept its working strings.

Table 5-1. Integer BASIC Error Messages

Message	Description
*** SYNTAX ERR	Results from a syntactic or typing error.
*** >32767 ERR	A value entered or calculated was less than − 32767 or greater than 32767.
*** > 255 ERR	A value restricted to the range ∅ to 255 was outside that range.
*** BAD BRANCH ERR	Results from an attempt to branch to a nonexistent line, number.
*** BAD RETURN ERR	Results from an attempt to execute more RETURNs than previously executed GOSUBs.
*** BAD NEXT ERR	Results from an attempt to execute a NEXT statement for which there was not a corresponding FOR statement.
*** 16 GOSUBS ERR	Results from more than 16 nested GOSUBs.
*** 16 FORS ERR	Results from more than 16 nested FOR loops.
*** NO END ERR	The last statement executed was not an END.
*** MEM FULL ERR	The memory needed for the program has exceeded the memory size allotted.
*** TOO LONG ERR	Results from more than 12 nested parentheses or more than 128 characters in input line.
*** DIM ERR	Results from an attempt to DIMension a string array which has been previously dimensioned.
*** RANGE ERR	An array was larger than the DIMensioned value or smaller than 1 or HLIN, VLIN, PLOT, TAB, or VTAB arguments are out of range.
*** STR OVFL ERR	The number of characters assigned to a string exceeded the DIMensioned value for that string.
*** STRING ERR	Results from an attempt to execute an illegal string operation.
RETYPE LINE	Results from illegal data being typed in response to an INPUT statement. This message also requests that the illegal item be retyped.

But that is all right because Integer doesn't keep strings dynamically; they are kept within the variables themselves. So, unlike Applesoft programs that always load at the bottom, Integer programs load at the top of memory so as to end just before the HIMEM address.

In Applesoft, variable storage begins at the address in VARPNT, normally set to the end of program text and changed by the HIMEM: command. In Integer, variable storage begins at the address in the

LOMEM pointer that is changed from $800 by the LOMEM: command. The only problem is that LOMEM: is illegal in a program — it must be given in command mode. So, a program must use POKEs instead to set LOMEM and PV to the variable start address before any variables are referenced in the program. For example, protecting HIRES1 by setting LOMEM and CV to $4000 is done by

```
30000 POKE  75,64 :REM set LOMEM-hi to $40
30010 POKE205,64 :REM set PV-hi to $40
```

at the beginning of the program mainline. Only the high bytes need be set because LOMEM-lo is normally zero and CV is set to LOMEM at RUN time.

Like Applesoft memory mapping, Integer memory usage can be optimized if you work from the normal memory map and sketch out what you want first. Integer is a little easier because it has fewer parts: only program text and variables. So, by knowing where things are you can use the same techniques.

5.2.2 Variables

Variables that are kept between the LOMEM and PV addresses during a BASIC program run are random-length records. Each variable has a name and that name can be any length up to 100 characters. Integer is not restricted to two-character names as is Applesoft. Each name identifies its variable, so each variable record may have a different length. Integer BASIC manages this by linking each variable to the next, using a link pointer. Remember, that was the way program text was kept; in Integer BASIC, variables are kept the same way.

There are four kinds of variables: simple numbers, characters, DIMensioned numbers and DIMensioned strings. If a reference to a new variable is made in your program without a DIMension declaration, then it is created as a simple variable of a number or a single character string.

All variables are composed of a record having four fields: *variable name, display attribute, link,* and *data.* The length of the record depends upon the length of the name.

The variable name, abbreviated VN, contains the name you give it in negative ASCII. Unlike normal seven-bit ASCII, these characters all have their bit 7s equal to one so that the characters in Integer are

Table 5-2. Integer Variable Names and Tokens

Integer	Description
Variable name	In 8-bit ASCII, $80 to $FF. First character "A" to "Z", $C1 to $DA String name "$" tokened as $40
Tokens	All $00 to $7F. See Table 5-3.
Integer constant	3 bytes: flag($B0 to $B9), low, high
String constant	Left quote ($28), 8-bit ASCII string, right quote ($29)
REM statement	Begins with REM token $5D, ends with $01.

between $80 and $FF instead of $00 to $7F. In this scheme, "A" is $C1, "B" is $C1, and so on. So, a variable name consists of a string of negative ASCII characters. If the name is that of a string, the name has an extra byte, $40, at the end. The $40 is Integer's token code for "$" and appears in its place in string names.

The end of the variable name is marked by the display *attribute byte,* $00 or $01. Being positive and therefore not a negative ASCII character, and having only one of two values, it is easily spotted. What it does is tell the run-time interpreter to display any variable that has it set to one. This is how the DSP command works when it sets the display attribute in the variable. Then, NODSP clears it to zero. Normally, all variables have their display attributes zeroed to suppress automatic display by the run-time interpreter as your BASIC program executes.

Following the DSP byte in the variable is the link field called NVA, *Next Variable Address.* This contains the absolute address of the next variable. This way all variables are linked in a list for searching.

For simple variables, the data field is two bytes long. A number appears in the address format of low byte followed by high byte. A string consists of one negative ASCII character followed by a *sentinel byte* in the $00.7F range, usually $1E. If it is a null, the sentinel appears first followed by a zero, $1E $00.

Suppose you had a program like:

```
10 NUM = 128
20 CHAR$ = "B"
30 END
```

After running and entering the monitor (CALL-151), you could dump the variables to get

```
0800: CE D5 CD 00 08 08 80 00
0808: C3 C8 C1 D2 40 00 12 08 C2 1e
```

where LOMEM points to $0800. Reading, the CE D5 CD are negative ASCII for the name of the first variable, NUM. Next, the zero ends the name by flagging the DSP off. The link is to the next variable address at $0808. And finally, the data of the first variable is the number $0080, which is 128. The next variable at the address $0808 is a string, because the four negative ASCII characters C3, C8, C1, and D2 are followed by a $40 to give "CHARS" as the variable name. Then the display byte is off (zero). Next variable address is $0812 in the following two bytes. And finally the variable's data — the character "B" as $C2 and suitably terminated with a $1E.

DIMensioned variables work much the same way (see Fig. 5-2). Only the DATA field is longer. What the DIM does is reserve one byte
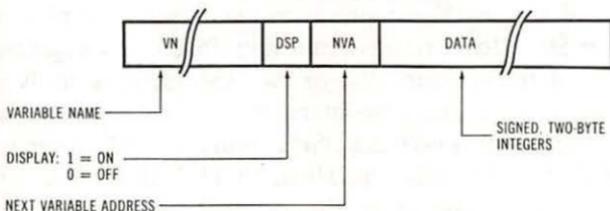
Fig. 5-2. Integer number variable.

for each character if it is a string or two bytes for each additional number if it is numeric. So, a string must be dimensioned with the largest number of characters expected to be contained by the variable in the life of the program, up to 255 (see Fig. 5-3). When not full, the $1E
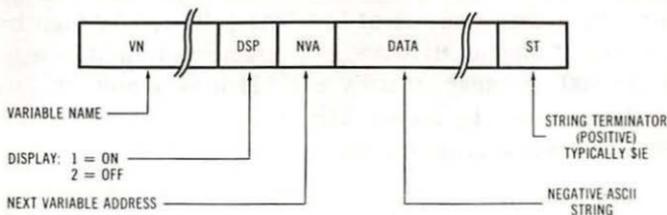
Fig. 5-3. Integer string variable.

marks the end of the current string; you can find it by using the LEN function. Any element within a string can be found as a substring by referencing the string with two subscripts, like

$$A\$ = B\$(4,7)$$

which assigns the fourth through seventh characters in A$ to B$. Concatenation is done by assigning to the last-plus-one position in a string, like:

$$L = LEN(BIG\$)$$
$$BIG\$(L+1) = SUFFIX\$$$

which copies the string from SUFFIX$ beginning at one byte beyond the last character in BIG$. This leaves BIG$ longer by the length of SUFFIX$.

The fact that you can only DIMension once restricts your expressions to single dimension arrays. But, suppose you wanted a 10 by 12 array. You can reserve enough space to work with in the variable by

$$DIM\ ARRAY(\ 9*11\ )$$

where the subscripts you use begin at zero. If you had subscripts X and Y, where X ranged from zero to nine and Y ranged from zero to eleven, you could then address any element in ARRAY as:

$$element = ARRAY(\ X + 12*Y\ )$$

As far as BASIC is concerned, you are subscripting with only one expression, but to you it is like having two subscripts operating, X and Y.

By using tricks like this, you can overcome many of the restrictions of Integer BASIC.

5.2.3 Program Text

Program text is kept in the highest chunk of memory that the HIMEM pointer allows. Each line is stored as a single record and may contain several statements, separated within the line by ":" — colons. The records are stored in line number order, in ascending sequence. The CC pointer gives the address of the first line, and HIMEM points to one location beyond the end of the last line. The program text consists of records of one line each in memory pointed from CC to HIMEM.

The lines are in *linked records,* so they are read as a linked list. The link field for Integer BASIC text is not absolute, however, but relative. The first byte in each record gives the length of the record (see Fig. 5-4). By adding that length to the address of the current record,
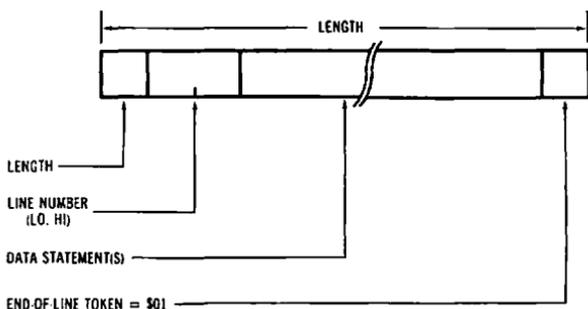
Fig. 5-4. Integer BASIC program line.

you get the address of the next record. In this way, the length acts as a link field in the record to connect each following line to its predecessor.

Next in the record is the line number in low-byte/high-byte format. The line number is always a two-byte field.

The contents of the line follows the line number beginning at the third byte of the record. Statements are separated by colons tokenized as $03 in lines with multiple statements. The ends of all lines are marked by a $01 sentinel token. Then within each statement appears a mixture of negative ASCII characters and positive-valued tokens. The tokens are usually commands (verbs) and the characters, labels (nouns). Of these, constants and REM statements will take a little study before you can read a dump of program text easily.

REM statements have $5D which is the REM token followed by a bunch of negative ASCII characters. Together with their line numbers, they make good reference points when you are scanning through a dump. Use them to find the *neighborhood* of your target lines when searching program text.

String constants appear as a string of negative bytes with $28 at the start and $29 at the end. These are the tokens for opening and closing quotes that Integer encodes differently. Look for the $28 . . . $29 pattern.

Integer constants are three bytes long; first byte is usually $B0. This flag is negative and may be any byte from $B0 to $B9. Look for this flag followed by the value in low-byte/high-byte order in hex.

Otherwise, you should be able to cruise through a dump of program text armed with a listing, the Integer Token Table 5-3 and the Negative ASCII Table 5-4.

Table 5-3. Integer BASIC Text Tokens

Hex		Hex		Hex	
00	start line	30	SGN	60	IF
01	end line	31	ABS	61	PRINT
02	internal use	32	PDL	62	PRINT
03	:	33		63	PRINT
04	LOAD	34	(	64	POKE
05	SAVE	35	+	65	,
06	CON	36	–	66	COLOR =
07	RUN	37	NOT	67	PLOT
08	RUN	38	(	68	,
09	DEL	39	=	69	HLIN
0A	,	3A	#	6A	,
0B	NEW	3B	LEN(	6B	AT
0C	CLR	3C	ASC(	6C	VLIN
0D	AUTO	3D	SCRN(	6D	,
0E	,	3E	,	6E	AT
0F	MAN	3F	(	6F	VTAB
10	HIMEM:	40	$	70	= string
11	LOMEM:	41		71	= number
12	+	42	(	72	)
13	–	43	,	73	
14	*	44	,	74	LIST
15	/	45	;	75	,
16	=	46	;	76	LIST
17	#	47	;	77	POP
18	> =	48	,	78	NODSP
19	>	49	,	79	NODSP
1A	< =	4A	,	7A	NOTRACE
1B	<>	4B	TEXT	7B	DSP
1C	<	4C	GR	7C	DSP
1D	AND	4D	CALL	7D	TRACE
1E	OR	4E	DIM	7E	PR#
1F	MOD	4F	DIM	7F	IN#
20	∧	50	TAB		
21		51	END		
22	(	52	INPUT		
23	,	53	INPUT		
24	THEN	54	INPUT		
25	THEN	55	FOR		
26	,	56	=		
27	,	57	TO		
28	" begin	58	STEP		

29	" end	59	NEXT
2A	(	5A	,
2B		5B	RETURN
2C		5C	GOSUB
2D	(	5D	REM
2E	PEEK	5E	LET
2F	RND	5F	GOTO

Table 5-4. The 8-bit ASCII Character Set (negative-ASCII)

DEC	HEX		DEC	HEX		DEC	HEX		DEC	HEX	
128	80	NUL	160	A0	SP	192	C0	@	224	E0	
129	81	SOH	161	A1	!	193	C1	A	225	E1	a
130	82	STX	162	A2	"	194	C2	B	226	E2	b
131	83	ETX	163	A3	#	195	C3	C	227	E3	c
132	84	EOT	164	A4	$	196	C4	D	228	E4	d
133	85	ENQ	165	A5	%	197	C5	E	229	E5	e
134	86	ACK	166	A6	&	198	C6	F	230	E6	f
135	87	BEL	167	A7	'	199	C7	G	231	E7	g
136	88	BS	168	A8	(	200	C8	H	232	E8	h
137	89	HT	169	A9	)	201	C9	I	233	E9	i
138	8A	LF	170	AA	*	202	CA	J	234	EA	j
139	8B	VT	171	AB	+	203	CB	K	235	EB	k
140	8C	FF	172	AC	,	204	CC	L	236	EC	l
141	8D	CR	173	AD	−	205	DC	M	237	ED	m
142	8E	SO	174	AE	.	206	CE	N	238	EE	n
143	8F	SI	175	AF	/	207	CF	O	239	EF	o
144	90	DLE	176	B0	0	208	D0	P	240	F0	p
145	91	DC1	177	B1	1	209	D1	Q	241	F1	q
146	92	DC2	178	B2	2	210	D2	R	242	F2	r
147	93	DC3	179	B3	3	211	D3	S	243	F3	s
148	94	DC4	180	B4	4	212	D4	T	244	F4	t
149	95	NAK	181	B5	5	213	D5	U	245	F5	u
150	96	SYN	182	B6	6	214	D6	V	246	F6	v
151	97	ETB	183	B7	7	215	D7	W	247	F7	w
152	98	CAN	184	B8	8	216	D8	X	248	F8	x
153	99	EM	185	B9	9	217	D9	Y	249	F9	y
154	9A	SUB	186	BA	:	218	DA	Z	250	FA	z
155	9B	ESC	187	BB	;	219	DB	⊏	251	FB	{
156	9C	FS	188	BC	<	220	DC	\	252	FC		
157	9D	GS	189	BD	=	221	DD	⊐	253	FD	}
158	9E	RS	190	BE	>	222	DE	↑	254	FE	~
159	9F	US	191	BF	?	223	DF	__	255	FF	DEL

DEL	delete	VT	vertical tab
NUL	null character	DC1	device control
SOH	start of header	DC2	device control
STX	start of text	DC3	device control
ETX	end of text	DC4	device control
EOT	end of transmission	NAK	negative acknowledge
ENQ	enquiry	SYN	synchronous idle
ACK	acknowledge	ETB	end transmission block
BEL	bell	CAN	cancel
BS	back space	EM	end of medium
HT	horizontal tab	SUB	substitute
LF	line feed	ESC	escape
FF	form feed	FS	file separator
CR	carriage return	GS	group separator
SO	shift out	RS	record separator
SI	shift in	US	unit separator
DLE	data link escape	SP	space

5.2.4 Tricks

To get around some of Integer BASIC's shortcomings, programmers have long used a few standard tricks. These include making VAL and CHR$ functions, making illegal LOMEM: statements, and combining BASIC and ML using a method called *pack and load*.

Two functions from Applesoft that would be very useful in Integer BASIC programming are the CHR$ and VAL functions. CHR$ returns a string variable of one character having the ASCII value of the argument of the function. VAL is the inverse: the ASCII value is returned when a single character is given as its string argument. If a number, N, has a value from zero to 128, then

$$N = VAL(CHR\$(N))$$

because VAL and CHR$ are inverses. Similarly,

$$A\$ = CHR\$(VAL(A\$))$$

for the same reason.

You can have CHR$ and VAL functions in Integer by using a little trick. Make the very first variable you declare in your program a single character string:

30100 CHR$ = "A"

When the program RUNs, it creates CHR$ in variable storage at LOMEM. If LOMEM is $800, then it will look like

0800: C3 D8 D2 40 00 09 08 C1 1E

in memory. The "A" you assigned is at $807 as hex C1. So in your program you can change CHR$ anytime with a number by

POKE 2055,VAL

where VAL is any byte value between 128 and 255. This gives you the character having that value in CHR$.

If you want the number from the CHR$ character, just

VAL = PEEK(2055)

It is that simple.

There are two restrictions to that trick. First, LOMEM must be $800 for the 2055 address to work. And second, the value you POKE must always be between 128 and 255 so Integer knows that it is a character. If you don't like these hangups, then use two other smart statements that avoid them:

POKE 7+PEEK(74)+256*PEEK(75),VAL+128*(VAL<128)

for the CHR$ function, and

VAL = PEEK(7+PEEK(74)+256*PEEK(75))−128

for the VAL. This way, LOMEM is used to find the location of the CHR$ character in variable storage and the ASCII value of the character is kept in VAL.

One of the differences between Integer and Applesoft BASICs is that Integer parses your statements much more rigorously in order to tokenize them at the time you type them in. With much of the parsing in the tokens, Integer BASIC text then executes much faster than does Applesoft. But many commands are not allowed by Integer because Integer won't parse them into deferred statements like Applesoft will.

So, HIMEM:, LOMEM, LIST, RUN, and other commands are illegal
in Integer BASIC.

Sometimes you need an illegal command in your program. Like a
LIST to capture it to a text file, for instance. The trick to making il-
legal statements is simple enough. As an example, suppose you wanted
to

<div align="center">

100 LOMEM: 4096

</div>

in your program instead of the usual POKEs. Since the statement is il-
legal, write a similar statement; any one that has a command followed
by a number like LOMEM: but one that's legal. Suppose you choose

<div align="center">

100 PRINT 4096

</div>

which is legal. Enter the monitor and find line 100 as hex 64 00
(low/high hex format):

<div align="center">

07 64 00 62 B0 00 10 01

</div>

The PRINT token is the hex 62. Now, replace it with the LOMEM:
token which is hex 11. Your line will look like

<div align="center">

07 64 00 11 B0 00 10 01

</div>

now, in memory. Return to BASIC and list it. You should see

<div align="center">

100 LOMEM: 4096

</div>

which will execute properly at run time.

Since the days of tape, programmers have been putting their ML
files inside Integer BASIC programs to make one file to copy and
load. You can tell when this trick is used, when the program you load
cannot be LISTed: and it gives garbage instead. To understand what's
happening in such programs or to pack your own single file programs,
here's how it is done. The method is called *pack and load*. See Fig. 5-5.

STEP ONE: Start with a good, working program in two files, one
   BASIC and one binary. The binary file BLOADs at $800 and
   LOMEM protects it because of two instructions:
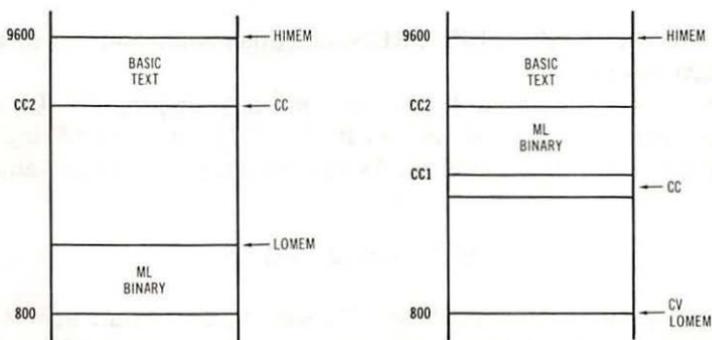
ORIGINAL

WHEN **BASIC** IS WRITTEN,
POINTER **CC** TELLS
WHERE IT STARTS.
WHEN **RUN**, **LOMEM** IS
RAISED TO PROTECT **ML**.

ML LOADED AND
POINTER **CC** CHANGED
TO INCLUDE THE
ML IN THE **BASIC**
LOAD AND SAVES.
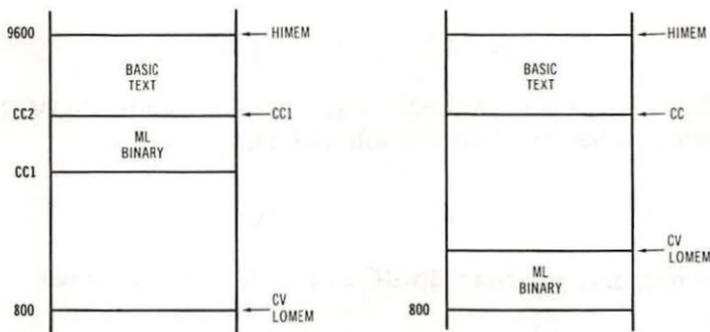
WHEN **RUN**, POINTER
CC IS RESTORED TO
BASIC ONLY, LEAVING **ML**.

TO PROTECT **ML** FROM
ENCROACHMENT COPY
IT DOWN TO $800 AND
CHANGE **LOMEM** AND **CV**.

Fig. 5-5. Integer BASIC pack and load.

```
30200   PRINT"BLOAD PROGNAME.BIN"
30250   POKE  75,high :REM sets LOMEM
30260   POKE205,high :REM sets PV
```

where the cntrl/D is hidden at the beginning of the quote. The
byte value *high* is the next page number following your ML rou-
tines, as usual.

STEP TWO: Replace the BLOAD statement line with

```
30200   POKE 60,low: POKE 61,high: REM PP1
30210   POKE 62,low: POKE 63,high: REM PP2
30220   POKE 64,00  : POKE 65,08   : REM $800
30230   CALL −468: REM MOVE PP1.PP2 to $0800
```

Just use zero byte values for the *high* and *low* values of the PP1 and PP2 values.

STEP THREE: Enter the Monitor, CALL − 151. Find the pointer value in PP ($CA.CB) and note it as PP2. Then subtract the length of your ML file from this PP2 value to get the value of PP1. Then, 3D0G to BASIC again.

STEP FOUR: Change the POKEs you entered in STEP TWO by giving them the address bytes of PP1 and PP2.

STEP FIVE: Load the ML file that normally resides at $800 but force the load to the PP1 address. Example:

BLOAD PROGNAME.BIN,A$7423

where $7423 is the value of PP1 in this mythical example.

STEP SIX: Enter the Monitor, CALL − 151. Change PP at $CA.CB from PP2 to PP1. Return to BASIC, 3D0G.

STEP SEVEN: Add one line to the program, line zero.

0 POKE 202,low; POKE 203,high: GOTO 10

where 10 is the first line of your program. The address you POKE as *low* and *high* is PP2. This line restores PP2 when the program is RUN.

STEP EIGHT: Now, SAVE it to disk. Don't use the same name as your Integer original; you need it as your source for any further changes. The file you SAVE is a *pack and load* file for RUNning only.

If you want to make any changes, do so in the two original files. Then rebuild the pack and load file using these steps again.

For reference, see Integer BASIC locations in Table 5-5.

Table 5-5. Some Useful Integer BASIC Locations

Label	Hex	Dec	Description
LOMEM	4A.4B	74.75	lowest RAM, start of variables
HIMEM	4C.4D	76.77	highest RAM, end of program text
GOTOA	C6.C7	198.199	address of line for GOTO
PP	CA.CB	202.203	pointer to program text start
PV	CC.CD	204.205	pointer to variables end
VAL	CE.CF	206.207	next line number
RUNMODE	D9	217	(+)ve is immediate; (−)ve is run
PR	DC.DD	220.221	current line number
LNA	E4.E5	228.229	line number address
CNTLB	E000	− 8192	cold entry point
CNTLC	E003	− 8189	warm entry point
	E3E3	− 6997	displays current line number, PR
HEXDEC	E51B	− 6885	displays A-reg/X-reg in decimal
LINADR	E56D	− 6803	at (LNA), finds line number, VAL
GOVAL	E85E		GOTO line number, VAL
GOLNA	E867	− 6041	GOTO line at address, GOTOA
LAM	E88A		return from Monitor command CALL

5.3 UTILITIES

5.3.1 CALL Extensions

With the Programmer's Aid #1 chip, you can CALL several useful routines.

You can renumber an Integer BASIC program in whole or in part. To renumber the complete program, type

```
CLR
START = 1000
STEP = 10
CALL − 10531
```

if you want the new numbers to be 1000, 1010, 1020, etc. Any references in GOSUBs and GOTOs will be changed if they are simple numbers. For instance,

```
GOSUB 215
```

will be changed to the new line number, but

```
GOTO 35 + 100
```

and

GOSUB 1000*N

won't be. So for complex expressions you'll have to go through your renumbered program and correct them yourself.

To renumber only a portion of your program, type

CLR
START = 3000
STEP = 5
FROM = 360
TO = 480
CALL – 10521

where START and STEP again refer to the new line numbers. The FROM and TO tell the renumbering routine exactly which part of your program you want renumbered. You can't renumber with numbers within the FROM/TO range as the routine will quit with an error message. Keep your old line numbers (FROM/TO) disjoint from your new line numbers (START/STEP).

When using tape to store Integer BASIC programs, you can append one program to another. This will give you a new program with all the lines of each of the two original programs in it. This appending routine is great for using tape libraries of subroutines. With various subroutines on tape files, a program can append those needed to itself without having to type them in each time. For this to work, your Integer programs must all have high line numbers and each subroutine must have its own block of line numbers below those of the program.

Here's how it works. Suppose you had a tape of ten subroutines from lines 100 to 199, 200 to 299, 300 to 399, etc. And suppose your program was written between lines 30000 and 32767. Such a system could use the append feature to include any or all of the subroutines in the program. With the program in memory and the tape positioned at the highest-numbered of the wanted subroutines, type

CALL = 11076

to load the subroutine. Then position the tape to the subroutine with the next lowest line numbers and repeat the load procedure. As long as

the routine on tape has lower line numbers than the lowest in your program, the CALL = 11076 will load and append the subroutine to the beginning of your program. Of course, you will wait until adding the

<div align="center">10 GOTO 30000</div>

line to your final program.

Whenever you save an Integer program or subroutine to tape, you probably reload it to make sure that it is intact. For short routines this is the best way, but for long programs checking out the entire listing for possible errors becomes impractical. Fortunately you have an Integer BASIC tape verify routine that can do the job for you.

To do a tape verify of an Integer program, use

<div align="center">CALL = 10955</div>

instead of the LOAD command to reload the program from tape. Do this immediately after you SAVEd it, while the copy is still in memory. The verify routine will load and verify the program on tape with the copy in memory. Two audible beeps tell you that the verify went all right; one beep and the ERR message indicates that the verify did not work. Resave the program and try again.

You can make sounds easily from Integer BASIC. Your programs can even play musical tunes. Here is the call sequence to put in your program initialization:

<div align="center">TIMBRE = 765:TIME = 766:PITCH = 767
MUSIC = − 10473</div>

To sound a note, POKE values into TIMBRE, TIME, and PITCH, then CALL MUSIC. Normal notes have a timbre of 32. The scale is closely chromatic so that PITCH can be POKEd as follows:

POKE PITCH, 1	for the lowest note
POKE PITCH, 13	for same note, 2nd octave
POKE PITCH, 25	for same note, 3rd octave
POKE PITCH, 37	for same note, 4th octave
POKE PITCH, 49	for same note, 5th octave

So, to play a middle note you would use:

```
POKE TIMBRE, 32
POKE TIME, 100
POKE PITCH, 25
CALL MUSIC
```

If you just want game sounds, play around, especially with the timbre. If you want tunes, try different notes to get the *flavor* of your tune in one tonic note. Then use the intervals from the tonic to count up and down the tempered scale. If this sounds obscure, get a music student to help you translate notes to chromatic scale intervals.

To use HIRES graphics in Integer BASIC, you have to set the very first variables in your program as follows:

$$\text{CLR:X0} = \text{Y0} = \text{COLR}$$

If you use shape tables, you must follow immediately with

$$\text{SHAPE} = \text{ROT} = \text{SCALE}$$

The HIRES routines expect these variables to be declared exactly like this and first in the variable storage area, at the memory pointed to by LOMEM. After declaring these variables, your program can continue with any other variables it needs. In particular, you will want some or all of the following:

INIT = − 12288	works like Applesoft's GR
CLEAR = − 12274	sets screen to black
BKGND = − 11471	sets screen to a color
POSN = − 11527	moves HIRES cursor
PLOT = − 11506	moves cursor, plots point
LINE = − 11500	draws while moving cursor
DRAW = − 11465	draws a shape
DRAW1 = − 11462	draws a shape at cursor
SHLOAD = − 11355	loads shape table

You will POKE parameters for COLR, X0, and Y0 to set the cursor and current drawing color. These COLR values may be given names by typing

BLACK = 0: LET GREEN = 42: VIOLET = 85: WHITE = 127
BLACK2 = 128: ORANGE = 170: BLUE = 213: WHITE2 = 255

Note you must use LET for GREEN to keep Integer BASIC from interpreting the GR as a command. X0 must be set to a value in the 0...279 range while Y0 must be set to a value in the 0...191 range.

The HIRES routines normally plot on HIRES1 screen. If you want HIRES2 plots, then:

```
POKE   806,64       for HIRES 2 plotting
POKE – 16299,0      for HIRES 2 display
POKE – 16302,0      for full-screen display
```

To return again to HIRES1 you must

```
POKE   806,32       for HIRES 1 plotting
POKE – 16300,0      for HIRES 1 display
```

And, to reset the screen to four lines of text at the bottom

```
POKE – 16301,0      for mixed graphics/text
```

While HIRES graphics techniques are covered in Chapter Six, they are intended for the Applesoft user. If you are using Integer BASIC, Chapter Six is still useful, but you will use the Integer routines defined above instead of the built-in commands of Applesoft. They do the same thing. For clarity, here are the Integer BASIC HIRES routines call sequences.

To invoke the HIRES package and initialize its parameters, use the instruction:

CALL INIT

To re-clear the screen to black at any future time, use

CALL CLEAR

instead. If you want the screen to be cleared to a background color other than black, you must first set COLR, for example,

COLR = VIOLET
CALL BKGND

will set your screen to violet.

To plot single points, you must set the cursor and the drawing color. For example,

COLR = WHITE:X0 = 140:Y0 = 95
CALL PLOT

puts a single white dot in the middle of the screen. To plot a line, first you must be sure that the cursor is at the beginning point. Then set the cursor to the end point and use LINE to plot the line:

X0 = 0:Y0 = 0:CALL POSN
X0 = 279:Y0 = 191:CALL LINE

draws a diagonal line from upper left to lower right in the current drawing color. The cursor remains at the end position, so you could continue without CALLing POSN again if you were drawing a polygon.

The remaining routines are used with shape tables. To load a shape table,

CALL SHLOAD

and run your tape player at the same time. The shape table loads at $0800 and locations $0328.0329 contain the pointer. If you do a binary load from disk, set $0328 and $0329 to the low byte and high byte of your shape table's starting address and don't use SHLOAD. If you load to $0800, you can set

LOMEM:16384   protect shapes and HIRES1

or

LOMEM:24576   protect shapes, HIRES 1 & 2

to protect the $800.1FFF area for your shape tables.

To draw a shape, set the cursor, the color, and the shape parameters — SHAPE, ROT, and SCALE.

SHAPE = 1: ROT = 0: SCALE = 1
CALL DRAW

This example draws the first shape in the table whose address is in $328.329. The shape is not rotated (zero) and its size is not increased (scale of one). X0, Y0, and COLR were used and X0, Y0 remain unchanged. If you want to draw another shape but at the cursor instead of the X0, Y0 point, then CALL DRAW1 instead.

Remember when using shapes that you cannot use too large a SCALE because the shape routines don't do clipping. Make sure your shape won't try to be drawn off the screen. A scale of 2 is twice normal size, 4 is four times normal size, and so on. Rotation can be from 0 to 63. The shape is rotated clockwise, so that ROT = 16 gives you 90 degrees, ROT = 32 gives you 180 degrees, ROT = 48 gives you 270 degrees, and ROT = 0 gives you zero degrees or no rotation. ROT = 64 is the same as ROT = 0, so just use the range 0. . .63 for ROT to avoid confusion.

5.3.2 Monitor Extensions

Like the extra CALLs, the Programmer's Aid #1 gives you some very useful routines that you access from the Monitor.

Just as there is a utility that verifies an Integer BASIC program, there is a utility that verifies a binary file saved to a tape. To use it after a W command, just rewind the tape and position it as if you were going to use the R command to read it. Then type

addr1.addr2
D52EG

to set the ctrl/Y hook. Then type

addr1.addr2(ctrl/Y)(return)

for the same address range that you saved. Play the tape and it will be verified.

If the verify routine finds a discrepancy, it will give an audible beep and give you an ERR message. Otherwise the program will finish normally.

Here's how to test the RAM of a 48K Apple from Integer BASIC. Enter the Monitor with a CALL − 151. Engage the RAM test routines by

<div align="center">

D5BCG

</div>

which sets the ctrl/Y feature.

To test the first block of 16K of memory, type

<div align="center">

400.4(ctrl/Y)(return)
800.8(ctrl/Y)(return)
1000.10(ctrl/Y)(return)
2000.20(ctrl/Y)(return)

</div>

Each line tests a chunk of memory and results in an error message in case of RAM failure. It may take time, and the Monitor's asterisk will return when the test is finished.

To test the second block of 16K of RAM, type

<div align="center">

4000.40(ctrl/Y)(return)

</div>

And for the third block, the command is

<div align="center">

8000.40(ctrl/Y)(return)

</div>

If you want to test the boundaries between each block, the commands are

<div align="center">

3000.20(ctrl/Y)(return)
7000.20(ctrl/Y)(return)

</div>

A good test to make of a 48K system is this one-liner:

<div align="center">

N 400.4(ctrl/Y)800.8(ctrl/Y)1000.10(ctrl/Y)2000.20(ctrl/Y)
3000.20(ctrl/Y)4000.40(ctrl/Y)7000.20(ctrl/Y)8000.40
(ctrl/Y)34:0(return)

</div>

Note that there is only one space in the command string, between the N and the 400 at the beginning. What this one does is to run all the 48K memory tests, then repeat them indefinitely. If you run this test overnight with the Apple's cover in place, the RAM chips will be well tested for intermittent failure at normal operating temperatures.

You may want to test other RAM, such as on peripheral cards. Use the same format for any one test:

(start).(length)(ctrl/Y)

where start address and length number of pages tell the routine which RAM addresses to test. And you can combine several tests if you wish. Once the command string is known, you can run the test overnight by

N (command string)34:0(return)

just like the 48K example above.

A program written to run at one memory location in the Apple, or any 6502 machine, can be modified to run in the Apple at any free location. The modification you must make to the original program is called *relocation*. From Integer BASIC, you can use a relocating utility to help you in this task.

Normally, you relocate programs by reassembling them with a new ORG directive. If it is just a short routine that you wrote with the Miniassembler, then you can probably just change any JSR and JMP addresses by hand to conform to the addresses at the new program location. However, if you want to modify a ROM routine or move a utility that you didn't write yourself, then you'll have to relocate without the source code. For any sizeable program that you have to move this way, the relocator in the Integer BASIC utilities will save you a lot of time.

Load the program to be relocated if it isn't in the Apple already. If it is in the Applesoft ROM area ($D000.F7FF), then copy it down into the RAM area. Switch to INT if necessary and CALL-151. From the Monitor, you can access both the Miniassembler and the Relocator packages. You need access to both.

There are four steps to performing a relocation. First you must make an accurate memory map of the old program and use it to make a new memory map. Then you can use the utility to relocate code and move data to achieve the new memory map. The utility may not relo-

cate perfectly, so your third step will be to search for and make corrections to these exceptions. And finally you may use the relocator again to change the range of Page Zero usage if the program originated in another computer. Follow each step carefully.

STEP ONE: Use the Monitor's disassembler to examine the program. You must identify the code and data segments of the program, accounting for every byte. Sketch a memory map that shows all program segments and all data segments. The same map should also be labeled with the actual running addresses as well, if they are different from the locations where you are storing it.

With a memory map of the old program, you can sketch out the map of the new program. Label both data and program segments by giving the addresses of each block in the final running program. You may not wish to relocate to the running location right away, especially if it is going into PROM. In such cases, you can designate an unused segment of RAM to build the new program. The new program should be built in an area that is not otherwise being used, especially by the copy of the old program.

At this point you should have two memory maps. The first map should show the program and data segments as they are in memory. The same map should show the old locations where the program came from — the blocks of memory in which it ran. These blocks of memory may or may not be the same as the segments of memory where you are presently storing it for relocation. The lengths and relative positions are the same and that's all that matters. The running locations are called *blocks* and the relocation storage locations are called *segments.* Look at the example and compare it to your map.

The second map you have should show the program you want after relocation. The locations of each program and data block should show the final, running addresses. For relocation, you should also show the relocated storage positions as segments of the same lengths and relative positions. Again, look at the example.

Be sure there is no overlapping of the old and new segments. If everything is all right, then proceed to the next step.

STEP TWO: Engage the relocation utility to give you ctrl/Y commands for relocation with the command

D4D5G

to the monitor. Start by telling the relocator your old and new program addresses (their blocks) as follows:

<p style="text-align:center">new1<old1.old2(ctrl/Y)*(return)</p>

using the syntax of the Monitor's M command for the addresses. The asterisk tells it that you are specifying blocks of running addresses rather than segments of working storage locations. Then relocate each program segment and move each data segment in ascending sequence:

<p style="text-align:center">new<old1.old2(ctrl/Y)(return)   relocates
new<old1.old2M(return)   moves</p>

until all segments have been relocated or moved.

STEP THREE: After moving and relocating, you must go through the new code with the disassembler and look for any exceptions — address references not found by the relocator. One of these is immediate addresses. Code like

```
LDA   #$36
STA   $80
LDA   #$45
STA   $81
```

obviously sets a pointer to an immediate value, $4536. If that address is part of the relocation, then you'll have to change the immediate values to agree with your new program. Another exception to watch for is references to addresses outside the relocation addresses. This is rare. The third exception is a change to relative branches when your relocation adds new code or removes old code. If you relocate for either reason, check any branches in the region. Usually, the only exceptions you will have to correct are immediate addresses.

STEP FOUR: If your old program came from another computer like Commodore or Atari, you will probably have to change Page Zero references at this point. For instance, if the old program uses Monitor addresses like $40.5F, say, then you would change the new program so that it uses addresses in the $80.9F range instead. Continuing this example, you would type:

```
80<40.5F)ctrl/Y)*(return)
1000<.1093(ctrl/Y)(return)
.1A34M(return)
.1FD2(ctrl/Y)(return)
```

The first line sets the Page Zero ranges to be relocated. Then a program segment ($1000.1093) is relocated, a data segment ($1094.1A34) is moved, and a second program segment ($1A35.1FD2) is relocated. The syntax has a short form because the result is left in place, which is in the same location as the original.

When the relocation is complete, the new program should be saved to disk before you attempt to RUN it.

For example, the memory maps to relocate SWEET16 are given in Fig. 5-6. The new program is to run at $948C, but the example uses the


```
BLOCK                    SEGMENT        BLOCK                    SEGMENT
F689                     F689           948C                     4000
        PROGRAM                                 PROGRAM
        LENGTH = 5A                             LENGTH = 5A
F6E3                     F6E3           94E6                     405A
        DATA                                    DATA
        LENGTH = 20                             LENGTH = 20
F703                     F703           9506                     407A
        PROGRAM                                 PROGRAM
        LENGTH = FA                             LENGTH = FA
F7FD                     F7FD           9600                     4174
```

(A) Old.   (B) New.
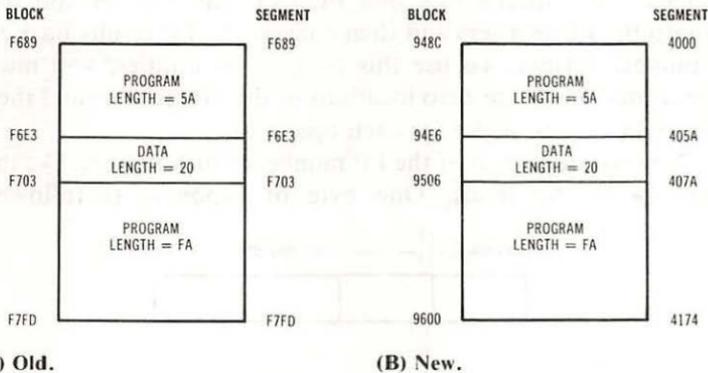
Fig. 5-6. Memory maps to relocate SWEET 16.

segment at $4000.4173 to build it. To relocate according to those maps, then, you would type the commands:

```
948C F689.F7FC(ctrl/Y)*(return)
4000 F689.F6E2(ctrl/Y)(return)
405A F6E3.F702M(return)
407A F703.F7EC(ctrl/Y)(return)
```

The first one tells the relocator to change any references in the $F689.F7FC range to ones in the $948C.95FF range. The second com-

mand relocates actual program segment $F689.F6E3 to $4000.405A in RAM. The references are changed to the $948C.95FF range, where applicable. Next, the M command moves the data segment. Finally, the second program segment is relocated. The resulting new program is at $4000.4173 in RAM, but when it is moved or loaded to $948C.95FF it will execute properly. The example used $4000.4173 to illustrate the difference between blocks and segments; in the case of SWEET16 it wasn't really necessary.

5.3.3 Floating-Point Utility Package

It is possible to have floating-point arithmetic with Integer BASIC. Although not used by the BASIC interpreter itself, a set of floating-point utility routines live at $F425.F65D in the Integer ROMs. And you can use them from machine language yourself to make floating-point calculations. Special FIX and FLOAT calls will let you put integers into the FP registers and then convert the FP results back to Integer number format. To use this package of utilities, you must know the format and Page Zero locations of the FP registers and then know the right calls to make for each operation.

Fig. 5-7 shows the format of the FP numbers which is much like the one you saw in Applesoft. One byte of exponent is followed


```
                    |–EXPONENT–| |————— 23-BIT MANTISSA —————|
                    ┌──────────┬──────┬──────┬──────────────┐
                    │          │      │      │              │
                    └──────────┴──────┴──────┴──────────────┘

SIGN OF EXPONENT ————┘         │
    1 IS + ve                  │
    0 IS – ve          IMPLIED BINARY POINT
                       BETWEEN BITS 6 AND 7
SIGN OF NUMBER ————————————————┘
    1 IS – ve
    0 IS + ve
```
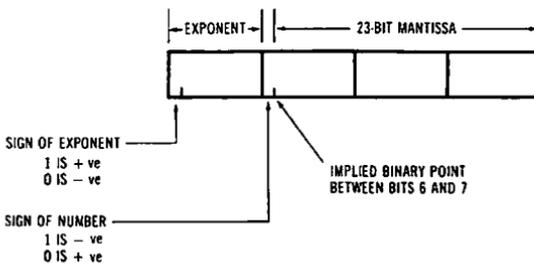
Fig. 5-7. Format of FP number.

immediately by three bytes of mantissa. The exponent is in the usual excess-$80 form, so that $80 represents an exponent of zero, $7F of minus one, $81 of plus one, and so on. The mantissa is always signed; never unpacked like Applesoft. So, the binary fraction begins at bit six of the FP number's second byte. The remainder of the mantissa is in the following two bytes — the third and fourth — in the usual decreasing order of significance. You can read one of these FP numbers just

as you would read an Applesoft FP number in storage: an exponent between $81 and $FF being positive to give a number greater than one, an exponent between $FE and $FF being negative to give a number less than one, an exponent of $80 being zero for a large, but proper, fraction. When the first byte of the mantissa is negative, the number is negative; when positive, positive.

In memory, there are two FP registers we call FP1 (from $F8 to FB) and FP2 (from $F4 to $F7) (see Fig. 5-8). In the usual binary opera-

LOCATION	FORMAT				USAGE
$F3					PRODUCT/QUOTIENT SIGN
$F4	X2	M2H	M2M	M2L	FP2 REGISTER
$F8	X1	M1H	M1M	M1L	FP1 REGISTER
$FC					WORKSPACE
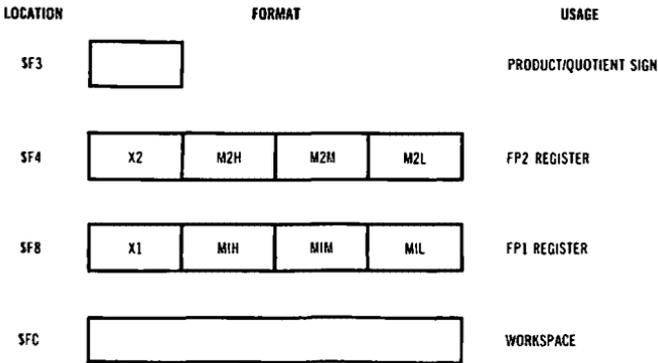
Fig. 5-8. Page zero registers for FP utility package.

tions like addition and subtraction, the operands must be put into FP format and located in FP1 and FP2. Then the call to the operation is made. When it returns, the operation leaves your result in FP1. If you are chaining operations — performing one after the other, using previous results — you should arrange to replace FP2 each time. If you juggle the registers to commute, remember that FP2 may be destroyed during an operation. Always replace FP2 before each operation, regardless.

Here are the binary operations.

FADD $F425 — Adds the contents of FP2 and FP1, putting the result in FP1. Remember that it must align the binary points before adding, so a small number may lose some significant bits to a larger number.

FSUB $F468 — Subtracts the contents of FP1 from FP2 and leaves the result in FP1. Like addition, the binary point adjustment may drop significant bits.

FMUL $F48C — Multiplies FP2 by FP1, result in FP1. May be
a danger of *overflow* from the addition of exponents.

FDIV $F4B2 — Divides FP2 by FP1, result in FP1. May cause
underflow from the subtraction of exponents, but that
merely zeros the result. The greatest danger is from a
small divisor causing an overflow and from division by
zero (extreme case). Overflow and underflow are each
handled differently. In the case of an underflow result,
you get a zero value returned, which is a nonfatal error.
However, for overflow, you must have a JMP instruc-
tion at $3F5, called OVLOC, that vectors to your over-
flow handling routine. Overflow is a fatal error other-
wise. Any result that gives an exponent greater than $8D
will most likely cause overflow.

 The *unary functions* let you get FP numbers con-
verted to and from Integer format. Integers are in low-
byte/high-byte order, opposite to that of FP mantissas.
With that in mind, using the conversion routines is
straightforward.

FCOMPL $F4A4 — Complements the FP number in FP1. That is,
it changes its sign: FP1 becomes − (FP1).

FLOAT $F451 — Converts a two-byte integer to FP by nor-
malizing the mantissa of FP1. Put the high byte in
M1H, the low byte in M1M, and zero M1L. If you do
have a fractional part, it can go into M1L instead of
zero. Since an integer has its sign in the high byte, all
you have to do now is CALL FLOAT and the conver-
sion is done.

FIX $F460 — This *undoes* the action of FLOAT; it results in
an integer-low in M1M and an integer-high in M1H. Re-
maining significant figures are in M1L which you can
pick up as the binary fraction if you need it. Use M1L to
round off your integer result if need be, by carrying the
sign bit. Again it all happens in FP1.

FIX1 $F68D — There is a bug in FIX: it won't work properly
with negative numbers. If you test M1H, you can JSR
FIX1 instead whenever FP1 is negative. FIX1 works all
right for them.

 Use FIX for positive numbers, FIX1 for negative ones.

You can write routines to copy in and out of FP1 and FP2 to make the package useful. For example,

```
GETFP:  LDY #0
        STY FP1 + 3     ;zero M1L
        LDA (ZNUM),Y   ;ZNUM points to integer
        STA FP1 + 2     ;low byte to M1M
        LDA (ZNUM),Y
        INY
        LDA (ZNUM),Y   :high byte to M1H
        STA FP1 + 1
        JMP FLOAT
```

will fetch the integer whose address you have in ZNUM and convert it to FP format in FP1. To copy an integer back into storage:

```
PUTFP:   BIT FP1 + 1     ;test sign of mantissa
         BPL PUTFP1
         JSR FIX1        ;case: Positive
         CLC
         BCC PUTFP2      ;always
PUTFP1:  JSR FIX         ;case: Negative
PUTFP2:  LDY #0          ;continue
         LDA FP1 + 2     ;low byte of integer
         STA (ZNUM),Y
         INY
         LDA FP1 + 1     ;high byte of integer
         STA (ZNUM),Y
         RTS
```

Depending on how elaborate your needs are, you can add more routines to swap F1 with F2, copy the entire FP number to and from temporary storage, stack and unstack them, etc.

5.3.4 SWEET16 Pseudo-Processor

The 6502 processor is fast and powerful when it comes to working with eight-bit data. However, fancy software needs the ability to work with *sixteen-bit* addresses as data, something the 6502 won't handle

easily. Sure, we have the indirect indexed form of addressing, but that only reaches one page of memory at a time. What we need are some of the features that sixteen-bit processors like the PDP-11 and the 6809 have for addressing.

If you can tolerate the loss in speed, the Apple II can run a sixteen-bit *emulator* that will give you sixteen-bit addressing instructions, without having to plug in a processor board. This emulator is called SWEET16 and it was one of the first programs written for the Apple II. It is easier to use and it's also faster than BASIC.

With SWEET16 you can write editors, languages, parsers, memory moves, or anything that needs simple address manipulation instructions. If 6502 is your first machine language, then working with SWEET16 is one way of *bridging the gap* in learning the larger processors like the 6809.

What SWEET16 does is pretend to be a sixteen-bit processor and interprets its own set of op codes to execute little routines that it regards as instructions. You just JSR SW16 where you want to switch to SWEET16 instructions in your program. Following the JSR, you put SWEET16 code. The end of SWEET16 is a zero byte that SWEET16 regards as an RTN (return) op code. Following the RTN, you continue normally with your next 6502 instruction. Of course, since SWEET16 is a subroutine, you are merely passing parameter string by immediate value, but the effect is to switch from 6502 processor instructions at the JSR SW16 to *SWEET16 processor* instructions. Then, a second change from SWEET16 instructions at the RTN switches you back to the 6502. So, the subroutine called SW16 emulates another processor.

To emulate a processor, SWEET16 needs a set of registers and refers to them during execution of the instruction set. Let's look at these registers and then at the instructions that modify them. See Table 5-6.

There are sixteen registers; each register must have a sixteen-bit capacity. This means two bytes of 6502 memory for each register or 32 bytes total. SWEET16 uses the first 32 locations in Page Zero for registers, $00.1F. The first register is R0 that resides at $00.01 in conventional low-byte/high-byte order. The next register is R1 at $02.03 and so on up to R15 which is the last register occupying $1E.1F. Of these sixteen registers some are dedicated by SWEET16 while others are free for user definition.

The registers R1 to R11 inclusive are yours to define any way you want. These are *user-defined registers* and you keep your memory

Table 5-6. SWEET16 Op-code Summary

Register Ops				Nonregister Ops		
1n	SET	Rn	load 2 bytes immediate	00	RTN	return to 6502 mode
2n	LD	Rn	load ACC from Rn	01	BR ra	branch always
3n	ST	Rn	store ACC to Rn	02	BNC ra	branch if no carry
4n	LD	@ Rn	load ACC-lo, indirect: (Rn +)	03	BC ra	branch if carry
5n	ST	@ Rn	store ACC-lo, indirect: (Rn +)	04	BP ra	branch if plus result
6n	LDD	@ Rn	load ACC, indirect: (Rn + +)	05	BM ra	branch if minus result
7n	STD	@ Rn	store ACC, indirect: (Rn + +)	06	BZ ra	branch if zero result
8n	POP	@ Rn	(− Rn):load ACC, indirect	07	BNZ ra	branch if nonzero result
9n	STP	@ Rn	(− Rn):store ACC, indirect	08	BM1 ra	branch if minus-one result
An	ADD	Rn	add Rn to ACC	09	BNM1 ra	branch if not minus-one result
Bn	SUB	Rn	subtract Rn from ACC	0A	BK ra	causes 6502 BRK event
Cn	POPD	@ Rn	(− − Rn):load ACC, indirect	0B	RS	return from SW16 subroutine
Dn	CPR	Rn	compare (ACC − Rn to R13)	0C	BS ra	branch to SW16 subroutine
En	INR	Rn	(Rn +), i.e. increment by 1	0D		unassigned
Fn	DCR	Rn	(− Rn), i.e. decrement by 1	0E		unassigned
				0F		unassigned

Registers Usage	Notes

R0 = ACC, the accumulator
R1 . . . R11 are User-Defined
R12 = SP, the stack pointer
R13 is the result of last compare
R14 is the status register
R15 = PC, the program counter
	n is the register number
	ra is the relative address, ± 127
	SET is 3 bytes, branches are 2 bytes, and all others
	  are 1 byte long.
	To call: JSR SW16 (at $F689 in Integer) (list SWEET16 code)
	  exit with RTN($00), last byte

pointers, counters, and perhaps an important constant or two in them. Much the same way that you would normally use Page Zero with the 6502.

The first register, R0, is special. You use it directly as the accumulator and we call it ACC instead of R0 most of the time. It's the busiest of all registers.

The registers R12 to R15 are used by SWEET16 as special registers. R12 is the stack pointer and may be referred to as SP. R13 holds the result of the last compare and is used by the branch instructions. R14 is the status register and keeps the pointer to the current register as well as the carry flag. R15 is the program counter that SWEET16 uses to point to the instruction list currently being read. You can address any register including these special ones with SWEET16 op codes, but be careful. Unless you understand fully what you are doing, leave R12, R13, R14, and R15 alone.

There are two groups of instructions in SWEET16: register OPs that address and modify explicit registers, and nonregister OPs that implicitly alter the special registers. For example, a branch instruction will test R13 then perhaps modify the program counter in R15 to effect a branch. That is a nonregister OP because no register is actually specified in the instruction. A SET R0, $0000 instruction is a register OP because it explicitly names R0, which is the ACC, to be set to zero. All the register OPs give a result in R13 for use by the branch nonregister OPs. This follows the pattern of actions and decisions we need to build structures when we program: register OPs are your actions and nonregister OP branches are your decisions.

You can get 6502 Assemblers that support SWEET16 instructions such as BIGMAC from Apple, Pugetsound. Or you can hand assemble the routine yourself quite easily. The instruction set is logical and you only need the SWEET16 op code summary of Table 5-6 to code.

It is easy to assemble a register OP. They are only one byte each, except for the SET instruction. You just lookup the code — 1 to $F as the first digit of a two-digit hex number. The second digit is the register number, $0 to $f. The SET has a two-byte operand as well, so you assemble that in low-byte/high-byte order. For example,

<center>1A 00 03   SET   R10, $300</center>

for the SET and

<div style="text-align: center">

35   ST   R5

</div>

for the others.

Nonregister OPs may have one or two bytes, usually two. The first byte has zero as its first hex digit and then zero to $c as its second digit. Branches have a second byte for the relative address, exactly the same as 6502 branches do. You calculate the offset the same way:

<div style="text-align: center">

1400:01 0A   BR   BELOW
140C: . . .   BELOW: . . .

</div>

The relative address of $0A here is the difference between the branch address and the address of the next instruction (the current PC). There minus here. Just like 6502.

If you look at any of the SWEET16 examples you will see the call sequence. The JSR SW16 starts the emulator. The instructions for SWEET16 follow with the last instruction being physically in the last byte of the call. It is zero; its mnemonic is RTN for return to 6502. You must always return at the very end of a chunk of SWEET16 code so that a normal 6502 instruction can immediately follow the RTN (zero).

Whenever you use SWEET16, she saves all your 6502 registers before doing anything. Then she restores them just before setting the PC-reg of the 6502 to your next instruction following the RTN. So, your registers are unchanged by a SW16 call.

If you are just learning SWEET16, write a short routine to make sure that you call the program properly. The following will do:

```
0300: 20 89 F6    JSR  SW16            ;enter SWEET16
0303: 10 80 02    SET  R0,    $280
0306: 11 10 01    SET  R1,    $110
0309: 12 40 01    SET  R2,    $140
030C: A1          ADD R1
030D: 53          ST   R3
030E: B2          SUB  R2
030F: 54          ST   R4
0310: 00          RTN                  ;enter 6502 again
0311: 4C 69 FF    JMP  MONZ            ;back to Monitor
```

Use the SWEET16 op code summary in Table 5-6 to check that you

have followed the code and its assembly. Then enter the routine and run by 300G. What will be the contents of the five registers — R0, R1, R2, R3, and R4? Dump them ($00.09) and see:

```
0000: 50 02 10 01 40 01 90 03
0008: 50 02
```

which is:

```
R0 = $0250 , the ACC after calculations
R1 = $0110
R2 = $0140
R3 = $0390 or $280 + $110
R4 = $0250 or $280 + $110 - $140
```

Now, write a couple of routines of your own design to see what you can do. Use the SET, INR, and DCR instructions on any one register, R0 (ACC) to R11. And you can modify the ACC with LD, ADD, and SUB. The ST copies from ACC to any register. These seven OPs are the simplest to use; use them to get familiar with SWEET16. When you can use the register OPs listed in Table 5-7 easily, you can then learn the ones that use indirect addressing.

Table 5-7. Beginners' Register Ops

One register	ACC and another register
SET Rn, const INR Rn DCR Rn	LD Rn ST Rn ADD Rn SUB Rn

While the beginners' OPs let you play with the registers, you need indirect addressing OPs to reach other memory in the Apple. This is where SWEET16 becomes useful. For instance,

```
0300: 20 89 F6 MOVE:   JSR  SW16            ;to SWEET16 mode
0303: 11 00 20         SET  R1,   $2000     ;from
0306: 12 00 40         SET  R2,   $4000     ;to
0309: 13 00 20         SET  R3,   $2000     ;length
030C: 41       MOVE1:  LD   @R1             ;get byte
030D: 52               ST   @R2             ;put byte
```

030E: F3	DCR R3	;count length
030F: 07 FB	BNZ MOVE1	;next byte
0311: 00	RTN	;to 6502 mode
0312: 60	RTS	

uses a SWEET16 call to copy HIRES1 screen to HIRES2. The R1 register points to the address copied *from* and the R2 register points to the address copied *to*. The R3 register handles the length in bytes of the copy. Compare this to the code needed to do the same job with the 6502.

Look at the MOVE example in detail. The length in R3 is a loop counter only; it plays no part in the memory addressing. The BNZ detects the end of the loop when it has counted down 8,192 times. So, without considering the actions of LD @ and ST @, the code is straightforward.

First, look at the LD indirect. With $2000 in R1 it fetches the contents of location $2000 to the ACC. Only the low byte of ACC is loaded from memory; the high byte is simply zerod. That done, the R1 register itself is automatically incremented. So, at the end of the instruction, R1 points to $2001. Second, look at the ST indirect. It stores the contents of ACC-low to memory location $4000. Then it increments R2 to $4001 from $4000. After executing the copy from $2000 to $4000, R1 points to $2001 and R2 points to $4001.

The auto-increment feature of the indirect instructions makes routines like MOVE easy to write. Start at the lowest addresses of interest, set up any counters, then use the auto-increment to pass through the memory blocks one location at a time, in increasing sequence.

If you look at Table 5-6, you can see other instructions with the "@" of indirection. These work the same way as the LD @ and ST @ by loading or storing the ACC from or to memory. The difference is in the number of bytes, one or two, and in the direction of stepping through memory. The auto-increments are noted as (Rn +) and (RN + +) for one or two locations at a time; the auto-decrements are noted as (− Rn) and (− − Rn) for one or two locations at a time. They each have special uses that go beyond the simple MOVE example and they make SWEET16 a powerful tool.
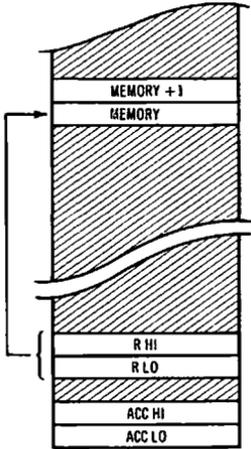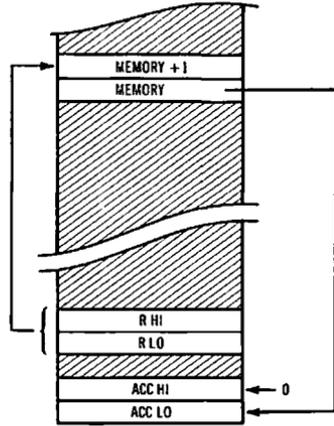
(A) Before.   (B) After.

Fig. 5-9. The LD@ instruction.
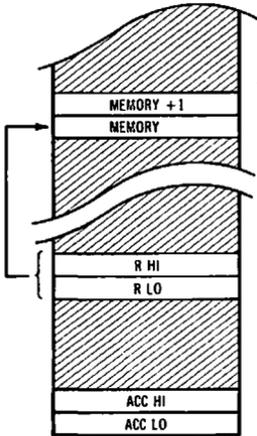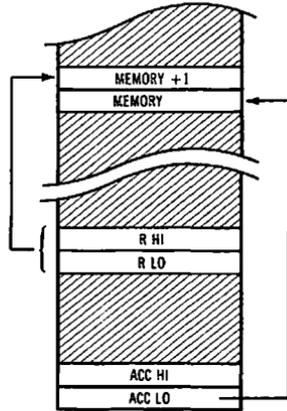
(A) Before.   (B) After.

Fig. 5-10. The ST@ instruction.
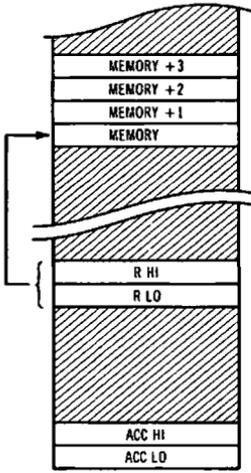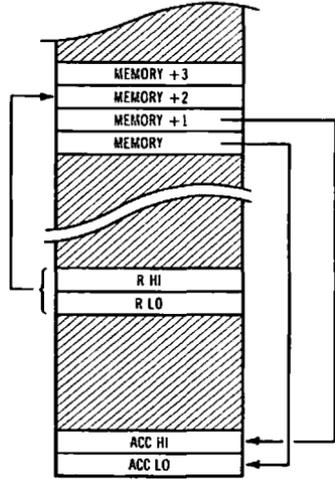
(A) After. (B) Before.

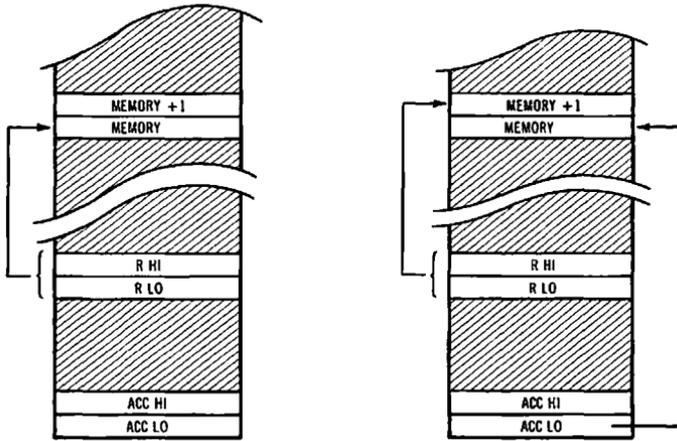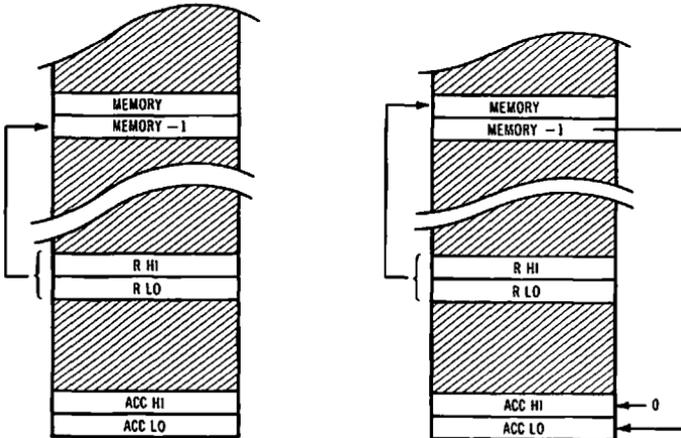Fig. 5-11. The LDD@ instruction.
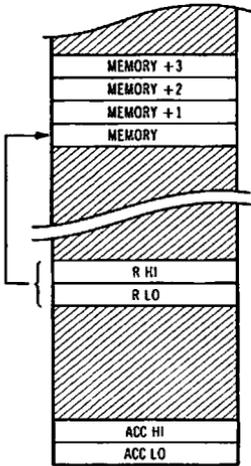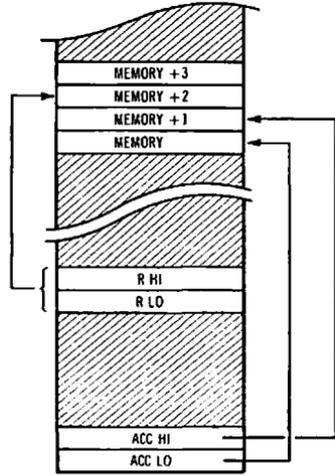
(A) After. (B) Before.

Fig. 5-12. The STD@ instruction.

(A) After.   (B) Before.

Fig. 5-13. Pushing one byte on a stack using ST@ instruction.

(A) After.   (B) Before.

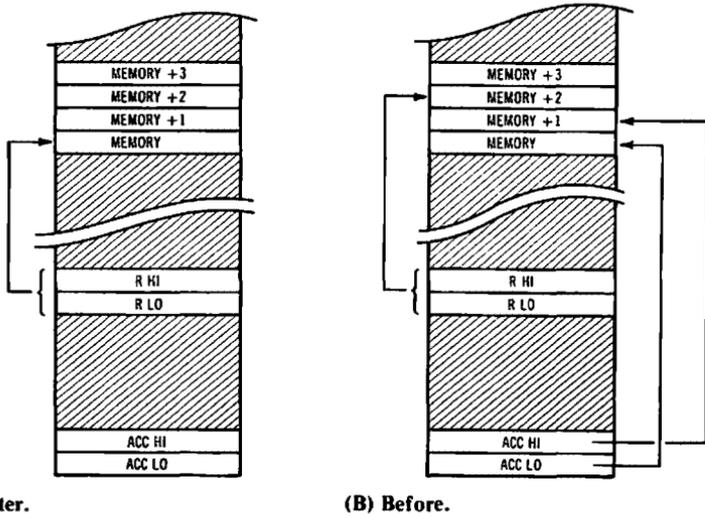**Fig. 5-14. Pulling one byte from a stack using POP@ instruction.**

(A) After.   (B) Before.

Fig. 5-15. Pushing two bytes on a stack using STD@ instruction.

(A) After.   (B) Before.
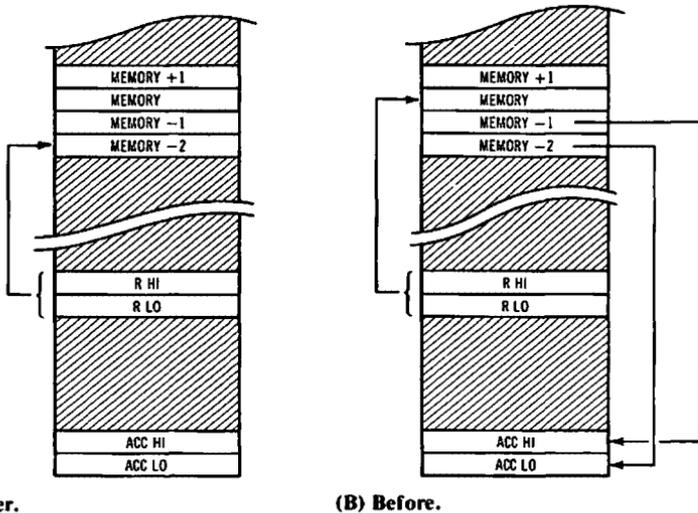
Fig. 5-16. Pulling two bytes from a stack using POP@ instruction.

CHAPTER SIX

Text and Graphics

6.1 THE MONITOR TERMINAL

All the hardware and software that is needed for the Apple to function as its own terminal is built-in. All that you need to add is a television or video monitor. For an 80-column display, add a video board in Slot Three (or in the Auxiliary Slot in the IIe model). The built-in software can be switched to use its own 40-column routines or the ones on the card. The switches for the keyboard and video display allow different types of input and output to be made in place of the built-in monitor; these switches are called *hooks*.

6.1.1 Monitor Hooks

The Apple works with inputs and outputs similar to modern sound systems. In such a system you can select one of the several inputs: a record player, a tuner, a tape player, or a microphone. Similarly, you could switch one of several outputs: main speakers, remote speakers, headphones, or tape recorder. So to operate your sound system you would set the input switch to your choice of device and then set the output switch to your choice of reproduction device. You can use your Apple with inputs and outputs in much the same way.

The Apple has a keyboard built-in as an input device. Input can also be from peripherals you can plug in: disk controller, serial communications devices, or other devices with compatible interfaces. And you

can use a built-in video display or a peripheral such as disk, serial communications, or an 80-column video. Your Apple has two "switches" to do this job — an *input switch* and an *output switch*. You can set these switches yourself or use one of the built-in routines to set them for you. In either case, you control the input and the output that the Apple sees.

The input "switch" is called the *input hook* or *link*. It is simple enough, just two Page Zero locations at $0038.0039. Location $0038 is called KSW and contains the address of the current input device's routine. Whenever the Apple wants an input, it calls the routine whose address is in KSW. Similarly, there is an output hook or line at $0037.0038 called CSW. So, the Apple can send a character to the current output device by calling the routine whose address is in CSW. It is by setting the input hook KSW and the output hook CSW that you control the information flowing in and out of your Apple.

Perhaps the simplest arrangement of the hooks is made by the RESET routine. It sets the input hook KSW to the built-in keyboard and sets the output hook CSW to the built-in video display. Unless you bootstrap using a disk, you get the use of the keyboard and video display, by default.

The RESET routine sets the input hook to the built-in keyboard routine at $FD1B called KEYIN. Then whenever input is wanted, KEYIN supplies it one character at a time. A call to KEYIN results in a wait until you press a key. While it waits, it runs a counter to generate a random number. When you press a key, the counter stops and the random number is available to you at $4E.4F in Page Zero in low-byte/high-byte order. And the character you typed from the keyboard is returned in the A-register. There are important differences in the way KEYIN is used between the Apple II Plus and the Apple IIe, in the location and extent of other keyboard functions such as escape handling and cursor display. In both models, however, KEYIN generates a random number in RND ($004E), gets a character which it returns in the A-reg, and removes the displayed cursor by replacing it with the original character on the screen.

The built-in video display address is $FDF0 and is called COUT1. The RESET routine puts it into the CSW output hook at power up so that all Apple output will go to the COUT1 routine. There, the character in the A-reg is interpreted so that display characters will appear on the screen and control characters invoke changes to the display parameters. For instance, a ctrl/G sounds the *beep*. Whenever a printable

character is displayed, the cursor is advanced. Most of the control characters are cursor-related: ctrl/H is the backspace, ctrl/M is the carriage return, ctrl/L is the form feed (homes cursor and clears screen), and so forth. COUT1 also masks its display characters to get various character sets: normal, inverse, flashing, in either upper or lower case. The display is restricted to an area of the screen called the window. Normally, the window is the full screen, but may be changed at Page Zero locations $20.23. So, COUT1 either interprets a control character or displays a printable character within the text window from one of several possible character sets.

With COUT1 or some other output routine address is CSW and RDKEY or some other input routine address in KSW, the Apple programs can put characters out or get characters in. The way to do it is to call an indirect JMP that jumps at the required hook. To get an input character,

JMP (KSW)

is needed, and to put a character to the current output use:

JMP (CSW)

It isn't necessary to code these jumps if you want to reach the current input and output routines. They are in the Monitor and are called RDKEY and COUT. To get a character from the current input routine whose address is at KSW ($38.39) you just

JSR RDKEY

where RDKEY is $FD0C. Similarly, to put a character to the current output routine whose address is at CSW, you write

JSR COUT

where COUT is $FDED. It is only the routines that initialize and maintain peripherals that access CSW and KSW directly to set them up; all other routines in the Apple can just call to RDKEY for input and to COUT for output. Hook addresses are summarized in Table 6-1.

Table 6-1. Input-Output Hook Addresses

Decimal	Hex	Label	Contents
54	38.39	KSW	Input hook address
56	36.37	CSW	Output hook address
1002	3EA	MVSW	DOS routine reconnects hooks
40577	9E81		DOS input hook routine
40637	9EBD		DOS output hook routine
43603	AA53		Current DOS hook for CSW
46305	AA55		Current DOS hook for KSW
64780	FD0C	RDKEY	Advance cursor then input at KSW
64792	FD18		Input at KSW
64795	FD1B	KEYIN	Keyboard input routine
65005	FDED	COUT	Output at CSW
65008	FDF0	COUT1	Video output routine
65161	FE89	SETKB	Resets KSW to KEYIN
65171	FE93	SETVID	Resets CSW to COUT1

Here's how you use the hooks without DOS active in your Apple. From BASIC, you can use the commands

<div align="center">

PR#s

IN#s

</div>

where s is the slot number of the peripheral you are selecting. If s is zero, the commands will set the hooks to the built-in terminal, just like RESET. The PR#s command sets the output hook CSW to the address of slot s. The IN#s command sets the input hook KSW to the address of slot s. The slot s can be any number from one to seven. You can choose any slot as the current input and any slot as the current output. Then you can remove the current input with a IN#0 and the current output with a PR#0.

This is what happens when you give a PR#0: BASIC puts the address of COUT1 ($FDF0) into CSW at $38.39. And when you give an IN#0 command, it puts the address of KEYIN ($FD1B) into KSW at $36.37. The result is to make the built-in video and keyboard terminal routines the current output and input devices, replacing any others that may have been in CSW and KSW. But, if you commanded PR#1, then CSW would be set to the address of Slot One ROM which is $C100. Or, if you gave a PR#2 command, CSW would be set to $C200 which is the Slot Two ROM address. Whichever slot you choose as the output device, BASIC will set CSW to $Cs00 (where s is the slot number). You select the input device the same way: IN#0 sets KSW to

the KEYIN ($FD1B) routine, IN#1 sets KSW to Slot One ROM at $C100, IN#2 sets KSW to Slot Two ROM at $C200, and so forth.

By using only one input device and one output device at any one time, the Apple input/output system is simply a matter of selecting each device as needed. You just use the IN# and PR# command to make the selection. What can complicate matters is when you have to keep DOS active at the same time.

The Disk Operating System was added to the Apple later, after BASIC was used for some time. So, it had to fit into the Apple's simple I/O system as best it could. For the most part it does so, but the odd glitch here and there can trap the unwary. The problem is that DOS must occupy KSW and CSW itself in order to trap commands to itself, get output from BASIC PRINT commands, and supply input to INPUT and GET commands in BASIC statements. It must also allow one other input device and one other output device in addition to itself so that you can use a printer or a serial communications device with the DOS active. The result is a DOS that occupies the hooks and maintains a pair of I/O hooks for these other devices.

Here's how you use the hooks with DOS. Again, you use the commands

PR#s

IN#s

where s is the slot number. Under DOS these commands are trapped and BASIC never gets to see them. DOS uses them to set its own pair of hooks. As before, the PR#0 removes the current output device and IN#0 removes the current input device. Watch out for the glitches when using PR# and IN# commands with DOS active. You can use them immediately as keyboard commands, or you can use them in BASIC statements by prefixing with ctrl/D just like any other DOS command. But if you use them directly in a BASIC statement without the ctrl/D you will disconnect DOS. If you didn't intend to do so, this can be a hard-to-find bug in your BASIC program.

If DOS becomes unhooked, you can reconnect DOS by a

CALL 1002

which is a DOS routine vector at $3EA. You can make this call directly or from within a BASIC statement. The call causes DOS to reconnect CSW and KSW while saving the contents of the hooks as its

own hooks. So, whatever the input and output devices were before the call, they will still be current, but DOS will occupy the system hooks.

You can deliberately unhook DOS by using PR#0 and IN#0 from BASIC or

```
        JSR   SETVID
        JSR   SETKBD
```

from Assembly (SETVID is $FE93, SETKBD is $FE89). Or, use any other PR#s and IN#s you wish. From Assembly, you just set CSW and KSW to the device address, $Cs00. Then you can always rehook DOS back into CSW and KSW with a CALL 1002 (or JSR $03EA) whenever you wish.

Most of the time, however, you will simply use the IN# and PR# commands normally with a ctrl/D prefix.

In addition to using the hooks for input and output from peripheral devices, you can use them with your own device drivers. For instance, you can have lowercase input from the old keyboard even though it won't generate any lowercase letters by itself. Here's how.

Your keyboard routine will use the ESC key to shift from lower- to upper-case, and to lock into uppercase. This routine replaces KEYIN by substituting its own address for that of KEYIN's. Then it will display the cursor of your choice, get a character, handle any uppercase to lowercase conversion, and finally replace the cursor with the old character before returning with the new character in the A-reg. The routine must have the cursor screen position in BAS (in Page Zero) and the Y-reg, and the old character from the screen in the A-reg when it is called via KSW. A routine in the Apple Monitor does that; it's called RDKEY and it resides at $FD0C.

To enable the routine, you can call these instructions

```
HOOKUP  LDA   #GETCH
        STA   KSW
        LDA   #GETCH
        STA   KSW+1
        JSR   MVSW      to set DOS
        RTS
```

and to remove it, the code is:

```
UNHOOK   JSR   SETKBD
         JSR   MVSW
         RTS
```

You call HOOKUP and UNHOOK whenever you use the IN#3 and IN#0 commands to setup and remove a device.

You can write and use a routine to replace the CRT output routine COUT1 in the same manner. The address of the routine must be put into CSW and a call to MVSW made to get DOS hookup. This can be used for a HIRES character generator and display routine. Such routines are available or you can write them yourself. A good one will support scrolling and windowing, but a simple one is quite satisfactory for labeling graphs. Then you can alter the font if you wish, and you will get lowercase letters that you don't get with the older Apples.

A simple alternative to HIRES character generation is the use of inverse video for uppercase letters. On old Apples, the lowercase will come out as normal uppercase, so it is only necessary to mask uppercase letters for them to show as such. You just use a routine to edit each character and then send the edited character along to COUT1. Instead of replacing the COUT1 CRT output routine, like a HIRES routine would, you cascade your routine with COUT1 to make a smarter routine with less work. Here it is.

```
LCOUT    PHA
         CMP   #$C0      test for uppercase
         BCC   LCOUT1    range $C0...DF
         CMP   #$E0
         BCS   LCOUT1
         LDA   #$3F      mask for INVERSE
         BNE   LCOUT2
LCOUT1   LDA   #$FF      mask for NORMAL
LCOUT2   STA   INVFLG
         PLA
         JMP   COUT1
```

If you are using LCOUT together with GETCH for your terminal, you can use single routines to hook and unhook both:

```
HOOK      LDA   # GETCH        hook keyboard
          STA   KSW
          LDA   # GETCH
          STA   KSW + 1
          LDA   # LCOUT
          STA   CSW
          LDA   # LCOUT
          STA   CSW + 1
          JSR   MVSW
          RTS
UNHOOK    JSR   SETKBD         remove keyboard GETCH
          JSR   SETVID         remove display LCOUT
          JSR   MVSW           reset DOS hooks
          RTS
```

Whether you replace or cascade the Monitor terminal routines, the procedure is the same. An output or an input routine that replaces the Monitor won't reference COUT1 or KEYIN. An output routine that cascades with COUT1 does so by a JMP to COUT1 at the end. An input routine that cascades with KEYIN does so by a JSR to KEYIN at the beginning.

6.1.2 The Keyboard

Your first choice of a keybaord input routine is the Applesoft INPUT command. When used as an instruction, it returns one or a list of variables with the data entered from the keyboard. It requires no programming to make it work, it can include a prompt string of your choice, and it will work on all models of Apple II. Use it in one of three ways:

> INPUT "your prompt";*varlist*
> INPUT "''";*varlist*
> INPUT *varlist*

The first form uses your prompt string, the second form has a null prompt so that none will appear, and the third form gives the default prompt, "?". The varlist can be any list of variables, separated by commas. Often only one variable is INPUT. Any numeric variables will induce a VAL function to convert the entered string to a number. All in all, a versatile and easy-to-use keyboard entry tool.

The INPUT command will also fetch records from disk if the DOS command READ is in effect. A record is a string terminated by a CR character, just like INPUT wants from the keyboard. The fields of a DOS record separated by commas will be applied to your variable list in exactly the same way. A DOS INPUT is the same as a keyboard INPUT except for the source of characters.

But if you input strings that contain delimiters, commas, colons, and so forth, then INPUT won't work. For instance, you may INPUT from the keyboard for a natural string from the user who doesn't know or care about Applesoft or its hangups. Or you may be reading an unknown DOS record and want to scan the record to determine the fields yourself. You can't specify a field list if you don't know what the fields will be. So input an entire record as a single string, regardless of any delimiters in that string. You need what is called an *Input Anything* routine.

Here it is. You must find a place for the routine in memory and either CALL it there or, preferably, use the ampersand feature.
Here is an Input Anything routine:

```
INPUT    JSR   PTRGET      get string variable
         JSR   INLIN       input the string to $200
         LDX   #$FF        find string length
INPUT1   INX
         LDA   INBUF,X     buffer at $200
         BNE   INPUT1      NUL is end-of-string
         TXA
         LDY   #0          put length into string
         STA   (VARPNT),Y  descriptor of variable
         LDY   #1
         LDA   #>INBUF     put addr-lo into descriptor
         STA   (VARPNT),Y
         LDY   #2
         LDA   #<INBUF     put addr-hi into descriptor
         STA   (VARPNT),Y
         JMP   DATA
```

To call the routine and input a string from Applesoft, you use the line

$$\&A\$:A\$ = MID\$(A\$,1)$$

after the ampersand vector that has been set up at $3F5. The A$ can be any string variable. The MID$ must be used immediately to reassign the string from $200, or else the next INPUT will clobber it. INLIN is at $D52C.

If you need lower level access to the keyboard than Applesoft provides, then consider the Monitor routines. By using one of these routines instead of writing your own, you save the work of programming and the hassle of maintaining and loading a separate ML file that merely duplicates what's in the Monitor already. You can use the Monitor to get complete lines, get characters one at a time, or get characters without going through KSW while another device is occupying the input hook. Use the one best suited to your needs.

The routine to get complete lines is called GETLN and it resides at $FD6A. The Monitor's command parser, Applesoft, Integer BASIC, and the Miniassembler all use the GETLN routine. Just put your prompt character into PROMPT ($33 in Page Zero) and JSR. The length of the line is given to you in the X-register and the string itself begins at $0200, INBUF. For instance,

> Applesoft calls with "]" in PROMPT
> INTEGER calls with ">" in PROMPT
> Miniassembler calls,"!" in PROMPT
> Monitor calls with "*" in PROMPT

so you will have another choice for your routine — "#" perhaps, or "@". The prompt character you choose identifies your routine to the user.

The GETLN routine converts lowercase letters to uppercase. This is why you can't get lowercase letters on Apples, even if a lowercase keyboard is used. The one exception — at time of this writing — is the Apple IIe monitor. The IIe will not change the lowercase letters to uppercase. So, if you need lowercase letters then GETLN can't be used if your program is used in one of the older Apples. For RAM copies of Autostart such as those found in FPBASIC and INTBASIC, changing the contents of $FD83 from $DF to $FF will allow lowercase in GETLN.

Otherwise, if you want input without any of the drawbacks that are in GETLN, then use RDCHAR or RDKEY to get one character at a time. RDCHAR gets characters and handles any ESCape sequences while RDKEY just gets characters. The character on the screen at the

current cursor position is set to flash and is kept in its original form in the A-reg as it calls KEYIN via KSW. The KEYIN routine gets the character from the keyboard and replaces the old screen character at the cursor position. On the IIe, the flashing character is not used when lowercase is in effect, because lowercase and flash aren't available in the same character set. Instead, the IIe terminal uses its own cursor but supports the KSW terminal interface call sequence just like the Autostart Monitor. So, use RDCHAR to get characters with full ESC support and use RDKEY for simple character input without ESC sequences being trapped.

Finally, you can get characters from the keyboard without going through the KSW input hook. This is handy because you don't have to unhook and then re-hook an existing input device just to get one character from the keyboard. There are two cases of such low level keyboard gets. One is when you want the character and must wait until a key is pressed. Another is when you want to get the character only if a key was pressed. In the second case you just want to look for a keypress, then keep on with your program regardless of whether the key was pressed. You can write simple routines to do both.

To get a keystroke by actually waiting until a key is pressed, either call KEYIN or code the following:

```
GET        LDA    $C000
           BPL    GET
           BIT    $C010
           AND    #$7F            for positive ASCII
           RTS
```

Or, to test for a keypress:

```
KTEST      BIT    $C000
           BPL    KTEST1
           LDA    $C000
           AND    #$7F            for positive ASCII
           STA    $C010
           BNE    KTEST2
KTEST1     LDA    #0              NUL if no keystroke
KTEST2     RTS                    Z-flag = 0 if keystroke
```

The KTEST returns a zero (ASCII NUL) in the A-reg if no key was pressed and will return the character entered if a key was pressed.

There isn't any corresponding routine in the Monitor that tests the keyboard on the fly like KTEST.

6.1.3 The Video Display

The built-in terminal has Monitor routines to display and scroll text (see Fig. 6-1). To do this, the routine uses six Page Zero parameters. By controlling these parameters yourself, you can change the display for your custom screen routines.
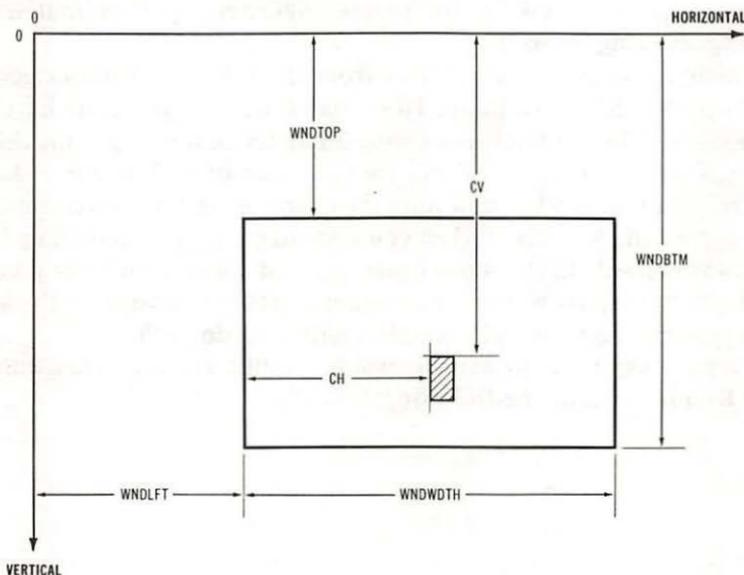
Cursor is CH and CV. A JSR to VTAB sets the cursor address. Note that CH and WNDWDTH are with respect to WNDLFT while CV and WNDBTM are with respect to zero. the top of screen. For scrolling, you must set CH and CV within the window area.

Fig. 6-1. How the scroll window works.

The cursor is kept as two parameters: CH and CV keep the horizontal and vertical cursor values. Scrolling is controlled by a scroll window defined by four parameters: WNDTOP, WNDBTM, WNDLFT, and WNDWDTH. You can set the parameters to any set of values you want.

To set CH and CV, use numbers that count from zero if you are in Assembly or from one if you are in BASIC. For instance,

```
LDA   #0
STA   CH
STA   CV
```

sets the cursor to the upper left corner of the screen from the Assembly, but

HTAB 1 : VTAB 1

would do it from BASIC. Similarly,

```
        LDA   #$27
        STA   CH
        LDA   #$17
        STA   CV
```

puts the cursor at the lower right of the 40-column screen, just like

HTAB 40 : VTAB 24

would from BASIC. On the IIe you can use forty more positions horizontally.

To set the four window parameters, use the TEXT command from BASIC. This sets the 40-column screen to a full screen window with parameter values of zero in WNDTOP, 24 ($18) in WNDBTM, zero in WNDLFT, and 40 ($28) in WNDWDTH. For the IIe you could change WNDWDTH to any value up to 80($50) when you're in the 80-column mode.

What the scroll window parameters let you do is reduce the window to a small prompt area somewhere on the screen. If you use the screen to display information and don't want everything scrolled off the screen by INPUTs, then you can set the window to cover just enough area for prompt-and-accept dialogue with the user. The dialog can continue for an indefinite number of prompts and retries without destroying your displays elsewhere on the screen. For example, you might use the bottommost row for error messages and prompts only. So when your program detects an error, the error handling subroutine has the last row all to itself:

```
TEXT                 puts the cursor at lower right and resets
                     window parameters.
POKE 34,23           sets WNDTOP
HOME                 clears window, 24th row only
. . .error dialog. . .
```

HOME	clears 24th row
TEXT	reset parameters
RETURN	

For entering data at any screen location, you will need a more general procedure. The WNDTOP and WNDLFT parameters set the upper left corner of the scroll window, while WNDBTM and WNDWDTH set the lower right corner but each in a different way. WNDWDTH sets the width — the distance from WNDLFT to the right edge of the window. WNDBTM sets the absolute value of the bottom edge regardless of the value of WNDTOP. See Fig. 6-1. You must position the cursor within the scroll window for it to work properly. When all this is done, a JSR to CROUT or a plain PRINT will send a carriage return forcing the scroll to take place. The window parameters ranges: WNDBTM must be 24 or less and greater than WNDTOP; WNDLFT and WNDWDTH must be 40 or less (80 or less for the IIe in 80-column mode). Here is a window setting routine that sets the scroll window to any location (X,Y) on the screen with a size of DX by DY. The window is cleared and the cursor is placed at the upper left position within the window.

```
SETWND   JSR   GETBYT    window left = X
         STX   WNDLFT
         LDA   #0
         STA   CH
         JSR   CHKCOM
         JSR   GETBYT    window top = Y
         STX   WNDTOP
         STX   CV
         JSR   CHKCOM
         JSR   GETBYT    window width = dX
         STX   WNDWDTH
         JSR   CHKCOM
         JSR   GETBYT    window height = dY
         CLC
         TXA
         ADC   WNDTOP    bottom = Y + dY
         STA   WNDBTM
         JSR   VTAB      realize cursor
         JSR   CROUT     carriage return
```

JSR   HOME   clear   screen   window,   home
                    cursor
JMP   DATA   . . .meanwhile, back in Apple-
                    soft

To use, set the ampersand vector at $3F5 to jump to SETWND, then you can call it by

& X,Y,DX,DY

where X and Y are the cursor locations for the upper left of the scroll window, DX is the width, and DY is the height of the window. For instance,

& 0,5,20,3

sets up a scroll window consisting of the leftmost twenty columns in the sixth through eighth rows. The zero sets the window to the first column and the five sets it to the sixth row. The width becomes 20 and the height is three.

There are two display character sets defined for the Apple; they are called *Primary* and *Alternate*. Of the two, the earlier Apples have only the Primary set available and the IIe model supports both sets. If your machine has the Alternate set, then you can display lowercase letters; otherwise you are restricted to the Primary set only. The Alternate set in the IIe is selected when you give

STA $C00F

from Assembler, or

POKE 49167,0

from Applesoft. All machines have the Primary set in order to be compatible, but on some Apples fitted with lower-case adaptors, you

may have lowercase in an alternate set. With the Apple IIe and most terminal arrangements, you use a CAPS LOCK key to switch between the lowercase and uppercase as described in Chapter Eight. Only the IIe has the soft switch at $C00F to select alternate characters. The Assembly instruction

STA $C00E

or the Applesoft

POKE 49116,0

will then change the character set from Alternate back to Primary.

Regardless of the set chosen, COUT should handle the display for you properly. If you choose the Primary set, you can display in normal, inverse, or flashing mode. Otherwise if you choose the Alternate set, you must display in normal or inverse mode only. Use the Applesoft command or set the INVFLG value at $32 in Page One:

to $FF for normal white-on-black
to $3F for inverse black-on-white
to $7F for flashing, Primary Set only

Then send the negative-ASCII characters to the video display routines at COUT1, either directly (Primary only) or via COUT (both Primary and Alternate).

If you want to write your own display routines, then you need to know the character display scheme. See Table 6-2.

The character display hardware decodes the character byte in two chunks — format and character code. The most significant bit selects normal if it is on and inverse if it is off. The next bit, bit 6, depends on the setting of $C00E/C00F soft switch (model IIe only). In the Primary set, if the bit is on it will cause the character to flash. In the Alternate set, if the bit is on it selects another set of characters to be used by the remaining six bits.

The six least significant bits of your character lookup the display pattern in the character set ROM. Each character ROM has 64 entries from $00 to $3F. The one that is selected by your character code is displayed on the screen. Older Apples have only one ROM which is the primary characters, while the IIe model has two ROMs, which are the primary and secondary. These codes are summarized in Table 6-2.

Table 6-2. Display Routines

	Primary	Secondary	Alternate
SIXBIT 00.0F 10.1F 20.2F 30.3F	@A B C D E F G H I J K L M N O P Q R S T U V W X Y Z — ! " # $ % & ' () * + , – . / 0 1 2 3 4 5 6 7 8 9 : ; = ?	@A B C D E F G H I J K L M N O P Q R S T U V W X Y Z — a b c d e f g h i j k l m n o p q r s t u v w x y z	
TWOBIT 00 01 10 11	Inverse Flash Normal Normal	.	Inverse (Pri) Inverse (Sec) Normal (Pri) Normal (Sec)
BYTE 00.3F 40.7F 80.BF C0.FF	Inverse Primary Flash Primary Normal Primary Normal Primary		Inverse Primary Inverse Secondary Normal Primary Normal Secondary

NOTE: The traditional Apple display supported in COUT1 uses the INVFLG to mask the character byte for the Primary Set. On the newer model IIe, the two bits can be used for the Alternate Set to get lower case letters instead of flashing.

If you want to display on Screen Two in text, then you must supply your own display routine; the Monitor won't work with Screen Two. First, ensure that Applesoft won't load and run at $801 as it usually does. Instead, force it to begin RAM access at $C00 by a

```
LDA #$0C      set lowest RAM pointer
STA TXTTAB + 1
LDA #$01
STA TXTTAB
LDA #0        zero first location
STA $0C00
```

preferably in a BRUN setup program before your Applesoft program is run. The HELLO program could do this, with an altered DOS to allow binary HELLO programs; see Chapter Seven on how to do this.

In your Applesoft program, you can

<div align="center">

POKE 49237,0

</div>

to switch the display from Screen One to Screen Two. You may wish to clear Screen Two ($0800.0BFF) with blanks ($A0) or copy a screen layout to it from Screen One ($0400.07FF) before using it. Then, you can switch back to normal Screen One display whenever you wish by:

<div align="center">

POKE 49236,0

</div>

This scheme will let you fill in a form on one screen while using the entire second screen for prompt menus. Or you may just want separate screens for a two-player game like Battleship where the players can take turns at the Apple without seeing each other's screen. All you need to start exploring Screen Two usage is a simple display routine like

```
DISPL     JSR  GETBYT      row number 0. . .23
          TXA
          ASL  A           times 2 for index
          TAX
          LDA  SCREEN2,X
          STA  A1
          LDA  SCREEN2 + 1,X
```

```
              STA  A1+1
              JSR  CHKCOM
              JSR  GETBYT        column number 0. . .39
              STX  A2
              JSR  CHKCOM
              JSR  CHRGET
              LDY  A2
              CMP  #$22          open quote
              BEQ  DISPL1
              JMP  SNERR
DISPL1        JSR  CHRGET        while not close quote
              CMP  #$22             get literals and
              BEQ  DISPL2           display on Screen Two
              ORA  #$80
              STA  (A1),Y
              INY
              BNE  DISPL1        endwhile
DISPL2        JMP  DATA


SCREEN2       DW   $0800         Row 0 (first row)
              DW   $0880         Row 1 (second row)
              DW   $0900         Row 2 (third row)
              . . .
              DW   $0BD0         Row 23 (24th row)
```

vectored from the ampersand at $3F5. The call sequence is just *& row-value, columnvalue, "literalstring"* from Applesoft. For example,

<div align="center">& 0, 16, "SELECTION"</div>

would display the string in the top row beginning with the seventeenth column (column 16). The quotes are needed to pass literal values to keep Applesoft from parsing inside the string.

When you get this one going, you can write more Screen Two routines for yourself. A spare screen is very handy when programming for heavy user interaction. See Table 6-3.

Sometimes, when you are using a *modem,* you'll want to save a text screen to disk. A better idea would be to buffer the incoming text and save it in that more compact form, but in a pinch you can just save the screen by

Table 6-3. Text Row Addresses

VTAB	Screen One		Screen Two	
Row	Dec	Hex	Dec	Hex
1	1024	0400	2048	0800
2	1152	0480	2176	0880
3	1280	0500	2304	0900
4	1408	0580	2072	0980
5	1536	0600	2560	0A00
6	1664	0680	2688	0A80
7	1792	0700	2816	0B00
8	1920	0780	2944	0B80
9	1064	0428	2088	0828
10	1192	04A8	2216	08A8
11	1320	0528	2344	0928
12	1448	05A8	2472	09A8
13	1576	0628	2600	0A28
14	1704	06A8	2728	0AA8
15	1832	07A8	2856	0B28
16	1960	07A8	2984	0BA8
17	1104	0450	2128	0850
18	1232	04D0	2256	08D0
19	1360	0550	2384	0950
20	1488	05D0	2512	09D0
21	1616	0650	2640	0A50
22	1744	06D0	2768	0AD0
23	1872	0750	2896	0B50
24	2000	07D0	3024	0BD0

BSAVE SCREEN, A$400,L$400

without any problem. However, you must be careful about how you read it back into memory later on.

The problem with BLOADing screens into the Screen One area, $400.7FF, is that the area is shared by peripheral devices. All bytes aren't screen display, and if you check in Chapter Two, you'll find some locations designated as *peripheral scratchpad*. So, if you overwrite $400.7FF with old data from disk, the current peripheral RAM data will be destroyed. Don't do it.

The solution is to use Screen Two. With Screen Two memory protected with the TXTTAB alteration given earlier, you can recall your saved screen from disk by

<div align="center">

BLOAD SCREEN, A$800
POKE 49237,0

</div>

and view it until you type

<div align="center">

POKE 49236,0

</div>

to redisplay Screen One. It's a little more trouble, but it is safer; you won't hang up by clobbering the peripherals.

6.2 GRAPHICS

6.2.1 Lo-Res Graphics

The LORES graphics uses the same memory as does the text display. Most LORES is confined to Screen One where Applesoft and Monitor routines are available to manage it. Chiefly, LORES finds its greatest application in arcade type games using the game paddles. But it also works well in Kaleidoscopes, plotting bar graphs, and shape creation programs.

If you write a LORES program, you should first try it in Applesoft. Where Applesoft is too slow, you can re-write it in Integer or write utilities in Assembly. The commands for Integer are the same as those for Applesoft, so just refer to Chapter Five for Integer BASIC details as you write. Here's how to use the commands with Applesoft. See Table 6-4.

<div align="center">

Table 6-4. Applesoft LORES

</div>

GR	Set and clear to graphics
COLOR = N	Set current plotting color
PLOT X,Y	Plot a pixel
HLIN X1, X2 AT Y	Plot a horizontal line
VLIN Y1, Y2 AT X	Plot a vertical line
N = SCRN (X, Y)	Identify screen color at pixel
TEXT: HOME	Set and clear to text

To initialize LORES for a full screen display, write:

```
TEXT
GR
POKE 49234,0
CALL 63538
```

The TEXT command makes sure any parameters and switches are reset: GR initializes the HIRES display; the POKE switches from mixed to full-screen graphics; and the CALL clears the full LORES screen to black.

To initialize the LORES screen for a mixed screen with the bottom-most four text lines intact, write:

```
TEXT
GR
POKE 34,20
```

Here, you use the mixed display with 40 lines of LORES followed by four rows of text. The POKE sets the scroll window to protect the graphics area of the screen.

You can remove LORES at any time, returning to text display with the TEXT command.

Here are the LORES drawing commands. The variables used here are: X and Y for the current position, X1 and Y1 for a beginning point, X2 and Y2 for an ending point, M is the slope of a line, DX and DY are the increments of cursor movement.

Always set the color before drawing, using the COLOR = command. Use one of the values in Table 6-5, zero to fifteen only.

- A vertical line is VLIN X1,X2 AT Y
- A horizontal line is HLIN Y1,Y2 AT X
- A single point is PLOT X,Y.

If you are PLOTting a cursor, then use a subroutine like

```
COLOR = BK
PLOT X,Y
X = X + DX : Y = Y + DY
BK = SCRN ( X, Y )
```

```
COLR = CU
PLOT X,Y
RETURN
```

that uses variable BK to remember the background color at the cursor position. You call it with the current cursor position (X,Y) and the increment (DX,DY). You initially set CU to the cursor color and BK to the background color.

If you want to draw sloping lines, you will need a straight line equation.

```
M = (Y2 – Y1)/(X2 – X1)
FOR X = X1 to X2
Y = M*(X – X1) + Y1
NEXT
```

This equation is crude but workable; don't use it on vertical lines.

LORES from Assembly is basically just another display character set for the text screens. Only instead of selecting one of sixty-four character patterns, LORES hardware displays two *pixels* for each

Table 6-5. LORES Colors

Dec	Hex	Color
00	00	Black
01	01	Magenta
02	02	Dark blue
03	03	Purple
04	04	Dark green
05	05	Grey 1
06	06	Medium blue
07	07	Light blue
08	08	Brown
09	09	Orange
10	0A	Grey 2
11	0B	Pink
12	0C	Light green
13	0D	Yellow
14	0E	Aquamarine
15	0F	White

character position. Each pixel is a small square of color on the screen. In the character position, the upper square comes from the four least significant bits while the lower square comes from the most significant bits. For example, if you type POKE 49232,0:POKE 49238,0 to Applesoft you will see the screen in LORES. Blanks, which have the negative-ASCII code of $A0, appear as grey over black: grey is color $A, black is color $0. Type POKE 49233,0 to switch back to text characters.

Like text characters, HIRES pixels appear 40 in each row. But because there are two pixels in each character, there are 48 pixels vertically in the 24 rows. In mixed mode the four bottom rows are text characters together with the top twenty rows of 40 pixels. This results in a 40-by-40-pixel display atop four text lines for the mixed mode and a 40-by-48-pixel display for the unmixed mode. You can switch to unmixed mode by

BIT $C052   unmixed graphics

or to mixed graphics and text by

BIT $C053   mixed graphics

anytime in an Assembly routine.

With this in mind you can write LORES graphics utilities for yourself. Just use the Monitor LORES Address in Table 6-6 to lookup the routine that you need. These are the same Monitor routines that the Applesoft commands use to execute LORES instructions, so you can use them yourself in Assembly to speed things up. Direct screen access to the $400.7FF area is possible but rarely necessary as the Monitor routines draw quite quickly.

6.2.2 Hi-Res Graphics

To initialize HIRES graphics use the HGR or the HGR2 commands in Applesoft, then the TEXT command to switch back to the Text screen. Because the text is in a different part of the memory than the graphics, you also switch back and forth between text and graphics with the soft switches so as not to clear the screen with the Applesoft commands. So, it's useful to have a complete set of initialization routines for each screen configuration at hand. Especially in Assembly programming, you can just JSR for each screen configuration as you need it.

Table 6-6. Monitor LORES Addresses

Hex	Label	Description
F800	PLOT	Plot pixel at (Y-reg, A-reg)
F819	HLINE	Plot horizontal, Y-reg to H2 at A-reg
F828	VLINE	Plot vertical, A-reg to V2 at Y-reg
F832	CLRSCR	Clear entire 48 by 40 pixel screen to black
F836	CLRTOP	Clear topmost 40 by 40 pixels to black
F864	SETCOL	Set COLOR according to A-reg
F871	SCRN	Get color of pixel at (Y-reg,A-reg) to A-reg
FB2F	INIT	Set TEXT modes
FB40	SETGR	Set GR modes
		Soft Switches for LORES
C050	GR	Set graphics display
C051	TEXT	Set text display
C052	UNMIX	Set for 40 by 48 pixels, no text
C053	MIX	Set for 40 by 40 pixels above four rows of text
C056	LORES	Set to ensure LORES display instead of HIRES
C054	SCREEN1	Set to ensure SCREEN1 display instead of SCREEN2
C055	SCREEN2	Set to ensure SCREEN2 display instead (rare)

To switch to the HIRES1 screen with mixed graphics and text, you can:

```
MIX1            STA   $C050
                STA   $C053
                STA   $C054
                STA   $C057
                RTS
```

Then, to switch to full graphics on HIRES1, you would write:

```
FULL1           STA   $C050
                STA   $C052
                STA   $C054
                STA   $C057
                RTS
```

Similarly, you would invoke full graphics on HIRES2 by:

```
FULL2             STA   $C050
                  STA   $C052
                  STA   $C055
                  STA   $C057
                  RTS
```

The mixed display for HIRES2 is rarely used because of the need for text in Screen Two ($800.BFF), but here it is:

```
MIX2              STA   $C050
                  STA   $C053
                  STA   $C055
                  STA   $C057
                  RTS
```

Finally, to switch back to the text only on Screen One, you use this set of switches:

```
TEXT1             STA   $C051
                  STA   $C053
                  STA   $C054
                  STA   $C056
                  RTS
```

By having these routines in your Assembly HIRES routines, you can JSR simply to ensure that the soft switches are all set properly each time you want to change displays.

You can have a separate routine to clear the HIRES screen to black or any other color for that matter. The simplest routine is just a call sequence to the Monitor's MOVE routine at $FE2C:

```
CLEAR1            LDA   #0        for black1
                  STA   $4000     clearing HIRES1
                  STA   A4
                  LDA   #1
                  STA   A1
                  LDA   #$40
                  STA   A1 + 1
```

```
STA   A4 + 1
LDA   #$FE
STA   A2
LDA   #$7F
STA   A2 + 1
JMP   MOVE
```

Write a call sequence to CLEAR2 that will clear HIRES2.

One of the things you can do in HIRES is to make your own character set. Usually, you must use the mixed mode and confine your labels to the bottom of the screen. With your own HIRES character set, you can use the full HIRES screen and put labels wherever you wish. See Example 6-1.

The HIRES display addresses are grouped into rows and columns just like the text addresses. There are twenty-four rows by forty columns. Each character position, however, has eight bytes of memory in HIRES as compared to one byte in text. These eight bytes can display any character you want, explicitly. Each byte displays to a different line on the screen, so the eight bytes display eight lines, one below the other, to makeup the character. Look at Table 6-7 which lists the HIRES row addresses. Each address given is for the top line of the leftmost character position of each row. Then look at Table 6-8 which lists the HIRES line offsets. These line offsets are the values to add to the top line address to address the remaining seven lines in each row. Also, the column number must be added if you want to reach any character position on the screen. By working through the additions of row address, line offset, and column number, you can access the HIRES screen by character position. In fact, the sum can be shown as a formula

byte address = row address + line offset + column number

for use in writing character access routines.

Each character to be displayed on a HIRES screen must be kept in eight corresponding bytes. Each byte will be used to display on one line of the row within the desired character position. In HIRES display, a byte will map to seven points on the screen. Each of these seven points can be turned on (white) or off (black) by setting the corresponding bit in the byte. Display points that can be independently con-

Example 6-1.

```
SOURCE FILE: EXAMPLE 6.1
0000:              1 **********************************
0000:              2 * EXAMPLE 6.1                    *
0000:              3 *                                *
0000:              4 *   HIRES CHARACTER DISPLAY      *
0000:              5 *                                *
0000:              6 * DISPLAY A CHARACTER AT (X,Y)   *
0000:              7 * FROM CHAR. TABLE 8 BYTES EACH. *
0000:              8 * A-REG = CHAR. CODE 0...255     *
0000:              9 * Y-REG = ROW NUMBER 0...23      *
0000:             10 * X-REG = COLUMN NUMBER 0...39   *
0000:             11 *                                *
0000:             12 * RESULT IS DISPLAYED ON HIRES1. *
0000:             13 **********************************
0000:             14 *
0000:             15 *
0000:             16 *      E Q U A T E S
0000:             17 *
0050:             18 ZSCREEN EQU $50       HIRES ADDRE
SS
0052:             19 ZCHAR   EQU $52       CHAR. TABLE
 ADDRESS
0000:             20 *
1800:             21 CHAR    EQU $1800     CHARACTER T
ABLE
0000:             22 *
0000:             23 *
0000:             24 *      R O U T I N E S
0000:             25 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 6.1.OBJ0

8000:             26         ORG  $8000
8000:             27 *
8000:             28 * WITH Y-REG, LOOKUP ROW ADDRESS.
8000:             29 *
8000:48           30 HCHAR   PHA
8001:98           31         TYA
8002:0A           32         ASL  A        MULT BY 2 F
OR INDEX
8003:A8           33         TAY
8004:B9 52 80     34         LDA  ROW,Y    GET THE ADD
R-LO
8007:85 50        35         STA  ZSCREEN
8009:B9 53 80     36         LDA  ROW+1,Y  GET THE ADD
R-HI
800C:85 51        37         STA  ZSCREEN+1
800E:             38 *
800E:             39 * WITH X-REG, OFFSET ROW ADDRESS
```

Example 6-1 Cont.

```
800E:              40 * BY THE COLUMN NUMBER.
800E:              41 *
800E:8A            42       TXA
800F:18            43       CLC
8010:65 50         44       ADC   ZSCREEN    ADDS L.S.BY
TE
8012:85 50         45       STA   ZSCREEN
8014:A9 00         46       LDA   #0
8016:65 51         47       ADC   ZSCREEN+1  ADDS M.S.BY
TE
8018:85 51         48       STA   ZSCREEN+1
801A:              49 *
801A:              50 * WITH A-REG, LOOKUP THE CHARACTER C
ODE AS
801A:              51 *   (ZCHAR) = 8*(A-REG) + CHAR
801A:              52 *
801A:68            53       PLA
801B:85 52         54       STA   ZCHAR      LOW BYTE
801D:A9 00         55       LDA   #0
801F:85 53         56       STA   ZCHAR+1    HIGH BYTE
8021:06 52         57       ASL   ZCHAR      MULTIPLY BY
8
8023:26 53         58       ROL   ZCHAR+1
8025:06 52         59       ASL   ZCHAR
8027:26 53         60       ROL   ZCHAR+1
8029:06 52         61       ASL   ZCHAR
802B:26 53         62       ROL   ZCHAR+1
802D:18            63       CLC
802E:A5 52         64       LDA   ZCHAR      THEN ADD TH
E ADDRESS
8030:69 00         65       ADC   #>CHAR     OF CHARACTE
R TABLE
8032:85 52         66       STA   ZCHAR
8034:A5 53         67       LDA   ZCHAR+1    TO GET ENTR
Y ADDRESS.
8036:69 18         68       ADC   #<CHAR
8038:85 53         69       STA   ZCHAR+1
803A:              70 *
803A:              71 * DISPLAY THE CHARACTER IN 8 LINES
803A:              72 * AT THE ZSCREEN POSITION.
803A:              73 *
803A:A0 00         74       LDY   #0
803C:A2 00         75       LDX   #0
803E:B1 52         76 HCHAR1 LDA  (ZCHAR),Y  FROM TABLE
8040:81 50         77       STA   (ZSCREEN,X) TO HIRES1.
8042:C8            78       INY
8043:C0 08         79       CPY   #8         DO 8 TIMES.
```

Example 6-1 Cont.

```
8045:F0 0A     80              BEQ   HCHAR2
8047:18        81              CLC
8048:A5 51     82              LDA   ZSCREEN+1    CALCULATE A
DDRESS OF
804A:69 04     83              ADC   #4           NEXT LINE I
N THE ROW.
804C:85 51     84              STA   ZSCREEN+1
804E:4C 3E 80  85              JMP   HCHAR1
8051:60        86 HCHAR2       RTS
8052:          87 *
8052:          88 *
8052:          89 *    L I T E R A L S
8052:          90 *
8052:          91 * HIRES1 CHARACTER ROW TABLE.
8052:          92 *
8052:00 20     93 ROW          DW    $2000        ROW   0
8054:80 20     94              DW    $2080        ROW   1
8056:00 21     95              DW    $2100        ROW   2
8058:80 21     96              DW    $2180        ROW   3
805A:00 22     97              DW    $2200        ROW   4
805C:80 22     98              DW    $2280        ROW   5
805E:00 23     99              DW    $2300        ROW   6
8060:80 23     100             DW    $2380        ROW   7
8062:28 20     101             DW    $2028        ROW   8
8064:A8 20     102             DW    $20A8        ROW   9
8066:28 21     103             DW    $2128        ROW  10
8068:A8 21     104             DW    $21A8        ROW  11
806A:28 22     105             DW    $2228        ROW  12
806C:A8 22     106             DW    $22A8        ROW  13
806E:28 23     107             DW    $2328        ROW  14
8070:A8 23     108             DW    $23A8        ROW  15
8072:50 20     109             DW    $2050        ROW  16
8074:D0 20     110             DW    $20D0        ROW  17
8076:50 21     111             DW    $2150        ROW  18
8078:D0 21     112             DW    $21D0        ROW  19
807A:50 22     113             DW    $2250        ROW  20
807C:D0 22     114             DW    $22D0        ROW  21
807E:50 23     115             DW    $2350        ROW  22
8080:D0 23     116             DW    $23D0        ROW  23
8082:          117 *
8082:00        118             BRK
```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

trolled like this are called *pixels*. The least significant bit controls the
leftmost pixel: a one for white and a zero for black. The other bits are
mapped left to right in sequence on the screen, so that bit 6 controls
the seventh pixel which is the rightmost. Usually, bit 7 is kept off. If
you turn it on, all the pixels will shift position by half a point. Because
you usually clear a screen with zeros, it is best to always leave bit 7 off
in all bytes that keep black-and-white pixels. Color is different; bit 7

Table 6-7. HIRES Row Addresses

Row	HIRES1 Dec	HIRES1 Hex	HIRES2 Dec	HIRES2 Hex
0	8192	2000	16384	4000
1	8320	2080	16512	4080
2	8448	2100	16640	4100
3	8576	2180	16768	4180
4	8704	2200	16896	4200
5	8832	2280	17024	4280
6	8960	2300	17152	4300
7	9088	2380	17280	4380
8	8232	2028	16424	4028
9	8360	20A8	16552	40A8
10	8488	2128	16680	4128
11	8616	21A8	16808	41A8
12	8744	2228	16936	4228
13	8872	22A8	17064	42A8
14	9000	2328	17192	4328
15	9128	23A8	17320	43A8
16	8272	20F0	16464	40F0
17	8400	20D0	16592	40D0
18	8528	2150	16720	4150
19	8656	21D0	16848	41D0
20	8784	2250	16976	4250
21	8912	22D0	17104	42D0
22	9040	2350	17232	4350
23	9168	23D0	17360	43D0

Table 6-8. HIRES Line Offsets

Line	Dec	Hex
0	0	0
1	1024	0400
2	2048	0800
3	3072	0C00
4	4096	1000
5	5120	1400
6	6144	1800
7	7168	1C00

used for color is often called the color bit. But for characters, always turn it off.

Eight bytes contain one character, seven pixels across by eight high. To have all byte values, a table of 256 characters needs two K of

memory; $1800.1FFF. To fetch any character, a lookup formula for eight bytes per entry will do the trick

entry address = 8 * char. code + table address

in getting the address of the top byte.

The routine listed here is HCHAR and it displays a character from your table. You must have the table in memory (EQUated to $1800 here), and pass the row number in the X-reg, the column number in the Y-reg, and the character code in the A-reg. This character code is the entry number for your character table. HCHAR displays your character at the row and column you gave and then returns with the registers clobbered.

All you have to do to get HCHAR working for you, aside from keying it in, is to create your characters. Here's how.

The secret to getting character layouts is to keep in mind that the bits run from *left* to *right* on the screen although they are represented in binary from *right* to *left* (see Fig. 6-2). The critical step is translating the pattern of squares to binary (see Fig. 6-3). Start with bit 7 on the right of the layout. Translate into a bit (0 or 1).Then do bit 6 to the left of bit 7 on the layout but to the right of bit 7 in the binary number. Study the three examples, especially Fig. 6-3B which is non-symmetrical.
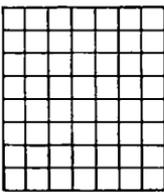
(A) The HIRES character.   (B) Text character area.
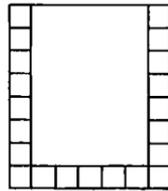
Fig. 6-2. Character layouts.

When laying out, keep text characters within the five by seven area shown to avoid the characters *bleeding* together on the screen. You may want some special characters to join together, but text should not. The one exception that you may have to make is for the descenders on lowercase letters. For example, see the "j" layout in Fig. 6-3.

(A) Character A.


(B) Number 3.

(C) Character J.

Fig. 6-3. Creating characters.

You can write HIRES graphics routines that execute much faster than the same routines in Applesoft. Avoiding the time taken to parse instructions and setup parameters is only a part of the savings in time. The big payoff is in speed when you bypass the lengthy calculations that Applesoft and the Applesoft programs must make to get useful graphics.

The problem with HIRES graphics is its coordinate system. Because it has its origin at the upper left, the horizontal values from zero to 279 and the vertical values from zero to 191 present several problems. First, the horizontal value requires two bytes of storage for just a little over one byte of data, not too efficient in an eight bit machine. Then the integral values are rather arbitrary and always require scaling after

calculations, especially trigonometric functions. Then you may have to reflect the vertical value because the plotting takes place in the fourth quadrant instead of the usual first. These coordinates simply follow the text screen convention of row numbering and use the number of points vertically and horizontally for the convenience of the original HIRES programming. But you don't have to follow that arrangement; there are better ways.

The simplest coordinate system you can use for HIRES work is one which uses one byte for the X-coordinate and one byte for the Y-coordinate (see Fig. 6-4A). The origin you take to be at the lower left corner of the full screen. Each coordinate represents a binary fraction — in X from zero to one, in Y from zero to three-fourths. This is called the system of *normalized coordinates* (see Fig. 6-4B).
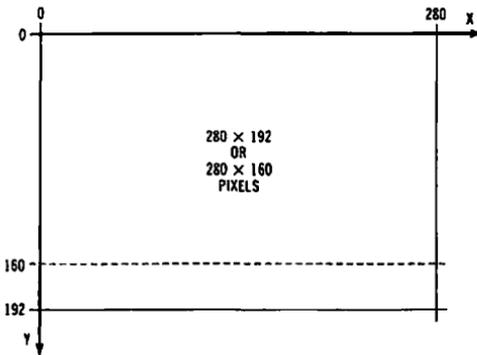
(A) Applesoft HIRES coordinates.
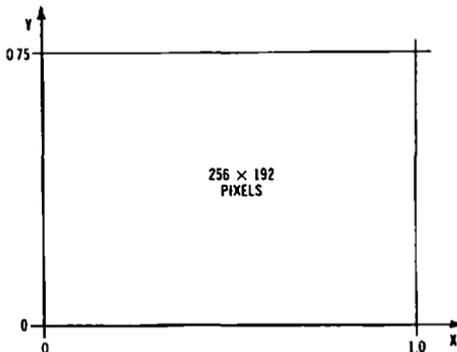
(B) Normalized HIRES coordinates.

Fig. 6-4. Coordinate system.

With normalized coordinates, you can plot any one of 256 horizontal positions and any one of 192 vertical positions. The 24 remaining horizontal positions are lost as they are used to provide the single byte X-coordinate. Normalized coordinates with a three-fourths vertical size are compatible with graphics communications standards like Videotex and Teletext. They get along with the input data requirements of many commercially available graphics packages. And, they can be used with Applesoft's floating-point routines with a minimum of scaling required. If you want HIRES routines in Assembly for speed and simplicity, then adopt normalized coordinates right away and don't use the built-in coordinate system.

Here's how normalized coordinates work. Keep your current coordinates in the X and Y registers. To plot a point, test the coordinate by

```
        CPY  #$C0           rangetest Y
        BCS  CLIP
```

where CLIP is a return location that does not plot the point. If the Y-coordinate is less than three-fourths ($C0 in hex) then it passes the test and you can plot it. This test is called *clipping*, and is the only clipping you need to do. In Applesoft, you would have to make *four* clipping tests. Clipping makes sure that you don't plot outside the screen area and any intelligent graphics plotter must do it. With normalized coordinates, it's simple.

To plot any point on the HIRES1 screen using normalized coordinates, you need your own routines. Such routines are simple to write, especially using table lookup for screen addressing and other functions. See Table 6-9. A complete table lookup routine to plot a pixel in black or white will use about one K of memory. You can use Example 6-2 (located at the end of this chapter) which will keep the coordinates in the X-reg and Y-reg, and the A-reg will keep the pixel value — say, zero for black and nonzero for white. Such a routine would look like this:

```
* HIRES PIXEL — PLOT A__REG PIXEL AT (X__REG,Y__REG)
HPIX    PHP
        CPY  #$C0           clip Y-coordinate
        BCS  HPIX2
        CMP  #0             If pixel value = zero.
        BNE  HIPIX1
```

```
        JSR  BLACK    then plot black pixel
        BEQ  HPIX2
HPIX1   JSR  WHITE    else plot white pixel.
HPIX2   PLP
        RTS
```

In this routine, all registers are preserved so you can use the routine in a loop. Then you can vary the X-reg, Y-reg, and the A-reg as you want and branch after any JSR with impunity. Each of the BLACK and WHITE routines also preserves registers to support your calling routine this way.

To understand how these BLACK and WHITE routines work, you must study the tables in the LITERAL section. By comparing LOLINE and HILINE with Table 6-9, HIRES1 Screen Lines, you can see that they are each one byte tables of the low bytes and high bytes of the screen addresses. In each, the first entry is for the bottom left screen address; the last entry is for the top left screen address. Using the Y-reg as index, the routines get the line address for the pixel from HILINE and LOLINE to the Page Zero pointer, SCREEN. This leaves only the X-coordinate to interpret. Now the X-reg is used to index two tables called DIV7 and BWPIX. First, DIV7 does a division by seven by table lookup instead of by algorithm, which is much faster. Second, BWPIX gives the mask byte to select the desired pixel bit from within the screen byte. With these four tables — LOLINE, HILINE, DIV7, and BWPIX — the routines can lookup their functions quickly.

After saving the registers, the routine gets the address of the line from LOLINE and HILINE to SCREEN. Then it finds the byte containing the X-th pixel on the line by DIV7 and puts this quotient into the Y-reg as the byte index. Finally, the mask for the bit within that byte is looked up in BWPIX with the X-reg; used to mask the screen; and that's it. BLACK masks the screen in a different way than does WHITE since BLACK must turn the pixel's bit OFF, and WHITE must turn it ON. Otherwise, the two routines work the same way.

The ability to access each pixel on the screen with separate bits is unique to HIRES black or white plotting. With color, you cannot reach pixels uniquely with separate bits like this, so you'll probably only use these routines satisfactorily with a black and white video display.

Table 6-9. HIRES1 Screen Lines

Line	Addr.	Line	Addr.	Line	Addr.	Line	Addr.	Line	Addr.
00	3FD0	28	3D50	50	3EA8	78	3C28	A0	3D80
01	3BD0	29	3950	51	3AA8	79	3828	A1	3980
02	37D0	2A	3550	52	36A8	7A	3428	A2	3580
03	33D0	2B	3150	53	3228	7B	3028	A3	3180
04	2FD0	2C	2D50	54	2EA8	7C	2C28	A4	2D80
05	2BD0	2D	2950	55	2AA8	7D	2828	A5	2980
06	27D0	2E	2550	56	26A8	7E	2428	A6	2580
07	23D0	2F	2150	57	22A8	7F	2028	A7	2180
08	3F50	30	3CD0	58	3E28	80	3F80	A8	3D00
09	3B50	31	38D0	59	3A28	81	3B80	A9	3900
0A	3750	32	34D0	5A	3628	82	3780	AA	3500
0B	3350	33	30D0	5B	3228	83	3380	AB	3100
0C	2F50	34	2CD0	5C	2E28	84	2F80	AC	2D00
0D	2B50	35	28D0	5D	2A28	85	2B80	AD	2900
0E	2750	36	24D0	5E	2628	86	2780	AE	2500
0F	2350	37	20D0	5F	2228	87	2380	AF	2100
10	3ED0	38	3C50	60	3DA8	88	3F00	B0	3C80
11	3AD0	39	3850	61	39A8	89	3B00	B1	3880
12	36D0	3A	3450	62	35A8	8A	3700	B2	3480
13	32D0	3B	3050	63	31A8	8B	3300	B3	3080
14	2ED0	3C	2C50	64	2DA8	8C	2F00	B4	2C80
15	2AD0	3D	2850	65	29A8	8D	2B00	B5	2880
16	26D0	3E	2450	66	25A8	8E	2700	B6	2480
17	22D0	3F	2050	67	21A8	8F	2300	B7	2080
18	3E50	40	3FA8	68	3D28	90	3E80	B8	3C00
19	3A50	41	3BA8	69	3928	91	3A80	B9	3800
1A	3650	42	37A8	6A	3528	92	3680	BA	3400
1B	3250	43	33A8	6B	3128	93	3280	BB	3000
1C	2E50	44	2FA8	6C	2D28	94	2E80	BC	2C00
1D	2A50	45	2BA8	6D	2928	95	2A80	BD	2800
1E	2650	46	27A8	6E	2528	96	2680	BE	2400
1F	2250	47	23A8	6F	2128	97	2280	BF	2000
20	3DD0	48	3F28	70	3CA8	98	3E00		
21	39D0	49	3B28	71	38A8	99	3A00		
22	35D0	4A	3728	72	34A8	9A	3600		
23	31D0	4B	3328	73	30A8	9B	3200		
24	2DD0	4C	2F28	74	2CA8	9C	2E00		
25	29D0	4D	2B28	75	28A8	9D	2A00		
26	25D0	4E	2728	76	24A8	9E	2600		
27	21D0	4F	2328	77	20A8	9F	2200		

NOTE: Each address is leftmost byte. Line Zero is at bottom.

6.2.3 Mid-Res Graphics

The Apple II can plot six different colors on the HIRES screens. Or, it can plot 280-by-192 independent pixels. But it cannot do both at the same time.

When you want to draw in color, you would expect to do so in the same way you drew in black and white. You set the HCOLOR to your choice and start drawing, but it doesn't work like it should. Some vertical lines may disappear; diagonals look jagged, and some color

combinations won't work side-by-side on the screen. Something is wrong, but it is not apparent exactly what. You certainly cannot plot colors the same way you plotted black and white; it just won't work.

Plotting colors by pixel, like plotting in black and white, is called the six-color problem. There is a solution if you want color pixels in HIRES. First, see how the HIRES colors work.

There are two sets of HIRES colors. One set has the colors black1, violet, green, and white1. The other set has the colors black2; blue, orange, and white2. The difference between the two is the high order bit in the bytes: the violet-green set has bit 7 clear; the blue-orange set has bit 7 set. Otherwise, they are exactly the same. In both sets, the colors are produced by alternately arranging the display bits, from bit 0 to bit 6, in an on-off or off-on pattern. Such a pattern takes two bytes to complete; for instance, violet is produced by the pattern

and   off-on-off-on-off-on-off

appearing in successive bytes on the line. See Fig. 6-5 for details of the violet-green pattern and the blue-orange pattern.

To draw a HIRES color, select a bit pattern from Table 6-10 and use it to fill in an area on the screen. For instance, the pattern for violet will be in byte pairs of $55 (left) and $2A (right). This gives us our colors in an area of fourteen points across each. Notice that turning all bits on gives white and turning all bits off gives black. For black and white, the color bit, bit 7, doesn't matter. This gives us the six colors, but with a very low horizontal resolution of fourteen dots out of 280. These bit patterns are used by Applesoft for the HCOLOR you

Table 6-10. Color Codes

Byte	Color	Left Value	Right Value
Violet-green	Black1	$00	$00
	Violet	$55	$2A
	Green	$2A	$55
	White1	$7F	$7F
Blue-orange	Black2	$80	$80
	Blue	$D5	$AA
	Orange	$AA	$D5
	White2	$FF	$FF
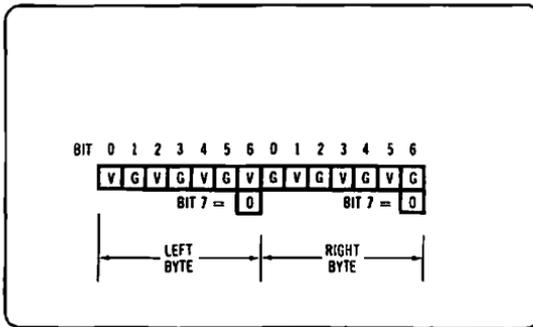
(A) Violet/green pattern.
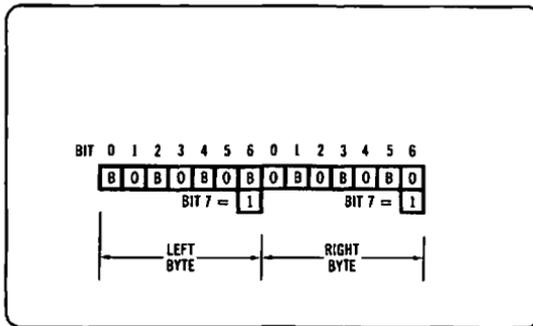
(B) Blue/orange pattern.

Fig. 6-5. HIRES color sets.

select. By keeping to the boundaries of fourteen points across, you can avoid having colors interfering with each other.

Look at HIRES color generation a little closer and you will see that colors used in pixels are much finer than colors in byte pairs of fourteen points.

When you plot dots alternately on and off, you create a 3.58-MHz square wave (see Fig. 6-6). This is the exact frequency of the video color subcarrier that the Apple generates, because the dot generator and the color burst generator both run from the same clock. Any 3.58-MHz signal in the video will be used by your tv set to generate color. The hue produced depends on the phase between the color burst and the video. Your video is the bit stream from the dot generator. If they are in phase, an orange line appears across the screen. If the difference is 180 degrees, a blue line appears. Changing the phase by 90

SIGNAL	WAVEFORM	PHASE
3.58 MHz REFERENCE		0°
VIOLET LINE		90°
BLUE LINE		180°
GREEN LINE		270°
ORANGE LINE		0°
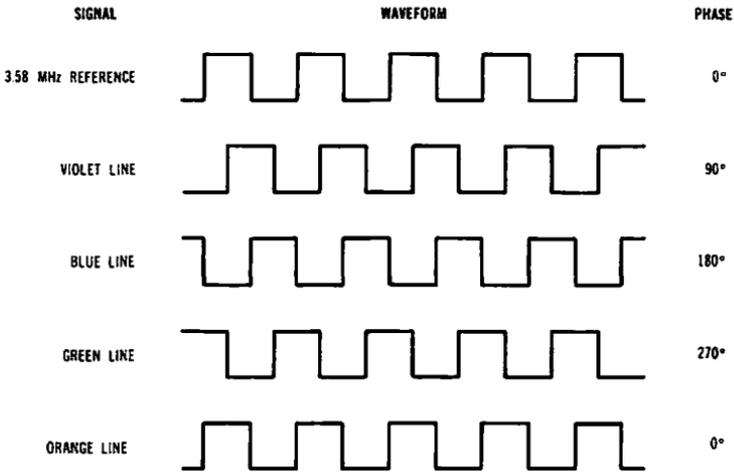
Fig. 6-6. HIRES color signals.

degrees gives green or violet. You can't do this directly, but bit 7 will do it for you for all seven display bits in its byte. With bit 7 off, the phases available make green and violet; with bit 7 on, they make blue and orange. You can see these four hues in the HIRES hues of Fig. 6-7.
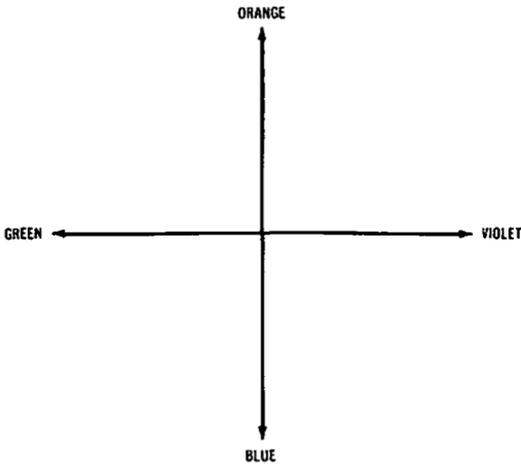

Fig. 6-7. The four HIRES hues.

The hangup is that any given byte must be switched to one of the two patterns: orange-blue or violet-green. You cannot mix between them in the same byte. And, considering that the bit pattern for any one color is different for left and right bytes (so that two bytes are needed to complete a color), you have no choice but to force all color bits to be either on or off. Any other way will cause trouble.

With this restriction, you can plot four colors by allotting two bits for each pixel. Each pixel will then be independent of the other. The trade off with black and white is the halving of horizontal resolution, from 280 to 140, and the gain is two colors. By staying within a four color set, you avoid colors at 90 degrees from each other contending for the color bit in the same byte.

You can have a color pixel scheme with any number of colors if you are willing to trade off the resolution. Since you have only half the horizontal resolution of the vertical resolution, the next logical trade-off is to halve the vertical resolution as well. This gives 140 by 96, coarser than HIRES (black and white) but still finer than LORES at 40 by 48. Call this scheme MIDRES, since it falls between the two.

The trick to getting more colors is to allow the color bit to vary from line to line. The restriction was that you couldn't use different color bits on the *same* line. So, set the color bit on alternate lines and see what happens.

For each pair of lines, set the color bit on the top and clear it on the bottom line as shown in Fig. 6-8. Then look at what happens when

```
... B O B O B O B O B O B O B O B O ...
... V G V G V G V G V G V G V G V G ...
... B O B O B O B O B O B O B O B O ...
... V G V G V G V G V G V G V G V G ...
... B O B O B O B O B O B O B O B O ...
... V G V G V G V G V G V G V G V G ...
... B O B O B O B O B O B O B O B O ...
... V G V G V G V G V G V G V G V G ...
... B O B O B O B O B O B O B O B O ...
...V G V G V G V G V G V G V G V G ...
```

B = blue.   O = orange
V = violet.   G = green

Mosaic of four colors on the HIRES· screen. By alternating blue orange with violet green on odd and even lines. contention for the color bit is avoided. Four bits in two or four bytes then control four dots to make a one-of-sixteen valued pixel — two by two each. Typical pixel is outlined here as a parallelogram.

Fig. 6-8. Varying the color bit from line to line.

you plot the four colors from each set on each line. There are sixteen possible combinations and each one gives a different color. (There are two greys.) For instance, orange on the top line with violet on the bottom line will blend to a brilliant pink color when viewed from a short distance. A blue and a green pair of lines will look aqua. Brown can be seen with orange and green. When white is used, you get a light color; when black is used, you get a dark color. The dark colors and greys show their lines clearly while the brilliant colors like pink and true blue have a somewhat textured look. All colors come from the field of lines having bit 7 set in all odd-line bytes and clear in all even-line bytes. These are summarized in Table 6-11.

The MIDRES colors use all four HIRES hues, so they have four additional hues where they combine. This gives you eight hues: orange, pink, violet, true blue, blue, aqua, green, and brown. You can see them as phase angles each 45 degrees apart in Fig. 6-9. Of these

Table 6-11. Creating Mid-res Colors

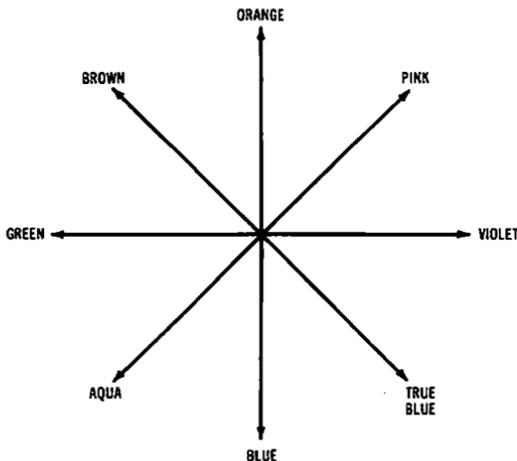|	Black2	Orange	Blue	White2
Black1	Black	Dark orange	Dark blue	Grey-2
Violet	Dark violet	Pink	True blue	Light violet
Green	Dark green	Brown	Aqua	Light green
White1	Grey-1	Light orange	Light blue	White

Fig. 6-9. The eight MIDRES hues.

eight hues, the four HIRES ones will display either light (when combined with a white) or dark (when combined with a black). Combining black with white can be done two ways as there are two greys.

Each of the sixteen colors has its own bit pattern as shown in Table 6-12. They need four bytes to keep them: two bytes of top line and two bytes of bottom line. Into each byte pair goes each combination of HIRES colors allowed on the line. Not shown in the patterns are the

Table 6-12. The Sixteen Mid-res Colors

Pixel	Bit Pattern 0123456	0123456	Color
0	0000000	0000000	Black
	0000000	0000000	
1	0000000	0000000	Dark violet
	1010101	0101010	
2	1010101	0101010	Dark blue
	0000000	0000000	
3	1010101	0101010	True blue
	1010101	0101010	
4	0000000	0000000	Dark green
	0101010	1010101	
5	0000000	0000000	Grey-1
	1111111	1111111	
6	1010101	0101010	Aqua
	0101010	1010101	
7	1010101	0101010	Light blue
	1111111	1111111	
8	0101010	1010101	Dark orange
	0000000	0000000	
9	0101010	1010101	Pink
	1010101	0101010	
10	1111111	1111111	Grey-2
	0000000	0000000	
11	1111111	1111111	Light violet
	1010101	0101010	
12	0101010	1010101	Brown
	0101010	1010101	
13	0101010	1010101	Light orange
	1111111	1111111	
14	1111111	1111111	Light green
	0101010	1010101	
15	1111111	1111111	White
	1111111	1111111	

color bits: top lines are always on, bottom lines always off. A numeric pixel value appears with each color whose four bits have the four HIRES colors encoded.

 bit 0 is violet
 bit 1 is blue
 bit 2 is green
 bit 3 is orange

With this scheme, 1 is dark violet, 2 is dark blue, 4 is dark green, and 8 is dark orange. Combinations like aqua are built from bits — aqua is bit 1 and bit 2 which is six. Such a scheme makes it easier for a routine to decode. The bit patterns then can be calculated or looked up using the HIRES bytes. Each set of four bytes has a color pixel value to identify it. All sixteen colors are listed in Table 6-12.

In the MIDRES pixels routine of Example 6-3, you can see how this scheme can be implemented. Call with normalized coordinates, the same way you do for black and white pixel plotting, but with the MIDRES color code in the A-reg. Notice that the routine clips the Y-coordinate for you.

Like the BLACK and WHITE routines of Example 6-2, this routine uses tables to reduce execution time to a minimum. It makes four plots, one for each byte, because the byte pattern repeats only every second byte in each line and there are two lines through each pixel. The CPLOT routine must do some fancy masking to plot only one pixel's bits within these bytes — it uses masks to do this.

While an analysis of how Example 6-3 works would take quite some time to work out, you can use it simply. Pass the X-coordinate in the X-reg, the Y-coordinate in the Y-reg, and the MIDRES color in the A-reg. The coordinates are normal, with the origin at the lower left of the full screen. The color is given in Table 6-12 and in COLOR at the end of the listing.

6.2.4 Shape Tables

One class of objects that you draw frequently on the HIRES graphics screens is shapes. Shapes include such sets of objects as characters, tokens, cursors, and missiles. What characterizes all these shapes is that they are defined without having any position. Ordinary drawings are plotted using the Applesoft PLOT command at specific

Example 6-2.

```
SOURCE FILE: EXAMPLE 6.2
0000:              1 ******************************
0000:              2 * EXAMPLE 6.2                *
0000:              3 *                            *
0000:              4 *  H I R E S    P I X E L S  *
0000:              5 *                            *
0000:              6 * CALL BLACK OR WHITE WITH   *
0000:              7 * CO-ORDS IN X-REG, Y-REG    *
0000:              8 * NORMAL ORIGIN LOWER LEFT   *
0000:              9 ******************************
0000:             10 *
0000:             11 *
0000:             12 *    E Q U A T E S
0000:             13 *
003C:             14 SCREEN  EQU  $3C          POINTER TO HIRES1
0000:             15 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 6.2.OBJ0
8000:             16          ORG  $8000
8000:             17 *
8000:             18 *    R O U T I N E S
8000:             19 *
8000:             20 * PLOT A WHITE HIRES PIXEL
8000:             21 * AT (X-REG,Y-REG)
8000:             22 *
8000:08           23 WHITE    PHP              SAVE REGISTERS
8001:48           24          PHA
8002:98           25          TYA
8003:48           26          PHA
8004:B9 FE 80     27          LDA  LOLINE,Y    USE Y-REG TO
8007:85 3C        28          STA  SCREEN      THE ADDRESS OF THE
8009:B9 3E 80     29          LDA  HILINE,Y    SCREEN LINE.
800C:85 3D        30          STA  SCREEN+1
800E:BD BE 81     31          LDA  DIV7,X      USE X-REG TO FIND
8011:A8           32          TAY              THE PIXEL'S BYTE.
8012:BD C1 82     33          LDA  BWPIX,X     USE X-REG AGAIN TO
8015:11 3C        34          ORA  (SCREEN),Y  FIND THE BIT MASK.
8017:91 3C        35          STA  (SCREEN),Y  TURN BIT ON!
8019:68           36          PLA
801A:A8           37          TAY
801B:68           38          PLA
801C:28           39          PLP              RESTORE REGISTERS
801D:60           40          RTS
801E:             41 *
801E:             42 *
801E:             43 * PLOT A HIRES PIXEL AS BLACK
801E:             44 * AT (X-REG,Y-REG)
801E:             45 *
801E:08           46 BLACK    PHP              SAVE REGISTERS
801F:48           47          PHA
8020:98           48          TYA
8021:48           49          PHA
8022:B9 FE 80     50          LDA  LOLINE,Y    USE Y-REG TO
```

Example 6-2 Cont.

```
8025:85 3C      51          STA     SCREEN       FIND THE ADDRESS OF
8027:B9 3E 80   52          LDA     HILINE,Y     THE SCREEN LINE.
802A:85 3D      53          STA     SCREEN+1
802C:BD BE 81   54          LDA     DIV7,X       USE X-REG TO FIND
802F:A8         55          TAY                  THE BYTE ON THE LINE.

8030:BD C1 82   56          LDA     BWPIX,X      USE X-REG TO FIND
8033:49 FF      57          EOR     #$FF         THE BIT MASK.
8035:31 3C      58          AND     (SCREEN),Y   TURN BIT OFF!
8037:91 3C      59          STA     (SCREEN),Y
8039:68         60          PLA
803A:A8         61          TAY
803B:68         62          PLA
803C:28         63          PLP                  RESTORE REGISTERS
803D:60         64          RTS
803E:           65  *
803E:           66  *
803E:           67  *    L I T E R A L S
803E:           68  *
803E:           69  *
803E:           70  * HIRES1 LINE ADDRESSES - HIGH
803E:           71  *
803E:3F 3B 37   72 HILINE   DFB  $3F,$3B,$37,$33,$2F,$2B,$27,$23
8041:33 2F 2B
8044:27 23
8046:3F 3B 37   73          DFB  $3F,$3B,$37,$33,$2F,$2B,$27,$23
8049:33 2F 2B
804C:27 23
804E:3E 3A 36   74          DFB  $3E,$3A,$36,$32,$2E,$2A,$26,$22
8051:32 2E 2A
8054:26 22
8056:3E 3A 36   75          DFB  $3E,$3A,$36,$32,$2E,$2A,$26,$22
8059:32 2E 2A
805C:26 22
805E:3D 39 35   76          DFB  $3D,$39,$35,$31,$2D,$29,$25,$21
8061:31 2D 29
8064:25 21
8066:3D 39 35   77          DFB  $3D,$39,$35,$31,$2D,$29,$25,$21
8069:31 2D 29
806C:25 21
806E:3C 38 34   78          DFB  $3C,$38,$34,$30,$2C,$28,$24,$20
8071:30 2C 28
8074:24 20
8076:3C 38 34   79          DFB  $3C,$38,$34,$30,$2C,$28,$24,$20
8079:30 2C 28
807C:24 20
807E:3F 3B 37   80          DFB  $3F,$3B,$37,$33,$2F,$2B,$27,$23
8081:33 2F 2B
8084:27 23
8086:3F 3B 37   81          DFB  $3F,$3B,$37,$33,$2F,$2B,$27,$23
8089:33 2F 2B
808C:27 23
```

Example 6-2 Cont.

```
808E:3E 3A 36    82          DFB   $3E,$3A,$36,$32,$2E,$2A,$26,$22
8091:32 2E 2A
8094:26 22
8096:3E 3A 36    83          DFB   $3E,$3A,$36,$32,$2E,$2A,$26,$22
8099:32 2E 2A
809C:26 22
809E:3D 39 35    84          DFB   $3D,$39,$35,$31,$2D,$29,$25,$21
80A1:31 2D 29
80A4:25 21
80A6:3D 39 35    85          DFB   $3D,$39,$35,$31,$2D,$29,$25,$21
80A9:31 2D 29
80AC:25 21
80AE:3C 38 34    86          DFB   $3C,$38,$34,$30,$2C,$28,$24,$20
80B1:30 2C 28
80B4:24 20
80B6:3C 38 34    87          DFB   $3C,$38,$34,$30,$2C,$28,$24,$20
80B9:30 2C 28
80BC:24 20
80BE:3F 3B 37    88          DFB   $3F,$3B,$37,$33,$2F,$2B,$27,$23
80C1:33 2F 2B
80C4:27 23
80C6:3F 3B 37    89          DFB   $3F,$3B,$37,$33,$2F,$2B,$27,$23
80C9:33 2F 2B
80CC:27 23
80CE:3E 3A 36    90          DFB   $3E,$3A,$36,$32,$2E,$2A,$26,$22
80D1:32 2E 2A
80D4:26 22
80D6:3E 3A 36    91          DFB   $3E,$3A,$36,$32,$2E,$2A,$26,$22
80D9:32 2E 2A
80DC:26 22
80DE:3D 39 35    92          DFB   $3D,$39,$35,$31,$2D,$29,$25,$21
80E1:31 2D 29
80E4:25 21
80E6:3D 39 35    93          DFB   $3D,$39,$35,$31,$2D,$29,$25,$21
80E9:31 2D 29
80EC:25 21
80EE:3C 38 34    94          DFB   $3C,$38,$34,$30,$2C,$28,$24,$20
80F1:30 2C 28
80F4:24 20
80F6:3C 38 34    95          DFB   $3C,$38,$34,$30,$2C,$28,$24,$20
80F9:30 2C 28
80FC:24 20
80FE:            96 *
80FE:            97 * HIRES LINE ADDRESSES - LOW
80FE:            98 *
80FE:D0 D0 D0    99 LOLINE  DFB   $D0,$D0,$D0,$D0,$D0,$D0,$D0,$D0
8101:D0 D0 D0
8104:D0 D0
8106:50 50 50    100         DFB   $50,$50,$50,$50,$50,$50,$50,$50
8109:50 50 50
810C:50 50
810E:D0 D0 D0    101         DFB   $D0,$D0,$D0,$D0,$D0,$D0,$D0,$D0
```

Example 6-2 Cont.

```
8111:D0 D0 D0
8114:D0 D0
8116:50 50 50    102    DFB    $50,$50,$50,$50,$50,$50,$50,$50
8119:50 50 50
811C:50 50
811E:D0 D0 D0    103    DFB    $D0,$D0,$D0,$D0,$D0,$D0,$D0,$D0
8121:D0 D0 D0
8124:D0 D0
8126:50 50 50    104    DFB    $50,$50,$50,$50,$50,$50,$50,$50
8129:50 50 50
812C:50 50
812E:D0 D0 D0    105    DFB    $D0,$D0,$D0,$D0,$D0,$D0,$D0,$D0
8131:D0 D0 D0
8134:D0 D0
8136:50 50 50    106    DFB    $50,$50,$50,$50,$50,$50,$50,$50
8139:50 50 50
813C:50 50
813E:A8 A8 A8    107    DFB    $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
8141:A8 A8 A8
8144:A8 A8
8146:28 28 28    108    DFB    $28,$28,$28,$28,$28,$28,$28,$28
8149:28 28 28
814C:28 28
814E:A8 A8 A8    109    DFB    $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
8151:A8 A8 A8
8154:A8 A8
8156:28 28 28    110    DFB    $28,$28,$28,$28,$28,$28,$28,$28
8159:28 28 28
815C:28 28
815E:A8 A8 A8    111    DFB    $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
8161:A8 A8 A8
8164:A8 A8
8166:28 28 28    112    DFB    $28,$28,$28,$28,$28,$28,$28,$28
8169:28 28 28
816C:28 28
816E:A8 A8 A8    113    DFB    $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
8171:A8 A8 A8
8174:A8 A8
8176:28 28 28    114    DFB    $28,$28,$28,$28,$28,$28,$28,$28
8179:28 28 28
817C:28 28
817E:80 80 80    115    DFB    $80,$80,$80,$80,$80,$80,$80,$80
8181:80 80 80
8184:80 80
8186:00 00 00    116    DFB    $00,$00,$00,$00,$00,$00,$00,$00
8189:00 00 00
818C:00 00
818E:80 80 80    117    DFB    $80,$80,$80,$80,$80,$80,$80,$80
8191:80 80 80
8194:80 80
8196:00 00 00    118    DFB    $00,$00,$00,$00,$00,$00,$00,$00
8199:00 00 00
```

Example 6-2 Cont.

```
819C:00 00
819E:80 80 80    119          DFB    $80,$80,$80,$80,$80,$80,$80,$80
81A1:80 80 80
81A4:80 80
81A6:00 00 00    120          DFB    $00,$00,$00,$00,$00,$00,$00,$00
81A9:00 00 00
81AC:00 00
81AE:80 80 80    121          DFB    $80,$80,$80,$80,$80,$80,$80,$80
81B1:80 80 80
81B4:80 80
81B6:00 00 00    122          DFB    $00,$00,$00,$00,$00,$00,$00,$00
81B9:00 00 00
81BC:00 00
81BE:            123 *
81BE:            124 * DIVISION TABLE FOR 2 MOD 7
81BE:            125 *
81BE:02 02 02    126 DIV7     DFB    2,2,2,2,2,2,2
81C1:02 02 02
81C4:02
81C5:03 03 03    127          DFB    3,3,3,3,3,3,3
81C8:03 03 03
81CB:03
81CC:04 04 04    128          DFB    4,4,4,4,4,4,4
81CF:04 04 04
81D2:04
81D3:05 05 05    129          DFB    5,5,5,5,5,5,5
81D6:05 05 05
81D9:05
81DA:06 06 06    130          DFB    6,6,6,6,6,6,6
81DD:06 06 06
81E0:06
81E1:07 07 07    131          DFB    7,7,7,7,7,7,7
81E4:07 07 07
81E7:07
81E8:08 08 08    132          DFB    8,8,8,8,8,8,8
81EB:08 08 08
81EE:08
81EF:09 09 09    133          DFB    9,9,9,9,9,9,9
81F2:09 09 09
81F5:09
81F6:0A 0A 0A    134          DFB    10,10,10,10,10,10,10
81F9:0A 0A 0A
81FC:0A
81FD:0B 0B 0B    135          DFB    11,11,11,11,11,11,11
8200:0B 0B 0B
8203:0B
8204:0C 0C 0C    136          DFB    12,12,12,12,12,12,12
8207:0C 0C 0C
820A:0C
820B:0D 0D 0D    137          DFB    13,13,13,13,13,13,13
820E:0D 0D 0D
8211:0D
```

Example 6-2 Cont.

```
8212:0E 0E 0E   138        DFB   14,14,14,14,14,14,14
8215:0E 0E 0E
8218:0E
8219:0F 0F 0F   139        DFB   15,15,15,15,15,15,15
821C:0F 0F 0F
821F:0F
8220:10 10 10   140        DFB   16,16,16,16,16,16,16
8223:10 10 10
8226:10
8227:11 11 11   141        DFB   17,17,17,17,17,17,17
822A:11 11 11
822D:11
822E:12 12 12   142        DFB   18,18,18,18,18,18,18
8231:12 12 12
8234:12
8235:13 13 13   143        DFB   19,19,19,19,19,19,19
8238:13 13 13
823B:13
823C:14 14 14   144        DFB   20,20,20,20,20,20,20
823F:14 14 14
8242:14
8243:15 15 15   145        DFB   21,21,21,21,21,21,21
8246:15 15 15
8249:15
824A:16 16 16   146        DFB   22,22,22,22,22,22,22
824D:16 16 16
8250:16
8251:17 17 17   147        DFB   23,23,23,23,23,23,23
8254:17 17 17
8257:17
8258:18 18 18   148        DFB   24,24,24,24,24,24,24
825B:18 18 18
825E:18
825F:19 19 19   149        DFB   25,25,25,25,25,25,25
8262:19 19 19
8265:19
8266:1A 1A 1A   150        DFB   26,26,26,26,26,26,26
8269:1A 1A 1A
826C:1A
826D:1B 1B 1B   151        DFB   27,27,27,27,27,27,27
8270:1B 1B 1B
8273:1B
8274:1C 1C 1C   152        DFB   28,28,28,28,28,28,28
8277:1C 1C 1C
827A:1C
827B:1D 1D 1D   153        DFB   29,29,29,29,29,29,29
827E:1D 1D 1D
8281:1D
8282:1E 1E 1E   154        DFB   30,30,30,30,30,30,30
8285:1E 1E 1E
8288:1E
8289:1F 1F 1F   155        DFB   31,31,31,31,31,31,31
```

Example 6-2 Cont.

```
828C:1F 1F 1F
828F:1F
8290:20 20 20    156          DFB    32,32,32,32,32,32,32
8293:20 20 20
8296:20
8297:21 21 21    157          DFB    33,33,33,33,33,33,33
829A:21 21 21
829D:21
829E:22 22 22    158          DFB    34,34,34,34,34,34,34
82A1:22 22 22
82A4:22
82A5:23 23 23    159          DFB    35,35,35,35,35,35,35
82A8:23 23 23
82AB:23
82AC:24 24 24    160          DFB    36,36,36,36,36,36,36
82A'5:24 24 24
82B2:24
82B3:25 25 25    161          DFB    37,37,37,37,37,37,37
82B6:25 25 25
82B9:25
82BA:26 26 26    162          DFB    38,38,38,38,38,38,38
82BD:26 26 26
82C0:26
82C1:            163 *
82C1:            164 * BIT MASKS FOR B/W PIXELS
82C1:            165 *
82C1:01 02 04    166 BWPIX    DFB    1,2,4,8,16,32,64
82C4:08 10 20
82C7:40
82C8:01 02 04    167          DFB    1,2,4,8,16,32,64
82CB:08 10 20
82CE:40
82CF:01 02 04    168          DFB    1,2,4,8,16,32,64
82D2:08 10 20
82D5:40
82D6:01 02 04    169          DFB    1,2,4,8,16,32,64
82D9:08 10 20
82DC:40
82DD:01 02 04    170          DFB    1,2,4,8,16,32,64
82E0:08 10 20
82E3:40
82E4:01 02 04    171          DFB    1,2,4,8,16,32,64
82E7:08 10 20
82EA:40
82EB:01 02 04    172          DFB    1,2,4,8,16,32,64
82EE:08 10 20
82F1:40
82F2:01 02 04    173          DFB    1,2,4,8,16,32,64
82F5:08 10 20
82F8:40
82F9:01 02 04    174          DFB    1,2,4,8,16,32,64
82FC:08 10 20
```

Example 6-2 Cont.

```
82FF:40
8300:01 02 04    175         DFB  1,2,4,8,16,32,64
8303:08 10 20
8306:40
8307:01 02 04    176         DFB  1,2,4,8,16,32,64
830A:08 10 20
830D:40
830E:01 02 04    177         DFB  1,2,4,8,16,32,64
8311:08 10 20
8314:40
8315:01 02 04    178         DFB  1,2,4,8,16,32,64
8318:08 10 20
831B:40
831C:01 02 04    179         DFB  1,2,4,8,16,32,64
831F:08 10 20
8322:40
8323:01 02 04    180         DFB  1,2,4,8,16,32,64
8326:08 10 20
8329:40
832A:01 02 04    181         DFB  1,2,4,8,16,32,64
832D:08 10 20
8330:40
8331:01 02 04    182         DFB  1,2,4,8,16,32,64
8334:08 10 20
8337:40
8338:01 02 04    183         DFB  1,2,4,8,16,32,64
833B:08 10 20
833E:40
833F:01 02 04    184         DFB  1,2,4,8,16,32,64
8342:08 10 20
8345:40
8346:01 02 04    185         DFB  1,2,4,8,16,32,64
8349:08 10 20
834C:40
834D:01 02 04    186         DFB  1,2,4,8,16,32,64
8350:08 10 20
8353:40
8354:01 02 04    187         DFB  1,2,4,8,16,32,64
8357:08 10 20
835A:40
835B:01 02 04    188         DFB  1,2,4,8,16,32,64
835E:08 10 20
8361:40
8362:01 02 04    189         DFB  1,2,4,8,16,32,64
8365:08 10 20
8368:40
8369:01 02 04    190         DFB  1,2,4,8,16,32,64
836C:08 10 20
836F:40
8370:01 02 04    191         DFB  1,2,4,8,16,32,64
8373:08 10 20
8376:40
8377:01 02 04    192         DFB  1,2,4,8,16,32,64
```

Example 6-2 Cont.

```
837A:08 10 20
837D:40
837E:01 02 04    193         DFB    1,2,4,8,16,32,64
8381:08 10 20
8384:40
8385:01 02 04    194         DFB    1,2,4,8,16,32,64
8388:08 10 20
838B:40
838C:01 02 04    195         DFB    1,2,4,8,16,32,64
838F:08 10 20
8392:40
8393:01 02 04    196         DFB    1,2,4,8,16,32,64
8396:08 10 20
8399:40
839A:01 02 04    197         DFB    1,2,4,8,16,32,64
839D:08 10 20
83A0:40
83A1:01 02 04    198         DFB    1,2,4,8,16,32,64
83A4:08 10 20
83A7:40
83A8:01 02 04    199         DFB    1,2,4,8,16,32,64
83AB:08 10 20
83AE:40
83AF:01 02 04    200         DFB    1,2,4,8,16,32,64
83B2:08 10 20
83B5:40
83B6:01 02 04    201         DFB    1,2,4,8,16,32,64
83B9:08 10 20
83BC:40
83BD:01 02 04    202         DFB    1,2,4,8,16,32,64
83C0:08 10 20
83C3:40
83C4:00          203         BRK

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

locations on the screen at the time they are defined. Shapes, on the other hand, are defined and kept in files to be loaded and drawn on the screen wherever the drawing program commands. Shapes don't have positions.

In order to use a shape table, you must know the number of shapes in the table. Also you need to know the largest size — in X and in Y — it contains, and the start position of each shape. For example, a character set may have 64 shapes, be 5 by 7 each in size, with the start of each shape at the lower left of the 5 by 7 area. With this information, you can draw shapes on the screen without the danger of any shape spreading itself off-screen and wrapping around the screen. If a shape table is unknown to you but you want to use it, then get the number of shapes by:

$$NS = PEEK(FN\ AD(232))$$

Example 6-3.

```
SOURCE FILE: EXAMPLE 6.3
0000:            1 *******************************
0000:            2 * EXAMPLE 6.3                 *
0000:            3 *                             *
0000:            4 *  M I D R E S    P I X E L S *
0000:            5 *                             *
0000:            6 * CALL SEQUENCE:              *
0000:            7 *   A-REG <--- COLOR 0...15   *
0000:            8 *   X-REG <--- X-COORD        *
0000:            9 *   Y-REG <--- Y-COORD        *
0000:           10 * ORIGIN IS LOWER LEFT        *
0000:           11 *******************************
0000:           12 *
0000:           13 *
0000:           14 *    E Q U A T E S
0000:           15 *
0030:           16 PMASK   EQU  $30       POSITION MASK
0031:           17 COLMX   EQU  $31       COLOR MASK INDEX
0045:           18 AREG    EQU  $45       PASSED PIXEL VALUE
0046:           19 XREG    EQU  $46       PASSED X-COORD.
0047:           20 YREG    EQU  $47       PASSED Y-COORD.
0050:           21 SCREEN  EQU  $50       POINTER TO HIRES1
0000:           22 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 6.3.OBJ0
8000:           23          ORG  $8000
8000:           24 *
8000:           25 *    R O U T I N E S
8000:           26 *
8000:08         27 MPIX     PHP
8001:85 45      28          STA  AREG
8003:86 46      29          STX  XREG
8005:84 47      30          STY  YREG      KEEP REGISTERS
8007:C0 C0      31          CPY  #$C0
8009:B0 6A      32          BCS  MPIX1      CLIP Y-COORD.
800B:           33 *
800B:           34 * PLOT IN UPPER LEFT BYTE.
800B:           35 *
800B:98         36          TYA
800C:09 01      37          ORA  #1        FORCE UPPER LINE.
800E:A8         38          TAY
800F:B9 51 81   39          LDA  LOLINE,Y
8012:85 50      40          STA  SCREEN
8014:B9 91 80   41          LDA  HILINE,Y
8017:85 51      42          STA  SCREEN+1
8019:8A         43          TXA
801A:29 FE      44          AND  #$FE      FORCE LEFTMOST BYTE.
801C:AA         45          TAX
801D:BD 11 82   46          LDA  DIV7,X
8020:A8         47          TAY
8021:BD 14 83   48          LDA  LEFT,X
8024:85 30      49          STA  PMASK
```

Example 6-3 Cont.

```
8026:A5 45      50          LDA     AREG
8028:0A         51          ASL     A
8029:0A         52          ASL     A           LOOKUP PIXEL IN COLOR
TABLE
802A:85 31      53          STA     COLMX
802C:20 7D 80   54          JSR     CPLOT       PLOTS UPPER LEFT.
802F:           55 *
802F:           56 * PLOT IN UPPER RIGHT BYTE.
802F:           57 *
802F:E6 31      58          INC     COLMX       NEXT COLOR MASK
8031:A5 46      59          LDA     XREG
8033:09 01      60          ORA     #1          FORCE RIGHTMOST BYTE.


8035:AA         61          TAX
8036:BD 11 82   62          LDA     DIV7,X
8039:A8         63          TAY
803A:BD 14 84   64          LDA     RIGHT,X
803D:85 30      65          STA     PMASK
803F:20 7D 80   66          JSR     CPLOT       PLOTS UPPER RIGHT.
8042:           67 *
8042:           68 * PLOT IN LOWER LEFT BYTE.
8042:           69 *
8042:E6 31      70          INC     COLMX       NEXT COLOR MASK
8044:A5 47      71          LDA     YREG
8046:29 FE      72          AND     #$FE        FORCE LOWER LINE.
8048:A8         73          TAY
8049:B9 51 81   74          LDA     LOLINE,Y
804C:85 50      75          STA     SCREEN
804E:B9 91 80   76          LDA     HILINE,Y
8051:85 51      77          STA     SCREEN+1
8053:A5 46      78          LDA     XREG
8055:29 FE      79          AND     #$FE        FORCE LEFTMOST BYTE.
8057:AA         80          TAX
8058:BD 11 82   81          LDA     DIV7,X
805B:A8         82          TAY
805C:BD 14 83   83          LDA     LEFT,X
805F:85 30      84          STA     PMASK
8061:20 7D 80   85          JSR     CPLOT
8064:           86 *
8064:           87 * PLOT IN LOWER RIGHT BYTE
8064:           88 *
8064:A5 46      89          LDA     XREG
8066:09 01      90          ORA     #$01        FORCE RIGHTMOST BYTE
8068:AA         91          TAX
8069:BD 11 82   92          LDA     DIV7,X
806C:A8         93          TAY
806D:BD 14 84   94          LDA     RIGHT,X
8070:85 30      95          STA     PMASK
8072:20 7D 80   96          JSR     CPLOT
8075:           97 *
```

Example 6-3 Cont.

```
8075:            98 * ALL FOUR DONE.  RETURN.
8075:            99 *
8075:A6 46      100 MPIX1   LDX   XREG
8077:A4 47      101         LDY   YREG
8079:A5 45      102         LDA   AREG
807B:28         103         PLP
807C:60         104         RTS
807D:           105 *
807D:           106 * COLOR PLOT OF ONE BYTE.
807D:           107 *
807D:A6 31      108 CPLOT   LDX   COLMX
807F:A5 30      109         LDA   PMASK
8081:49 FF      110         EOR   #$FF        COMPLIMENT P
8083:31 50      111         AND   (SCREEN),Y
8085:91 50      112         STA   (SCREEN),Y  TEMPORARILY
8087:A5 30      113         LDA   PMASK
8089:3D 14 85   114         AND   COLOR,X
808C:11 50      115         ORA   (SCREEN),Y
808E:91 50      116         STA   (SCREEN),Y
8090:60         117         RTS
8091:           118 *
8091:           119 *
8091:           120 *
8091:           121 *    L I T E R A L S
8091:           122 *
8091:           123 *
8091:           124 * HIRES1 LINE ADDRESSES - HIGH
8091:           125 *
8091:3F 3B 37   126 HILINE  DFB   $3F,$3B,$37,$33,$2F,$2B,$27,$23
8094:33 2F 2B
8097:27 23
8099:3F 3B 37   127         DFB   $3F,$3B,$37,$33,$2F,$2B,$27,$23
809C:33 2F 2B
809F:27 23
80A1:3E 3A 36   128         DFB   $3E,$3A,$36,$32,$2E,$2A,$26,$22
80A4:32 2E 2A
80A7:26 22
80A9:3E 3A 36   129         DFB   $3E,$3A,$36,$32,$2E,$2A,$26,$22
80AC:32 2E 2A
80AF:26 22
80B1:3D 39 35   130         DFB   $3D,$39,$35,$31,$2D,$29,$25,$21
80B4:31 2D 29
80B7:25 21
80B9:3D 39 35   131         DFB   $3D,$39,$35,$31,$2D,$29,$25,$21
80BC:31 2D 29
80BF:25 21
80C1:3C 38 34   132         DFB   $3C,$38,$34,$30,$2C,$28,$24,$20
80C4:30 2C 28
80C7:24 20
80C9:3C 38 34   133         DFB   $3C,$38,$34,$30,$2C,$28,$24,$20
```

Example 6-3 Cont.

```
80CC:30 2C 28
80CF:24 20
80D1:3F 3B 37  134        DFB    $3F,$3B,$37,$33,$2F,$2B,$27,$23
80D4:33 2F 2B
80D7:27 23
80D9:3F 3B 37  135        DFB    $3F,$3B,$37,$33,$2F,$2B,$27,$23
80DC:33 2F 2B
80DF:27 23
80E1:3E 3A 36  136        DFB    $3E,$3A,$36,$32,$2E,$2A,$26,$22
80E4:32 2E 2A
80E7:26 22
80E9:3E 3A 36  137        DFB    $3E,$3A,$36,$32,$2E,$2A,$26,$22
80EC:32 2E 2A
80EF:26 22
80F1:3D 39 35  138        DFB    $3D,$39,$35,$31,$2D,$29,$25,$21
80F4:31 2D 29
80F7:25 21
80F9:3D 39 35  139        DFB    $3D,$39,$35,$31,$2D,$29,$25,$21
80FC:31 2D 29
80FF:25 21
8101:3C 38 34  140        DFB    $3C,$38,$34,$30,$2C,$28,$24,$20
8104:30 2C 28
8107:24 20
8109:3C 38 34  141        DFB    $3C,$38,$34,$30,$2C,$28,$24,$20
810C:30 2C 28
810F:24 20
8111:3F 3B 37  142        DFB    $3F,$3B,$37,$33,$2F,$2B,$27,$23
8114:33 2F 2B
8117:27 23
8119:3F 3B 37  143        DFB    $3F,$3B,$37,$33,$2F,$2B,$27,$23
811C:33 2F 2B
811F:27 23
8121:3E 3A 36  144        DFB    $3E,$3A,$36,$32,$2E,$2A,$26,$22
8124:32 2E 2A
8127:26 22
8129:3E 3A 36  145        DFB    $3E,$3A,$36,$32,$2E,$2A,$26,$22
812C:32 2E 2A
812F:26 22
8131:3D 39 35  146        DFB    $3D,$39,$35,$31,$2D,$29,$25,$21
8134:31 2D 29
8137:25 21
8139:3D 39 35  147        DFB    $3D,$39,$35,$31,$2D,$29,$25,$21
813C:31 2D 29
813F:25 21
8141:3C 38 34  148        DFB    $3C,$38,$34,$30,$2C,$28,$24,$20
8144:30 2C 28
8147:24 20
8149:3C 38 34  149        DFB    $3C,$38,$34,$30,$2C,$28,$24,$20
814C:30 2C 28
814F:24 20
8151:          150 *
```

Example 6-3 Cont.

```
8151:                  151 * HIRES LINE ADDRESSES - LOW
8151:                  152 *
8151:DO DO DO          153 LOLINE   DFB     $DO,$DO,$DO,$DO,$DO,$DO,$DO,$DO
8154:DO DO DO
8157:DO DO
8159:50 50 50          154          DFB     $50,$50,$50,$50,$50,$50,$50,$50
815C:50 50 50
815F:50 50
8161:DO DO DO          155          DFB     $DO,$DO,$DO,$DO,$DO,$DO,$DO,$DO
8164:DO DO DO
8167:DO DO
8169:50 50 50          156          DFB     $50,$50,$50,$50,$50,$50,$50,$50
816C:50 50 50
816F:50 50
8171:DO DO DO          157          DFB     $DO,$DO,$DO,$DO,$DO,$DO,$DO,$DO
8174:DO DO DO
8177:DO DO
8179:50 50 50          158          DFB     $50,$50,$50,$50,$50,$50,$50,$50
817C:50 50 50
817F:50 50
8181:DO DO DO          159          DFB     $DO,$DO,$DO,$DO,$DO,$DO,$DO,$DO
8184:DO DO DO
8187:DO DO
8189:50 50 50          160          DFB     $50,$50,$50,$50,$50,$50,$50,$50
818C:50 50 50
818F:50 50
8191:A8 A8 A8          161          DFB     $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
8194:A8 A8 A8
8197:A8 A8
8199:28 28 28          162          DFB     $28,$28,$28,$28,$28,$28,$28,$28
819C:28 28 28
819F:28 28
81A1:A8 A8 A8          163          DFB     $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
81A4:A8 A8 A8
81A7:A8 A8
81A9:28 28 28          164          DFB     $28,$28,$28,$28,$28,$28,$28,$28
81AC:28 28 28
81AF:28 28
81B1:A8 A8 A8          165          DFB     $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
81B4:A8 A8 A8
81B7:A8 A8
81B9:28 28 28          166          DFB     $28,$28,$28,$28,$28,$28,$28,$28
81BC:28 28 28
81BF:28 28
81C1:A8 A8 A8          167          DFB     $A8,$A8,$A8,$A8,$A8,$A8,$A8,$A8
81C4:A8 A8 A8
81C7:A8 A8
81C9:28 28 28          168          DFB     $28,$28,$28,$28,$28,$28,$28,$28
81CC:28 28 28
81CF:28 28
81D1:80 80 80          169          DFB     $80,$80,$80,$80,$80,$80,$80,$80
```

Example 6-3 Cont.

```
81D4:80 80 80
81D7:80 80
81D9:00 00 00    170          DFB    $00,$00,$00,$00,$00,$00,$00,$00
81DC:00 00 00
81DF:00 00
81E1:80 80 80    171          DFB    $80,$80,$80,$80,$80,$80,$80,$80
81E4:80 80 80
81E7:80 80
81E9:00 00 00    172          DFB    $00,$00,$00,$00,$00,$00,$00,$00
81EC:00 00 00
81EF:00 00
81F1:80 80 80    173          DFB    $80,$80,$80,$80,$80,$80,$80,$80
81F4:80 80 80
81F7:80 80
81F9:00 00 00    174          DFB    $00,$00,$00,$00,$00,$00,$00,$00
81FC:00 00 00
81FF:00 00
8201:80 80 80    175          DFB    $80,$80,$80,$80,$80,$80,$80,$80
8204:80 80 80
8207:80 80
8209:00 00 00    176          DFB    $00,$00,$00,$00,$00,$00,$00,$00
820C:00 00 00
820F:00 00
8211:            177 *
8211:            178 * DIVISION TABLE FOR 2 MOD 7
8211:            179 *
8211:02 02 02    180 DIV7     DFB    2,2,2,2,2,2,2
8214:02 02 02
8217:02
8218:03 03 03    181          DFB    3,3,3,3,3,3,3
821B:03 03 03
821E:03
821F:04 04 04    182          DFB    4,4,4,4,4,4,4
8222:04 04 04
8225:04
8226:05 05 05    183          DFB    5,5,5,5,5,5,5
8229:05 05 05
822C:05
822D:06 06 06    184          DFB    6,6,6,6,6,6,6
8230:06 06 06
8233:06
8234:07 07 07    185          DFB    7,7,7,7,7,7,7
8237:07 07 07
823A:07
823B:08 08 08    186          DFB    8,8,8,8,8,8,8
823E:08 08 08
8241:08
8242:09 09 09    187          DFB    9,9,9,9,9,9,9
8245:09 09 09
8248:09
8249:0A 0A 0A    188          DFB    10,10,10,10,10,10,10
```

Example 6-3 Cont.

```
824C:0A 0A 0A
824F:0A
8250:0B 0B 0B    189        DFB   11,11,11,11,11,11,11
8253:0B 0B 0B
8256:0B
8257:0C 0C 0C    190        DFB   12,12,12,12,12,12,12
825A:0C 0C 0C
825D:0C
825E:0D 0D 0D    191        DFB   13,13,13,13,13,13,13
8261:0D 0D 0D
8264:0D
8265:0E 0E 0E    192        DFB   14,14,14,14,14,14,14
8268:0E 0E 0E
826B:0E
826C:0F 0F 0F    193        DFB   15,15,15,15,15,15,15
826F:0F 0F 0F '
8272:0F
8273:10 10 10    194        DFB   16,16,16,16,16,16,16
8276:10 10 10
8279:10
827A:11 11 11    195        DFB   17,17,17,17,17,17,17
827D:11 11 11
8280:11
8281:12 12 12    196        DFB   18,18,18,18,18,18,18
8284:12 12 12
8287:12
8288:13 13 13    197        DFB   19,19,19,19,19,19,19
828B:13 13 13
828E:13
828F:14 14 14    198        DFB   20,20,20,20,20,20,20
8292:14 14 14
8295:14
8296:15 15 15    199        DFB   21,21,21,21,21,21,21
8299:15 15 15
829C:15
829D:16 16 16    200        DFB   22,22,22,22,22,22,22
82A0:16 16 16
82A3:16
82A4:17 17 17    201        DFB   23,23,23,23,23,23,23
82A7:17 17 17
82AA:17
82AB:18 18 18    202        DFB   24,24,24,24,24,24,24
82AE:18 18 18
82B1:18
82B2:19 19 19    203        DFB   25,25,25,25,25,25,25
82B5:19 19 19
82B8:19
82B9:1A 1A 1A    204        DFB   26,26,26,26,26,26,26
82BC:1A 1A 1A
82BF:1A
82C0:1B 1B 1B    205        DFB   27,27,27,27,27,27,27
```

Example 6-3 Cont.

```
82C3:1B 1B 1B
82C6:1B
82C7:1C 1C 1C    206        DFB    28,28,28,28,28,28,28
82CA:1C 1C 1C
82CD:1C
82CE:1D 1D 1D    207        DFB    29,29,29,29,29,29,29
82D1:1D 1D 1D
82D4:1D
82D5:1E 1E 1E    208        DFB    30,30,30,30,30,30,30
82D8:1E 1E 1E
82DB:1E
82DC:1F 1F 1F    209        DFB    31,31,31,31,31,31,31
82DF:1F 1F 1F
82E2:1F
82E3:20 20 20    210        DFB    32,32,32,32,32,32,32
82E6:20 20 20
82E9:20
82EA:21 21 21    211        DFB    33,33,33,33,33,33,33
82ED:21 21 21
82F0:21
82F1:22 22 22    212        DFB    34,34,34,34,34,34,34
82F4:22 22 22
82F7:22
82F8:23 23 23    213        DFB    35,35,35,35,35,35,35
82FB:23 23 23
82FE:23
82FF:24 24 24    214        DFB    36,36,36,36,36,36,36
8302:24 24 24
8305:24
8306:25 25 25    215        DFB    37,37,37,37,37,37,37
8309:25 25 25
830C:25
830D:26 26 26    216        DFB    38,38,38,38,38,38,38
8310:26 26 26
8313:26
8314:            217 *
8314:            218 * POSITION MASKS FOR MIDRES BITS
8314:            219 *
8314:83 83 8C    220 LEFT   DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
8317:8C B0 B0
831A:C0
831B:C0 80 80    221        DFB    $C0,$80,$80,$80,$80,$80,$80
831E:80 80 80
8321:80
8322:83 83 8C    222        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
8325:8C B0 B0
8328:C0
8329:C0 80 80    223        DFB    $C0,$80,$80,$80,$80,$80,$80
832C:80 80 80
832F:80
8330:83 83 8C    224        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
```

Example 6-3 Cont.

```
8333:8C B0 B0
8336:C0
8337:C0 80 80    225        DFB    $C0,$80,$80,$80,$80,$80,$80
833A:80 80 80
833D:80
833E:83 83 8C    226        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
8341:8C B0 B0
8344:C0
8345:C0 80 80    227        DFB    $C0,$80,$80,$80,$80,$80,$80
8348:80 80 80
834B:80
834C:83 83 8C    228        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
834F:8C B0 B0
8352:C0
8353:C0 80 80    229        DFB    $C0,$80,$80,$80,$80,$80,$80
8356:80 80 80
8359:80
835A:83 83 8C    230        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
835D:8C B0 B0
8360:C0
8361:C0 80 80    231        DFB    $C0,$80,$80,$80,$80,$80,$80
8364:80 80 80
8367:80
8368:83 83 8C    232        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
836B:8C B0 B0
836E:C0
836F:C0 80 80    233        DFB    $C0,$80,$80,$80,$80,$80,$80
8372:80 80 80
8375:80
8376:83 83 8C    234        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
8379:8C B0 B0
837C:C0
837D:C0 80 80    235        DFB    $C0,$80,$80,$80,$80,$80,$80
8380:80 80 80
8383:80
8384:83 83 8C    236        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
8387:8C B0 B0
838A:C0
838B:C0 80 80    237        DFB    $C0,$80,$80,$80,$80,$80,$80
838E:80 80 80
8391:80
8392:83 83 8C    238        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
8395:8C B0 B0
8398:C0
8399:C0 80 80    239        DFB    $C0,$80,$80,$80,$80,$80,$80
839C:80 80 80
839F:80
83A0:83 83 8C    240        DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
83A3:8C B0 B0
83A6:C0
```

Example 6-3 Cont.

```
83A7:C0 80 80    241         DFB    $C0,$80,$80,$80,$80,$80,$80
83AA:80 80 80
83AD:80
83AE:83 83 8C    242         DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
83B1:8C B0 B0
83B4:C0
83B5:C0 80 80    243         DFB    $C0,$80,$80,$80,$80,$80,$80
83B8:80 80 80
83BB:80
83BC:83 83 8C    244         DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
83BF:8C B0 B0
83C2:C0
83C3:C0 80 80    245         DFB    $C0,$80,$80,$80,$80,$80,$80
83C6:80 80 80
83C9:80
83CA:83 83 8C    246         DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
83CD:8C B0 B0
83D0:C0
83D1:C0 80 80    247         DFB    $C0,$80,$80,$80,$80,$80,$80
83D4:80 80 80
83D7:80
83D8:83 83 8C    248         DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
83DB:8C B0 B0
83DE:C0
83DF:C0 80 80    249         DFB    $C0,$80,$80,$80,$80,$80,$80
83E2:80 80 80
83E5:80
83E6:83 83 8C    250         DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
83E9:8C B0 B0
83EC:C0
83ED:C0 80 80    251         DFB    $C0,$80,$80,$80,$80,$80,$80
83F0:80 80 80
83F3:80
83F4:83 83 8C    252         DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
83F7:8C B0 B0
83FA:C0
83FB:C0 80 80    253         DFB    $C0,$80,$80,$80,$80,$80,$80
83FE:80 80 80
8401:80
8402:83 83 8C    254         DFB    $83,$83,$8C,$8C,$B0,$B0,$C0
8405:8C B0 B0
8408:C0
8409:C0 80 80    255         DFB    $C0,$80,$80,$80,$80,$80,$80
840C:80 80 80
840F:80
8410:83 83 8C    256         DFB    $83,$83,$8C,$8C
8413:8C
8414:            257 *
8414:80 80 80    258 RIGHT   DFB    $80,$80,$80,$80,$80,$80,$81
8417:80 80 80
```

Example 6-3 Cont.

```
841A:81
841B:81 86 86    259        DFB    $81,$86,$86,$98,$98,$E0,$E0
841E:98 98 E0
8421:E0
8422:80 80 80    260        DFB    $80,$80,$80,$80,$80,$80,$81
8425:80 80 80
8428:81
8429:81 86 86    261        DFB    $81,$86,$86,$98,$98,$E0,$E0
842C:98 98 E0
842F:E0
8430:80 80 80    262        DFB    $80,$80,$80,$80,$80,$80,$81
8433:80 80 80
8436:81
8437:81 86 86    263        DFB    $81,$86,$86,$98,$98,$E0,$E0
843A:98 98 E0
843D:E0
843E:80 80 80    264        DFB    $80,$80,$80,$80,$80,$80,$81
8441:80 80 80
8444:81
8445:81 86 86    265        DFB    $81,$86,$86,$98,$98,$E0,$E0
8448:98 98 E0
844B:E0
844C:80 80 80    266        DFB    $80,$80,$80,$80,$80,$80,$81
844F:80 80 80
8452:81
8453:81 86 86    267        DFB    $81,$86,$86,$98,$98,$E0,$E0
8456:98 98 E0
8459:E0
845A:80 80 80    268        DFB    $80,$80,$80,$80,$80,$80,$81
845D:80 80 80
8460:81
8461:81 86 86    269        DFB    $81,$86,$86,$98,$98,$E0,$E0
8464:98 98 E0
8467:E0
8468:80 80 80    270        DFB    $80,$80,$80,$80,$80,$80,$81
846B:80 80 80
846E:81
846F:81 86 86    271        DFB    $81,$86,$86,$98,$98,$E0,$E0
8472:98 98 E0
8475:E0
8476:80 80 80    272        DFB    $80,$80,$80,$80,$80,$80,$81
8479:80 80 80
847C:81
847D:81 86 86    273        DFB    $81,$86,$86,$98,$98,$E0,$E0
8480:98 98 E0
8483:E0
8484:80 80 80    274        DFB    $80,$80,$80,$80,$80,$80,$81
8487:80 80 80
848A:81
848B:81 86 86    275        DFB    $81,$86,$86,$98,$98,$E0,$E0
848E:98 98 E0
8491:E0
```

Example 6-3 Cont.

```
8492:80 80 80   276        DFB   $80,$80,$80,$80,$80,$80,$81
8495:80 80 80
8498:81
8499:81 86 86   277        DFB   $81,$86,$86,$98,$98,$E0,$E0
849C:98 98 E0
849F:E0
84A0:80 80 80   278        DFB   $80,$80,$80,$80,$80,$80,$81
84A3:80 80 80
84A6:81
84A7:81 86 86   279        DFB   $81,$86,$86,$98,$98,$E0,$E0
84AA:98 98 E0
84AD:E0
84AE:80 80 80   280        DFB   $80,$80,$80,$80,$80,$80,$81
84B1:80 80 80
84B4:81
84B5:81 86 86   281        DFB   $81,$86,$86,$98,$98,$E0,$E0
84B8:98 98 E0
84BB:E0
84BC:80 80 80   282        DFB   $80,$80,$80,$80,$80,$80,$81
84BF:80 80 80
84C2:81
84C3:81 86 86   283        DFB   $81,$86,$86,$98,$98,$E0,$E0
84C6:98 98 E0
84C9:E0
84CA:80 80 80   284        DFB   $80,$80,$80,$80,$80,$80,$81
84CD:80 80 80
84D0:81
84D1:81 86 86   285        DFB   $81,$86,$86,$98,$98,$E0,$E0
84D4:98 98 E0
84D7:E0
84D8:80 80 80   286        DFB   $80,$80,$80,$80,$80,$80,$81
84DB:80 80 80
84DE:81
84DF:81 86 86   287        DFB   $81,$86,$86,$98,$98,$E0,$E0
84E2:98 98 E0
84E5:E0
84E6:80 80 80   288        DFB   $80,$80,$80,$80,$80,$80,$81
84E9:80 80 80
84EC:81
84ED:81 86 86   289        DFB   $81,$86,$86,$98,$98,$E0,$E0
84F0:98 98 E0
84F3:E0
84F4:80 80 80   290        DFB   $80,$80,$80,$80,$80,$80,$81
84F7:80 80 80
84FA:81
84FB:81 86 86   291        DFB   $81,$86,$86,$98,$98,$E0,$E0
84FE:98 98 E0
8501:E0
8502:80 80 80   292        DFB   $80,$80,$80,$80,$80,$80,$81
8505:80 80 80
8508:81
8509:81 86 86   293        DFB   $81,$86,$86,$98,$98,$E0,$E0
```

Example 6-3 Cont.

```
850C:98 98 E0
850F:E0
8510:80 80 80    294          DFB    $80,$80,$80,$80
8513:80
8514:            295 *
8514:            296 *   MASKS FOR MIDRES COLORS
8514:            297 *
8514:80 80       298 COLOR   DFB    $80,$80        0...BLACK
8516:00 00       299         DFB    $00,$00
8518:            300 *
8518:80 80       301         DFB    $80,$80        1...DK.VIOLET
851A:55 2A       302         DFB    $55,$2A
851C:            303 *
851C:D5 AA       304         DFB    $D5,$AA        2...DK.BLUE
851E:00 00       305         DFB    $00,$00
8520:            306 *
8520:D5 AA       307         DFB    $D5,$AA        3...TRUE BLUE
8522:55 2A       308         DFB    $55,$2A
8524:            309 *
8524:80 80       310         DFB    $80,$80        4...DK.GREEN
8526:2A 55       311         DFB    $2A,$55
8528:            312 *
8528:80 80       313         DFB    $80,$80        5...GREY-1
852A:7F 7F       314         DFB    $7F,$7F
852C:            315 *
852C:D5 AA       316         DFB    $D5,$AA        6...AQUA
852E:2A 55       317         DFB    $2A,$55
8530:            318 *
8530:D5 AA       319         DFB    $D5,$AA        7...LT.BLUE
8532:7F 7F       320         DFB    $7F,$7F
8534:            321 *
8534:AA D5       322         DFB    $AA,$D5        8...DK.ORANGE
8536:00 00       323         DFB    $00,$00
8538:            324 *
8538:AA D5       325         DFB    $AA,$D5        9...PINK
853A:55 2A       326         DFB    $55,$2A
853C:            327 *
853C:FF FF       328         DFB    $FF,$FF        10..GREY-2
853E:00 00       329         DFB    $00,$00
8540:            330 *
8540:FF FF       331         DFB    $FF,$FF        11..LT.VIOLET
8542:55 2A       332         DFB    $55,$2A
8544:            333 *
8544:AA D5       334         DFB    $AA,$D5        12..BROWN
8546:2A 55       335         DFB    $2A,$55
8548:            336 *
8548:AA D5       337         DFB    $AA,$D5        13..LT.ORANGE
854A:7F 7F       338         DFB    $7F,$7F
854C:            339 *
854C:FF FF       340         DFB    $FF,$FF        14..LT.GREEN
854E:2A 55       341         DFB    $2A,$55
8550:            342 *
8550:FF FF       343         DFB    $FF,$FF        15..WHITE
```

Example 6-3 Cont.

```
8552:7F 7F      344           DFB   $7F,$7F
8554:           345 *
8554:00         346           BRK
```

```
*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Then set the Applesoft shape parameters

<div align="center">

HCOLOR = 3 : SCALE = 1 : ROT = 0

</div>

before drawing them one by one at the center of the screen with

<div align="center">

DRAW 1 AT 140, 80

</div>

to see what you have. Remember, you must know these things in order to write a program that uses a shape table.

When you program to draw shapes, you use five Applesoft commands — HCOLOR, SCALE =, ROT =, DRAW, and XDRAW.

The DRAW and XDRAW commands put the shape on the screen. The DRAW uses the current HCOLOR = value while the XDRAW uses the complement of the color on the screen. Use XDRAW for cursors because when used at an existing cursor, it will remove it, restoring any underlying shape. You can use DRAW best on background shapes and characters; XDRAW on foreground (moving) shapes and cursors. Syntax required is just the shape number, but you can give the position with an AT like the example given above. HCOLOR =, SCALE =, and ROT = must be set first.

The HCOLOR = sets the drawing color as described earlier. Shapes work best in black and white: zero for black, three for white.

The SCALE = command lets you magnify the shape about its start position on the screen. Use SCALE = 1 normally. If you increase the scale, you must be sure that there is enough room. SCALE = 2 doubles the size, SCALE = 3 triples the size, and so on.

The ROT = command lets you rotate the shape defined to another orientation. ROT = 0 is normal. ROT = 16 rotates by 90° clockwise; ROT = 32 rotates 180°; ROT = 48 rotates by 270° clockwise, 90° counterclockwise. Rotations above 63 aren't defined. These four — 0, 16, 32, 48 — are the most useful.

Loading a shape table is simple, and can be done by any program that uses it. Although Applesoft has a SHLOAD command for tape

shape table loads, you probably will want to keep shape tables on disk and load them from there. So, the disk loads are described first.

From disk, shape tables can be BLOADed just like any other binary file. Then you must adjust the memory map pointers to protect it from the running program. And finally, you have to put the address of the shape table into a Page Zero pointer so that Applesoft will know where to find it. Here's how you might do it:

```
           LOMEM:16384 : CLEAR          :REM $4000
           BLOAD SHAPETABLE,A$1800
           POKE 232,0 : POKE 233,24     :REM $1800
```

The Page Zero pointer is at $E8 and $E9; hence the POKEs to 232 and 233. Another way might be:

```
           HIMEM:32768 : CLEAR          :REM $8000
           BLOAD SHAPETABLE, A$8000
           POKE 232,0 : POKE 233,128    :REM $8000
```

It's up to you. The shape table can reside anywhere as long as its start address is stuffed into $E8.E9.

The SHLOAD command will load from tape, put the table to fit below the current HIMEM, then change HIMEM to the beginning of the table, protecting it. It also sets $E7.E8 to the beginning of the table. So, just be sure no strings were referenced before the SHLOAD and it will set all the pointers for you, automatically.

A shape table has three parts, each following the other in memory. First there is one byte containing the number of entries. You read this number when you use the

```
           NS = PEEK(FN AD(232) )
```

statement that reads the first byte of the shape table as referenced by $E8.E9 in Page Zero. The second byte of a shape table is unused and can be ignored.

The second part of a shape table is the index to the shapes. This index has two bytes for each shape in the table, so it is 2*NS in size, where NS is the number of shapes kept in the first part. Each of the indexes has two bytes and is a relative address in low-byte/high-byte order. The relative address is from the beginning of the shape table to

the first byte in the shape being indexed. So, the first index contains the relative address of the first shape, the second index contains the relative address of the second shape, and so on. There can be any number of shapes, from one to 255; however, it may be wise to limit this number to 127 if you make the shape tables for yourself. Then the indexes will be easier to access, since they contain two bytes each. See Figs. 6-10 and 6-11.

The third part of a shape table is the set of shapes themselves. Each shape contains one or more bytes, the last one being zero. This is important, because the zero byte tells the Applesoft drawing routines where the shape ends. All the nonzero bytes then contain the shape and will be used by the DRAW or XDRAW routines, one after the other, until the zero byte is reached.

Shapes are drawn by plotting single points and moving the current drawing location to an adjacent pixel and there plotting the next point. This is repeated until the shape is complete. After plotting, each move can be in one of four directions — up, down, left, or right — to reach the next pixel. Each plot-then-move appears in the shape table as a *small instruction* to the DRAW routine called a *vector*. The shape is defined as a sequence of plot-then-move vectors that instructs the DRAW routine in Applesoft.

There are four different plot-then-move vectors: up, right, down, and left. In addition, the PLOT routine will handle vectors that don't plot but just move. There are four of these: up, right, down, and left. So, there are eight vectors you can use to make shapes; here are their codes:

Binary	Hex	Symbol	Vector Description
000	0	↿	move up
001	1	o→	move right
010	2	↾	move down
011	3	→o	move left
100	4	↓	plot-then-move up
101	5	•→	plot-then-move right
110	6	�???	plot-then-move down
111	7	→•	plot-then-move left

All vectors move the drawing position by exactly one pixel. You use them to draw the shape that you want.

RELATIVE
ADDRESS

00	0 1	NUMBER OF SHAPES
01		UNUSED
02	0 4	0 0	INDEX TO 1ST SHAPE
04		

FIRST SHAPE

0 0
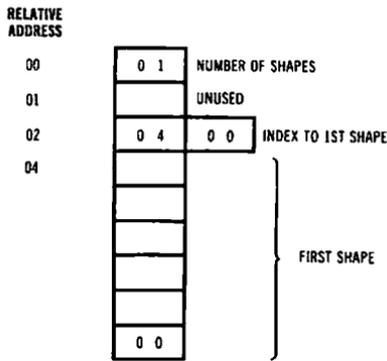
Fig. 6-10. A shape table with one entry.

RELATIVE
ADDRESS

00	0 3		NUMBER OF SHAPES
01			UNUSED
02	0 7	0 0	INDEX TO 1ST SHAPE
04	0 D	0 0	INDEX TO 2ND SHAPE
06	1 4	0 0	INDEX TO 3RD SHAPE
07			

FIRST SHAPE

0 0

0D

SECOND SHAPE

0 0

14

THIRD SHAPE

0 0
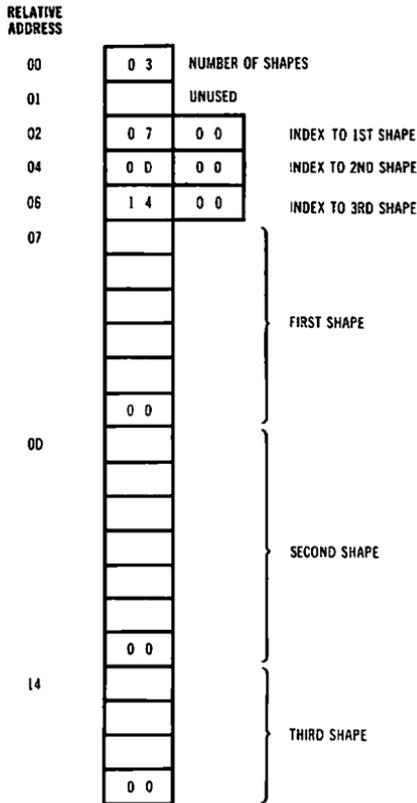
Fig. 6-11. A shape table with three entries.

The vectors are kept in the bytes of the entries in the shape table. However, since each vector only requires three bits for storage, the vectors are packed by stuffing two or three of them into each byte. To follow the packing, see Fig. 6-12. You can see it is partitioned into three chunks called Vector One, Vector Two, and Vector Three. Each of the first two are three bits in size while Vector Three is only two bits. This means that Vector Three can hold simple move vectors only; the larger plot-then-move vector codes are just too big.

The solution to the Vector Three size limitation is to defer the plot-then-move vectors to Vector One of the next byte. The rule for packing vectors into bytes is to start with Vector One. The next vector goes into Vector Two. After that, if the third vector is less than four in size, then it goes into Vector Three. Otherwise, the byte is full and the third vector goes into Vector One of the following byte. Continue like that until the end of the shape, at which time an extra zero byte is appended to complete the entry.

To create shape tables, the procedure is to pack the vectors into bytes to create each shape entry. Then, a table is built with the number of entries (shapes) in the first byte, an index table of relative addresses starting at the third byte, and the index table followed by the shape entries, with each entry ending in a zero byte. The routines needed to do these things are not difficult to write and some examples are given later on. However, the procedure for packing vectors into bytes becomes complicated because of an anomaly in the original design.

The vector code for move up is zero. This code conflicts with the end-of-shape marker which is also zero. One result of this conflict is that three move ups in a single byte gives a zero byte terminating the entire shape. Any further shape vectors are ignored, so you get only that part of the shape drawn before the three move ups. .

Another consequence of the clashing codes is ignored move ups. If Vector Three is zero, it is presumed to be empty and ignored. So, you

VECTOR THREE

VECTOR TWO
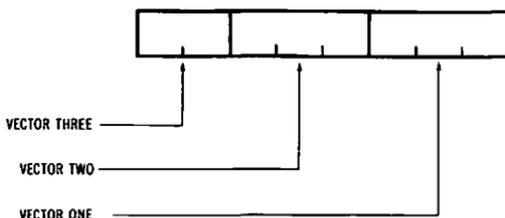
VECTOR ONE

Fig. 6-12. A shape byte.

can't put a move up in Vector Three, but it can be deferred to the next byte. But if both Vector Three and Vector Two are zero, then the Vector Two that you think *must* be a move up, isn't. It is ignored as well. A zero in Vector One is all right as long as there is a nonzero Vector Two or Vector Three, but you can't encode move ups in Vector Two unless Vector Three turns out to be a nonzero entry.

At first it appears that you can't use the simple move-up vector. By avoiding its use entirely, you can build shape tables without any hassles. If you try to use them, you get unexpected results. Shapes are distorted from any missed move ups. Some shapes are only partially drawn because of several move ups in sequence. But some move ups *do* work properly, so unless you know about the move-up anomaly, you will become confused and frustrated with your results.

Therefore, one solution to the anomaly is to avoid using the move up in any shape creation programs.

Another solution is to write a more complicated packing routine that handles move ups. An example of such a packing routine is given here.

Here's how vectors are packed into a shape table entry while avoiding the move-up anomaly. The routine is called VPACK; you can see it in the listing of Example 6-4.

The current byte being packed is pointed to by ZEND in Page Zero. The number of the vector within that byte is 1, 2, or 3 in VECNUM. The vector to be packed is in the A-reg when VPACK is called.

VPACK interprets the vector number of VECNUM and branches accordingly to VPACK1, VPACK2, or VPACK3.

For Vector One, the VPACK1 block just writes the A-reg to the current byte. Since the vector code is between zero and seven, it is already positioned in the three least significant bits, so it becomes the byte with Vector One.

For Vector Two, the vector in the A-reg must be shifted left by three bits to be in the Vector Two position. Then it must be put into the current byte without altering the Vector One already resident there. The ORA instruction in the VPACK2 block accomplishes this.

Vector Three is the tricky one. The first thing done at VPACK3 is to check if the vector is zero or a plot-then-move type. If so, then the packing must be deferred to the next byte. Otherwise, the vector is 1, 2, or 3 in value and may be packed into the current byte — shifted left six bits and an ORA. After packing Vector Three, the pointer ZEND must be advanced to access the next byte; the block that advances ZEND is called NEXT.

Example 6-4.

```
SOURCE FILE: EXAMPLE 6.4
0000:            1  *******************************
0000:            2  * EXAMPLE 6.4                  *
0000:            3  *                              *
0000:            4  *      SHAPE TABLE WRITER      *
0000:            5  *                              *
0000:            6  * FOR DETAILS ON HOW TO USE    *
0000:            7  * THESE ROUTINES, SEE TEXT.    *
0000:            8  *                              *
0000:            9  *******************************
0000:           10  *
0000:           11  *         E Q A T E S
0000:           12  *
0000:           13  * CONSTANTS
0000:           14  *
0000:           15  KUP     EQU  $00       FOR UPCURSOR
0001:           16  KRIGHT  EQU  $01       FOR RIGHTCURSOR
0002:           17  KDOWN   EQU  $02       FOR DOWNCURSOR
0003:           18  KLEFT   EQU  $03       FOR LEFTCURSOR
0058:           19  KLANDR  EQU  $58       LEFT AND RIGHT
0000:           20  *
0000:           21  *
0000:           22  * PAGE ZERO
0000:           23  *
00E7:           24  ZEND    EQU  $E7       TABLE POINTER
0050:           25  VECNUM  EQU  $50
0052:           26  PEN     EQU  $52       PEN MASK
0053:           27  PENX    EQU  $53       ALTERNATE PEN MASK
0094:           28  HIGHDS  EQU  $94       USED BY BLTU
0096:           29  HIGHTR  EQU  $96       USED BY BLTU
009B:           30  LOWTR   EQU  $9B       USED BY BLTU
0000:           31  *
0000:           32  *
1800:           33  TABLE   EQU  $1800     START SHAPE TABLE
0000:           34  *
0000:           35  *
0000:           36  * APPLESOFT & MONITOR
0000:           37  *
D393:           38  BLTU    EQU  $D393     BLOCK TRANSFER UP
0000:           39  *
0000:           40  *
0000:           41  *
0000:           42  *
0000:           43  *     R O U T I N E S
0000:           44  *
0000:           45  *
----- NEXT OBJECT FILE NAME IS EXAMPLE 6.4.OBJ0
8000:           46            ORG  $8000
8000:           47  *
8000:           48  *
8000:           49  * TEST MAINLINE *****
8000:           50  *
8000:20 60 80    51            JSR  SETUP
8003:20 95 80    52            JSR  KEY
8006:20 0C 80    53            JSR  ACCEPT
8009:4C 69 FF    54            JMP  $FF69     MONITOR
```

Example 6-4 Cont.

```
800C:              55 *
800C:              56 *
800C:              57 * ACCEPTED SHAPE TABLE - INSERT
800C:              58 * WITH NEW INDEX, THEN SETUP.
800C:              59 *
800C:AD 00 18      60 ACCEPT LDA TABLE       IF ONLY NULL
800F:C9 01         61            CMP #1       THEN DON'T BLTU.
8011:F0 07         62            BEQ ACC1
8013:A4 94         63            LDY HIGHDS
8015:A5 95         64            LDA HIGHDS+1
8017:20 93 D3      65            JSR BLTU      MOVE OLD SHAPES BY 2.
801A:              66 *
801A:A0 01         67 ACC1   LDY #1
801C:A9 00         68            LDA #0        MAKE NEW NULL SHAPE.
801E:91 E7         69            STA (ZEND),Y
8020:C8            70            INY
8021:91 E7         71            STA (ZEND),Y
8023:              72 *
8023:EE 00 18      73            INC TABLE     BUMP NUMBER ENTRIES
8026:              74 *
8026:18            75            CLC
8027:A5 E7         76            LDA ZEND      CALCULATE ADDRESS OF
8029:69 02         77            ADC #2        NEW NULL RECORD
802B:85 E7         78            STA ZEND
802D:A5 E8         79            LDA ZEND+1
802F:69 00         80            ADC #0
8031:85 E8         81            STA ZEND+1
8033:              82 *
8033:AD 00 18      83            LDA TABLE     FIND INDEX TO NEW
8036:0A            84            ASL A         NULL.
8037:AA            85            TAX
8038:              86 *
8038:38            87            SEC
8039:A5 E7         88            LDA ZEND
803B:E9 00         89            SBC #>TABLE   CALCULATE INXEX
803D:9D 00 18      90            STA TABLE,X   FOR NEW NULL.
8040:A5 E8         91            LDA ZEND+1
8042:E9 18         92            SBC #<TABLE
8044:9D 01 18      93            STA TABLE+1,X
8047:              94 *
8047:CA            95 ACC2   DEX
8048:CA            96            DEX
8049:F0 14         97            BEQ ACC3      WHILE INDEX, DO
804B:18            98            CLC
804C:BD 00 18      99            LDA TABLE,X
804F:69 02        100            ADC #2
8051:9D 00 18     101            STA TABLE,X
8054:BD 01 18     102            LDA TABLE+1,X
8057:69 00        103            ADC #0
8059:9D 01 18     104            STA TABLE+1,X BUMP INDEX BY 2
805C:4C 47 80     105            JMP ACC2
805F:             106 *
805F:60           107 ACC3   RTS
8060:             108 *
8060:             109 *
8060:             110 *
```

Example 6-4 Cont.

```
8060:           111 * SET THE POINTERS FOR THE
8060:           112 * SHAPE TABLE.
8060:           113 *
8060:AD 02 18   114 SETUP   LDA   TABLE+2   FIRST INDEX
8063:18         115         CLC
8064:69 00      116         ADC   #>TABLE   OFFSET INDEX
8066:85 9B      117         STA   LOWTR     TO GET ADDRESS OF
8068:AD 03 18   118         LDA   TABLE+3   FIRST SHAPE.
806B:69 18      119         ADC   #<TABLE
806D:85 9C      120         STA   LOWTR+1
806F:           121 *
806F:AD 00 18   122         LDA   TABLE     NUMBER OF SHAPES
8072:0A         123         ASL   A
8073:AA         124         TAX
8074:BD 00 18   125         LDA   TABLE,X   LAST INDEX
8077:18         126         CLC
8078:69 00      127         ADC   #>TABLE
807A:85 96      128         STA   HIGHTR    OFFSET INDEX TO GET
807C:BD 01 18   129         LDA   TABLE+1,X ADDRESS OF NULL
807F:69 18      130         ADC   #<TABLE   SHAPE.
8081:85 97      131         STA   HIGHTR+1
8083:           132 *
8083:18         133         CLC
8084:A5 96      134         LDA   HIGHTR
8086:69 02      135         ADC   #2
8088:85 94      136         STA   HIGHDS    BEYOND NULL SHAPE
808A:85 E7      137         STA   ZEND      END OF TABLE.
808C:A5 97      138         LDA   HIGHTR+1
808E:69 00      139         ADC   #0
8090:85 95      140         STA   HIGHDS+1
8092:85 E8      141         STA   ZEND+1
8094:60         142         RTS
8095:           143 *
8095:           144 *
8095:           145 * KEYBOARD COMMAND INTERPETER
8095:           146 * FOR THE CURSOR KEYS: IJKM AND
8095:           147 * <SP> FOR THE PEN.  EXITS WITH
8095:           148 * 'A' FOR ACCEPT OR 'R' FOR
8095:           149 * REJECT IN THE A-REG.
8095:           150 *
8095:A9 00      151 KEY     LDA   #0        INIT PEN UP
8097:85 52      152         STA   PEN
8099:A9 01      153         LDA   #1
809B:85 50      154         STA   VECNUM    INIT VECTOR 1
809D:A9 04      155         LDA   #4        MASK FOR PEN DOWN
809F:85 53      156         STA   PENX
80A1:A2 00      157         LDX   #0        FOR INDEXED INDIRECT!
80A3:           158 *
80A3:AD 00 C0   159 KEY0    LDA   $C000     KEYBOARD
80A6:10 FB      160         BPL   KEY0
80A8:2C 10 C0   161         BIT   $C010     CLEAR STROBE
80AB:           162 *
80AB:C9 D2      163         CMP   #'R'      REJECT SHAPE?
80AD:D0 01      164         BNE   KEY1
80AF:60         165         RTS
```

Example 6-4 Cont.

```
80B0:            166 *
80B0:C9 C1       167 KEY1    CMP  #'A'     ACCEPT SHAPE?
80B2:D0 01       168         BNE  KEY2
80B4:60          169         RTS
80B5:            170 *
80B5:C9 A0       171 KEY2    CMP  #'      ' CHANGE PEN?
80B7:D0 0D       172         BNE  KEY3
80B9:A5 52       173         LDA  PEN
80BB:48          174         PHA
80BC:A5 53       175         LDA  PENX    SWAP PEN AND PENX
80BE:85 52       176         STA  PEN
80C0:68          177         PLA
80C1:85 53       178         STA  PENX
80C3:4C A3 80    179         JMP  KEY0
80C6:            180 *
80C6:C9 C9       181 KEY3    CMP  #'I'     UP?
80C8:D0 0A       182         BNE  KEY4
80CA:A9 00       183         LDA  #KUP
80CC:05 52       184         ORA  PEN
80CE:20 01 81    185         JSR  VPACK    INSERT UPVECTOR
80D1:4C A3 80    186         JMP  KEY0
80D4:            187 *
80D4:C9 CB       188 KEY4    CMP  #'K'     RIGHT?
80D6:D0 0A       189         BNE  KEY5
80D8:A9 01       190         LDA  #KRIGHT
80DA:05 52       191         ORA  PEN
80DC:20 01 81    192         JSR  VPACK    INSERT RIGHTVECTOR
80DF:4C A3 80    193         JMP  KEY0
80E2:            194 *
80E2:C9 CD       195 KEY5    CMP  #'M'     DOWN?
80E4:D0 0A       196         BNE  KEY6
80E6:A9 02       197         LDA  #KDOWN
80E8:05 52       198         ORA  PEN
80EA:20 01 81    199         JSR  VPACK    INSERT DOWNVECTOR
80ED:4C A3 80    200         JMP  KEY0
80F0:            201 *
80F0:C9 CA       202 KEY6    CMP  #'J'     LEFT?
80F2:D0 0A       203         BNE  KEY7
80F4:A9 03       204         LDA  #KLEFT
80F6:05 52       205         ORA  PEN
80F8:20 01 81    206         JSR  VPACK    INSERT LEFTVECTOR
80FB:4C A3 80    207         JMP  KEY0
80FE:            208 *
80FE:4C A3 80    209 KEY7    JMP  KEY0     JUST KIDDING!
8101:            210 *
8101:            211 *
8101:            212 *
8101:            213 *  VECTOR PACK ROUTINE TO PUT
8101:            214 * THE A-REG VECTOR INTO THE NEW
8101:            215 * SHAPE.  X-REG MUST BE ZERO.
8101:            216 * A-REG MUST BE VECTOR (0..7).
8101:            217 * Y-REG AND A-REG CLOBBERED.
8101:            218 *
8101:A4 50       219 VPACK   LDY  VECNUM
8103:C0 01       220         CPY  #1
```

Example 6-4 Cont.

```
8105:D0 05      221          BNE    VPACK2
8107:           222 *
8107:81 E7      223 VPACK1   STA    (ZEND,X)   VECTOR 1
8109:E6 50      224          INC    VECNUM
810B:60         225          RTS
810C:           226 *
810C:C0 02      227 VPACK2   CPY    #2
810E:D0 0A      228          BNE    VPACK3
8110:0A         229          ASL    A          VECTOR 2
8111:0A         230          ASL    A
8112:0A         231          ASL    A
8113:01 E7      232          ORA    (ZEND,X)
8115:81 E7      233          STA    (ZEND,X)
8117:E6 50      234          INC    VECNUM
8119:60         235          RTS
811A:           236 *
811A:C9 00      237 VPACK3   CMP    #0         VECTOR 3
811C:F0 19      238          BEQ    DEFER      IF VECTOR ISN'T
811E:C9 04      239          CMP    #4         1, 2, OR 3 THEN DEFER
8120:B0 15      240          BCS    DEFER      TO NEXT BYTE.
8122:0A         241          ASL    A
8123:0A         242          ASL    A
8124:0A         243          ASL    A
8125:0A         244          ASL    A
8126:0A         245          ASL    A
8127:0A         246          ASL    A
8128:01 E7      247          ORA    (ZEND,X)   PUT IN HIGHEST
812A:81 E7      248          STA    (ZEND,X)   TWO BITS.
812C:           249 *
812C:A9 01      250 NEXT     LDA    #1         POINT TO THE
812E:85 50      251          STA    VECNUM     FIRST VECTOR OF
8130:E6 E7      252          INC    ZEND       THE NEXT BYTE.
8132:D0 02      253          BNE    *+4
8134:E6 E8      254          INC    ZEND+1
8136:60         255          RTS
8137:           256 *
8137:48         257 DEFER    PHA
8138:A1 E7      258          LDA    (ZEND,X)   IF BYTE < 8
813A:C9 08      259          CMP    #8         THEN V2 IS AN ILLEGAL
813C:B0 10      260          BCS    DEFER1     MOVE-UP CODE.
813E:09 58      261          ORA    #KLANDR
8140:81 E7      262          STA    (ZEND,X)   SO, REPLACE WITH A
8142:20 2C 81   263          JSR    NEXT       LEFT-AND-RIGHT, THEN
8145:A9 00      264          LDA    #0         A MOVEUP IN NEXT V1.
8147:20 01 81   265          JSR    VPACK
814A:68         266          PLA
814B:4C 01 81   267          JMP    VPACK      FINALLY, INTO V2 OF NEX
T!
814E:           268 *
814E:20 2C 81   269 DEFER1   JSR    NEXT       JUST TOO BIG FOR V3
8151:68         270          PLA
8152:4C 01 81   271          JMP    VPACK      SO PUT IT IN V1 OF NEXT

8155:           272 *
```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

Now it is left with the deferred vector to pack. The DEFER block calls NEXT to advance the pointers, then calls the VPACK routine in the confidence that it will pack into Vector One of the next byte. A special case where both Vector Two and Vector Three are zero is treated slightly different. The zero in Vector Two must be a previous move up. To render it valid, it too must be moved to the next byte. A special mask called KLANDR (for left-and-right) ORAd into the byte makes Vector Two a move left and Vector Three a move right. These vectors cancel each other, so they act like a nonzero no operation. Then the move up is put into the next byte's Vector One, followed by the vector of the current call into Vector Two. The two JSR VPACK instructions after the JSR NEXT accomplish this.

Except for the tricks in packing for Vector Three, the VPACK routine is simple enough. To use it yourself, set ZEND and VECNUM initially to the first byte of your shape's memory area, and to Vector One with the value one. Each vector you add to the shape as you build must be in the A-reg when you call VPACK. Make sure the X-reg is zero. When finished, VPACK will point to the last byte.

Once you have a shape, you must somehow put it into a shape table, with or without any previous entries, and correct the indexes to point properly to your new entry and any previous entries. There are two ways to do this: either use a previously extended table with fixed blocks for the anticipated number of entries, or start with a null table and use an indexed sequential append routine. The first is easier to write from scratch, but the second is easy to use and more efficient in memory management.

The listing has routines for appending to a shape table. Here's how they work.

The routines assume a shape table already resides at $1800. See Fig. 6-13. To begin, you must put a table at $1800. Such an initial table without any shapes is called a *null table* and you can enter it easily from the Monitor by

1800: 01 00 04 00 00

where the first byte is the number of entries, the third and fourth bytes point to the fifth byte (as $0004), which is a zero. This single entry of zero is called the null entry or null shape. Using this method of indexed sequential management, all shape tables will have a null as the last entry.

1800	0	1			NUMBER OF SHAPES
1801	0	0			
1802	0	4	0	0	INDEX TO NULL
1804	0	0			NULL

(A) Null shape table.

1800	0	1			NUMBER OF SHAPES
1801	0	0			
1802	0	4	0	0	INDEX TO NULL
1804	0	0			NULL
1805					
1806	////				
1807	////				NEW SHAPE
1808	////				
1809	////				

1800	0	2			NUMBER OF SHAPES
1801	0	0			
1802	0	6	0	0	INDEX TO FIRST SHAPE
1804	0	8	0	0	INDEX TO NULL
1806	////				
1807	////				
1808	////				FIRST SHAPE ENTRY
1809	////				
180A	0	0			
180B	0	0			NULL

(B) Before insertion.   **(C) After insertion.**

Fig. 6-13. Inserting the first entry into a shape table.

To add a new shape to the table at $1800, you use a routine that sets up Page Zero pointers, SETUP. One result of SETUP is the initialization of ZEND to the second byte after the end of the shape table (after the null entry). After the shape has been created by a routine that uses VPACK, the pointed ZEND will give the last byte of the new shape. The shape table at this time has not been altered in any way, so you can accept or reject the new entry. If rejected, you merely use SETUP to reset ZEND for another shape. If accepted, then the ACCEPT routine will append the new shape and you can call SETUP after to reset the pointers for the new, longer shape table.

As an example, consider your creation of the first entry. First, you make a null shape table at $1800.1804 as described above. Then you make up your new shape starting at $1806, two bytes beyond the end of the table. Let's say it has four bytes: $1806.1809. By a JSR to ACCEPT, it is inserted by putting its index at $1802.1803 where the index to the null was before. The index to the new null then goes into $1804.1805 where it has just enough room. At $180A a zero is added to the new shape to give it its terminator. The second zero following is

the new null: you can see its address $180B is reflected in the index $000B at $1804. The first entry was inserted by ACCEPT so simply because you created it just two bytes after the old table.

Inserting any other entry works the same way except that all the previous entries must be moved forward in memory by two bytes to make room for the new index. An Applesoft routine called BLTU handles this easily as long as it has its pointers set up for it by the SETUP routine.

Like the first entry, any other entry uses the shape you created two bytes past the end of the old table. (See Fig. 6-14) With old entries,

(A) Before insertion.   (B) After insertion.
Fig. 6-14. Inserting the n-th entry into a shape table.

these are moved ahead two bytes and the new index inserted by over-writing the old null. The new null index then follows in the two freed bytes. All the indexes of the previous entries must be advanced by two — a DO-WHILE handles this in ACCEPT. Again, the new entry includes a zero byte and a further zero byte becomes the new null.

You can follow the action easily by using the KEY routine together with SETUP and ACCEPT to enter shapes to a null table and dump $1800.183F, say, to watch the table grow.

The KEY routine is a keyboard entry to create the vectors to pass to VPACK. It begins by allowing move only vectors — hitting the space-bar switches PEN and PENX to allow plot-then-move vectors. Hitting the spacebar again will switch back to plot only vectors again. After you have used it, you may want to expand the KEY routine to include a LORES display with different colors for the pen-up, pen-down, and plotted points. Make your own custom Shape Table Editor.

CHAPTER SEVEN

Disk Operating System

7.1 STRUCTURES

The purpose of DOS is to create, maintain, and use data structures on 5¼-inch floppy disks. In the first section of this chapter, you can find all the details of these structures. Later, in the second section, you can find the protocols to use DOS in maintaining any structure at the level needed.

7.1.1 DOS on Disk

Disks created and used by DOS 3.3 are formatted into 35 tracks of sixteen sectors each. Each track is circular in shape and is on the top surface of the disk, concentric with the others. Track Zero is the outside track and is the longest. Track 34 is the shortest, being the innermost track. Although the longer outside tracks can hold more data than the shorter inside tracks, all tracks on the disk have the same storage capacity of sixteen sectors of 256 bytes each.

It is easier to picture the disk as a rectangular map rather than to draw the circular tracks. This map has a grid of 35 tracks by sixteen sectors and is partitioned into the sections DOS creates with the INIT command.

Tracks Zero, One, and Two are dedicated to DOS. Whenever a disk is INITed (initialized), it formats the entire disk by writing markers that create tracks and sectors. Then it copies itself into the first three

tracks with its bootstrap routines in Track Zero. This gives you a slave disk that will bootstrap into the original DOS's memory area, below $C000.

The middle track, Track 17, contains a Catalog and a special sector called the Volume Table of Contents or VTOC. From here, DOS has a maximum arm motion of sixteen to reach any one of the other 34 tracks. Since it must always reference the VTOC and Catalog, this is ideal.

After using the space for itself and the Catalog, 496 of the original 560 sectors remain for storing the files. One file called the *greeting program* and usually named "HELLO" will be loaded and run whenever the disk is booted. DOS will use and release space in the files storage areas as needed.

Two utilities you may need to work with DOS disks are a DISK MAP that usually draws the disk map using LORES graphics to show the allocation of sectors, and a DISK ZAP that lets you read, examine, change, and write to any designated sector on the disk. For debugging and file recovery you should at least have a DISK ZAP, such as Example 7-1.

Assuming a 48K Apple with the disk controller card in Slot Six, here is what happens when a DOS disk bootstraps.

Typing PR#6 or otherwise running the firmware at $C600 starts the Stage Zero bootstrap routine. A chunk of Page Three is written with a table to translate disk codes, wiping out any vectors or routines there. Then it loads in the Track Zero, Sector Zero page at $800. Finally, it jumps to $801 which is the Stage One bootstrap just loaded.

The Stage One bootstrap routine loads the remainder of Track Zero to memory starting at $B700 (SLAVE) or $3700 (MASTER). The SLAVE is the simplest procedure, since the MASTER will have to relocate itself later on. Assuming a SLAVE, the Stage One bootstrap routine finishes by jumping to $B700. See Table 7-1.

Finally, the Stage Two bootstrap routine completes the load from Tracks One and Two. The Boot One stage in $800.8FF is moved to $B600 where it remains. It forces a subsequent load of the bank switched RAM by writing a $00 to a location there. If a BASIC was loaded by a previous bootstrap routine, this will cancel it, forcing it to be re-loaded.

You can remove this feature by canceling the instruction to zero the RAM. From the monitor, type

<div align="center">BFD3: EA EA EA</div>

Example 7-1.

```
>LIST
    1 REM EXAMPLE 7.1
    2 REM
    3 REM   D I S K    Z A P
    4 REM
    5 REM   IN INTEGER BASIC
    6 REM
   10 GOTO 30000
  100 REM
  101 REM   CHR$ FUNCTION (TOGNAZINNI)
  102 REM
  110 CHS=CHR+128*(CHR<128)
  120 LC1= PEEK (224):LC2= PEEK (225)-(LC1>243)
      : POKE 79+LC1-256*(LC2>127)+(LC2-255*(LC2
      >127))*256,CHS:CHR$=".": RETURN
  500 REM
  501 REM    MONITOR COMMAND CALL
  502 REM
  510 FOR H=1 TO LEN(HEX$): POKE 511+H, ASC(HEX
      $(H)): NEXT H: POKE 72,0
  520 CALL -144
  530 RETURN
  600 REM
  601 REM    CALL THE DISK
  602 REM
  610 CALL RWTS
  612 POKE IOBVOL,255: REM RESET DOS
  620 ERR=0: IF PEEK (0)#0 THEN ERR= PEEK (IOBC
      ODE)
  630 IF ERR#0 THEN RETURN
  640 VTAB 2: TAB 4
  650 PRINT "CURRENT TRACK = ";TRK;", SECTOR =
      ";SEC;".    "
  660 RETURN
 1200 REM
 1201 REM    PARSE TRACK & SECTOR
 1202 REM
 1210 ERR=1:P=2
 1212 IF P> LEN(A$) THEN RETURN
 1220 TRK= ASC(A$(P,P))-176: IF TRK<0 OR TRK>9 THEN
      RETURN
 1230 P=P+1: IF P> LEN(A$) THEN RETURN
 1240 TRK1= ASC(A$(P,P))-176: IF TRK1<0 OR TRK1
      >9 THEN 1270
 1250 TRK=TRK1+10*TRK: IF TRK<0 OR TRK>34 THEN
      RETURN
 1260 P=P+1: IF P> LEN(A$) THEN RETURN
 1270 IF A$(P,P)#"," THEN RETURN
 1272 P=P+1: IF P> LEN(A$) THEN RETURN
 1280 SEC= ASC(A$(P,P))-176: IF SEC<0 OR SEC>9 THEN
      RETURN
 1290 ERR=0:P=P+1: IF P> LEN(A$) THEN RETURN
 1300 ERR=1:SEC1= ASC(A$(P,P))-176: IF SEC1<0 OR
      SEC>9 THEN RETURN
```

Example 7-1 Cont.

```
 1310 SEC=SEC1+10*SEC: IF SEC<0 OR SEC>NSEC-1 THEN
      RETURN
 1320 IF P< LEN(A$) THEN RETURN
 1330 ERR=0: RETURN
12000 REM
12001 REM    READ COMMAND
12002 REM
12010 ERR=0
12020 P=2: GOSUB 1200: REM PARSE T,S
12040 IF ERR=0 THEN 12080
12060 ERR=0: PRINT " ₤₤₤    ???SYNTAX???": RETURN

12080 POKE IOBVOL,0
12100 POKE IOBTRK,TRK
12120 POKE IOBSEC,SEC
12140 POKE IOBBUF,0: REM $2000
12160 POKE IOBBUF+1,32
12180 POKE IOBCMD,1: REM    READ
12200 GOSUB 600: REM RWTS
12220 IF ERR=0 THEN 12260
12240 POKE 34,3: POKE 35,19: CALL -936: GOTO 12280

12260 A$="L":L=1: GOSUB 14000
12280 RETURN
13000 REM
13001 REM    WRITE COMMAND
13002 REM
13010 ERR=0
13020 P=2: GOSUB 1200: REM PARSE T,S
13040 IF ERR=0 THEN 13080
13060 ERR=0: PRINT " ₤₤₤    ???SYNTAX???": RETURN

13080 POKE IOBVOL,0
13100 POKE IOBTRK,TRK
13120 POKE IOBSEC,SEC
13140 POKE IOBBUF,0: REM    $2000
13160 POKE IOBBUF+1,32
13180 POKE IOBCMD,2: REM WRITE
13200 GOSUB 600: REM RWTS
13220 RETURN
14000 REM
14001 REM    LIST OTHER HALF BUFFER
14002 REM
14010 ERR=255
14090 IF LEN(A$)>1 THEN PRINT " ₤    ???EXTRA I
      GNORED???"
14100 POKE 34,2: POKE 35,19: CALL -936
14120 IF (L<1) OR (L>2) THEN 14160
14140 GOSUB 14100+L*200
14160 RETURN
14300 REM FIRST HALF OF BUFFER
14310 POKE 2,BUF1 MOD 256: POKE 3,BUF1/256
14320 FOR LINE=0 TO 15: POKE 2,(BUF1+LINE*8) MOD
      256
```

Example 7-1 Cont.

```
14330 VTAB LINE+4: TAB 1
14340 CALL 8448: REM DUMP
14350 NEXT LINE
14360 ERR=0:L=2: RETURN
14500 REM SECOND HALF OF BUFFER
14510 POKE 2,BUF2 MOD 256: POKE 3,BUF2/256
14520 FOR LINE=0 TO 15: POKE 2,(BUF2+LINE*8) MOD
      256
14530 VTAB LINE+4: TAB 1
14540 CALL 8448: REM DUMP
14550 NEXT LINE
14560 ERR=0:L=1: RETURN
15000 REM
15001 REM     CHANGE COMMAND
15002 REM
15010 ERR=0
15020 IF LEN(A$)>4 THEN 15060
15040 PRINT "   ₤   ???NOT ENOUGH???": RETURN
15060 HEX$="20":HEX$(3)=A$(2)
15080 HEX$( LEN(HEX$)+1)=" N E88AG"
15100 GOSUB 500: REM MONITOR
15120 RETURN
28000 REM
28001 REM     GET AN INSTRUCTION
28002 REM
28020 POKE 34,20: POKE 35,23
28030 VTAB 23: TAB 1
28060 PRINT ".";: INPUT A$: IF LEN(A$)<1 THEN 28060

28080 FOR INST=1 TO CMDSIZ
28100 IF A$(1,1)=CMD$(INST,INST) THEN RETURN
28120 NEXT INST
28140 PRINT " ₤    ???INVALID COMMAND(";CMD$;"
      )???"
28180 GOTO 28060
29000 REM
29001 REM     INITIAL MENU
29002 REM
29020 TEXT : CALL -936: POKE 50,63: TAB 16: PRINT
      "DISK ZAP": POKE 50,255
29040 PRINT : PRINT "THIS PROGRAM WILL READ, WR
      ITE, AND"
29060 PRINT "EXAMINE ANY SECTOR ON THE DISK IN"

29080 PRINT "THE CURRENT DRIVE.  THE CONTENTS O
      F"
29100 PRINT "THE CURRENT SECTOR (LAST READ) MAY
       BE"
29120 PRINT "CHANGED.  BACKUP ANY WORK BEFORE Y
      OU"
29140 PRINT "USE THIS PROGRAM!!!"
29160 PRINT : PRINT "THE COMMANDS ARE:"
29180 TAB 5: PRINT "R<T>,<S> ... READ TRACK, SE
      CTOR"
```

Example 7-1 Cont.

```
29200 TAB 5: PRINT "W<T>,<S> ... WRITE TRACK, S
      ECTOR"
29220 TAB 12: PRINT "L ... LIST HALF BUFFER"
29260 TAB 5: PRINT "CNN: ETC ... CHANGE AT $NN"

29300 TAB 12: PRINT "Q ... QUIT THIS PROGRAM"
29320 REM
29340 REM
29380 REM
29400 RETURN
30000 REM
30001 REM    MAIN LINE
30002 REM
30010 TEXT : CALL -936: TAB 16: PRINT "DISK ZAP
      "
30012 VTAB 11: TAB 10: PRINT "... INITIALIZING
      ..."
30020 DIM HEX$(150): DIM A$(128)
30030 CMDSIZ=5: DIM CMD$(CMDSIZ)
30032 CMD$="QRWLC"
30034 BUF1=8192: REM   $2000
30036 BUF2=BUF1+128
30040 D$="": REM CTRL/D
30060 NSEC=16: REM DOS 3.3
30062 HEX$="2100:A0 00 A5 02 20 DA FD A9 AD 20
      F0 FD A9 A0 20 F0 N E88AG"
30064 GOSUB 500
30066 HEX$="2110:FD A0 00 B1 02 20 DA FD A9 A0
      20 F0 FD C8 C0 08 N E88AG"
30068 GOSUB 500
30070 HEX$="2120:D0 F1 A9 1D 18 65 28 85 28 A9
      00 65 29 85 29 A0 N E88AG"
30072 GOSUB 500
30074 HEX$="2130:07 B1 02 91 28 88 10 F9 60 N E
      88AG"
30076 GOSUB 500
30080 HEX$="300:A9 B7 A0 E8 20 D9 03 A9 FF B0 0
      2 A9 00 85 00 60 N E88AG"
30100 GOSUB 500: REM  MONITOR COMMAND
30140 IOB=-18456: REM $B7E8
30160 RWTS=768: REM $300
30180 IOBDRVN=IOB+2
30200 IOBVOL=IOB+3
30220 IOBTRK=IOB+4
30240 IOBSEC=IOB+5
30260 IOBBUF=IOB+8
30280 IOBCMD=IOB+12
30300 IOBCODE=IOB+13
30320 IOBOLDV=IOB+15
31000 GOSUB 29000: REM  MENU
31020 GOSUB 28000: REM    GET INSTRUCTION
31040 IF INST=1 THEN 32000: REM   Q=QUIT
31060 GOSUB 10000+(1000*INST): REM    INTERPET
```

Example 7-1 Cont.

```
31080 IF ERR#0 THEN 31500
31100 GOTO 31020
31120 REM
31500 REM
31501 REM    ERROR ROUTINE
31502 REM
31520 POKE 34,20: POKE 35,23: VTAB 23
31530 TAB 1
31540 A$="DISK": IF ERR=128 THEN A$="READ"
31542 IF ERR=16 THEN A$="PROTECT"
31544 IF ERR=32 THEN A$="VOLUME"
31560 TAB 8: PRINT "???";A$;" ERROR???"
31640 GOTO 31020
31650 REM

32000 TEXT : CALL -936
32010 PRINT "BYE!": PRINT
32767 END
```

Table 7-1. DOS Locations After Bootstrap

	Track		
Sector	0	1	1
0	B6	A1	B1
1	B7	A2	B2
2	B8	A3	B3
3	B9	A4	B4
4	BA	A5	B5
5	BB	A6	
6	BC	A7	
7	BD	A8	
8	BE	A9	
9	BF	AA	
A		AB	
B		AC	
C	9D	AD	
D	9E	AE	
E	9F	AF	
F	A0	B0	

before INITing any disk you don't want to force reloads.

Finally, the Stage Two bootstrap finishes by forcing a DOS cold start at $9D84.

The DOS cold start routine sets up buffers and HIMEM, creates vectors, especially in Page Three, and grabs the CSW and KSW hooks, and finally runs the HELLO program. You can cold start DOS again if you want it to initialize itself from scratch by using the Page Three vector:

$3D3 jumps to cold start

A less drastic choice is the warm start that simply recognizes the current BASIC again and jumps to the BASIC warm start at $E003. Again, use the Page Three vector:

$3D0 jumps to warm start

A warm start won't clobber the current BASIC program, so it is a good choice when you want to re-enter BASIC after working with the Monitor by typing 3D0G.

The greeting program is run automatically at the end of a DOS bootstrap. If the program is in Applesoft and Integer is resident instead, the DOS looks for a program called APPLESOFT on disk to load into RAM. This is an earlier version of Applesoft, and not described in this book. System disks usually have an alternate greeting Integer program called APPLESOFT that loads FPBASIC into the 16K RAM area instead.

For situations where your HELLO program won't know what BASIC, if any, is available, or must setup a new memory map, you will want a binary HELLO. Ordinarily, DOS won't BRUN a greeting program; it wants to use the RUN command instead. You can change this by typing

9E42: 34

from the Monitor into a 48K DOS 3.3, then INITing your new slave disk. After, delete the HELLO file and BSAVE your new binary HELLO in its place.

When any HELLO program loads and runs, it must be below $9600, to avoid overwriting DOS or its buffers. After it is loaded, you can change the map and relocate a binary program to fit between DOS and its buffers. This protects it from changes in MAXFILES and makes it quite invisible to BASIC. Here's how.

First, load the binary program into low RAM, $801. If you BSAVEd it from there, the bootstrap routine will do this. Then move the program upmemory to its resting place below $9D00, overwriting the buffers. Then, change the value of the pointer at $9D00: it pointed to the first buffer below itself; now you point it to a location at least 38 bytes below your program. And finally you do a JSR $A251 to rebuild the buffers *below* your program. This leaves your program between the buffers and the start of DOS proper at $9D00. The HIMEM will be at the beginning of the new buffer's area, with your program *hiding* in DOS. See Fig. 7-1 and Table 7-2.

Once it is established — between DOS and buffers or elsewhere — you may have to explore the Apple to see what version it is. Then you can set soft switches and put out messages if you don't have the features in the machine you need. The Monitor version can be found by looking at $FBB3: a $38 is in Standards, a $EA is in Autostarts, and a $06 is in IIe Monitors. The BASIC can be identified at $E000: $4C for Applesoft, $20 for Integer. If you need certain peripheral cards, this would be the time to check for them, too. Disassemble them at $Cn00 — where n is the slot — to see what unique values you can identify them with. Don't use the $C800.C8FF ROM area; it's not unique to any one slot.

You can allow more space on data disks than on program disks. Program disks need DOS on Tracks Zero, One, and Two. Data disks can be made without DOS on Tracks One or Two, freeing 32 sectors for additional file storage. Here's how.

Make a new slave disk the usual way, INITing a HELLO program. Delete the HELLO file. Then, use a Disk Zap utility to alter Track Zero/Sector Zero as follows:

```
FE: 0A 01
```

Then, change Track Zero/Sector One:

```
00:   20 93 FE 20 89 FE 20 58
08:   FC A6 2B 9D 88 C0 20 31
10:   F8 BA CA 9A 68 85 3D A9
18:   2A 85 3C A0 00 B1 3C F0
20:   06 20 ED FD C8 D0 F6 4C
28:   00 E0 C4 C1 D4 C1 A0 CF
30:   CE CC D9 A0 AD A0 CE CF
```

```
38:   A0 C4 CF D3 A0 CF CE A0
40:   C4 C9 D3 CB 87 00
```

where the chunk $2A.44 is a screen message and the zero at $45 ends that message. On Track 17/Sector Zero, the VTOC, you can release the Tracks One and Two by

Fig. 7-1. Hiding a binary program in DOS.

Table 7-2. Hiding a Binary Program

STEP 1:	DOS uses its buffers to BLOAD the program to low RAM at $800.
STEP 2:	When run, the program first copies itself to high RAM, just below $9D00, and overwrites the buffers.
STEP 3:	After setting $9D00.9D01 to point 38 locations below the program copy, the program calls DOS at $A251 that rebuilds the buffers below the copy.
STEP 4:	Afterwards, the program is abandoned, leaving it in copy between DOS and its buffers.

3C: FF FF 00 00 FF FF

when so changed. The catalog remains intact and can be used by DOS, but the disk won't bootstrap; it gives an error message instead.
Slave disk patches are summarized in Table 7-3.

Table 7-3. Summary of Slave Disk Patches

9E42: 34	Allows binary HELLO
AE34: 60	Removes CATALOG pauses
BFD3: EA EA EA	Removes forced re-loads of BASIC

NOTE: Use the Monitor to make modifications to DOS before INITing a new, modified slave disk.

7.1.2 Disk Files

Files on disk are managed by accessing disk sectors for reading, updating, and writing. On each disk, access is controlled beginning with one sector, the VTOC (see Fig. 7-2).

The VTOC or Volume Table of Contents resides at the same location on all disks — Sector Zero of Track 17. From there, the DOS File Manager can find all other sectors it wants. This is because of two parts of the VTOC, the Track Bit Map and the First Directory Link. The Track Bit Map shows all 560 sectors on the disk as being either in use or free. Then, the Link to the first directory sector tells DOS where the directory of files begins. From there, each entry points to the files themselves; files that are managed with indexes are called Track-Sector-Lists or TSLs. Each sector in the chain from VTOC to data can be traced by links.

Fig. 7-2. File management sector linkage.

Each link consists of two bytes, track number and sector number. Like a memory address pointer inside the Apple, a link points from sector to sector on the disk. The first link in the chain is in Bytes One and Two of the VTOC; it points to the sector containing the first seven directory entries. This link is usually $11 and $0F, pointing to Track 17, Sector 15.

Within each directory entry is the CATALOG information for one file and another link. This file link points to the TSL of the file where an entire index of links resides, pointing sequentially to all the data sectors of the file.

Each sector containing a set of directory entries or indexes may not be large enough. For instance, a directory of ten entries won't fit into one sector; there is only enough room for seven. The remaining three

must go elsewhere. So, the directory sector has a link to point to the next directory sector. This way, a list of directory sectors can be chained together with the last one having zeros in its link. Similarly, the TSL can hold more than the 122 data links if the data file should exceed 122 sectors. Like the directory sector list, the TSL can build a list to extend itself by using a link to the next TSL sector. This way, there is no logical limit to the lengths of the directory and its files.

The entire structure of link lists on the disk begins in the VTOC with the link to the first directory sector.

The VTOC also keeps track of which sectors have been allocated. Each sector is represented by one bit in the Track Bit Map: one track of sixteen sectors in two bytes. For each byte pair, the first byte keeps Sectors 15 to 8 and the second byte keeps Sectors 7 to 0. A bit that is on (1) flags the sector as free; a bit that is off (0) flags the sector as in use. Each track actually has four bytes, but the last two are unused and zeroed. With 35 tracks on disk, the Track Bit Map occupies 4 x 35 or 140 bytes in the VTOC. See Table 7-4.

With the Track Bit Map and the link lists, the disk management is maintained, starting at the VTOC.

Table 7-4. Volume Table of Contents (VTOC)

Track 17/Sector 0	
00	Unused
01.02	Link to first directory sector
03	DOS release number
04.05	Unused
06	Volume number of this disk
07.26	Unused
27	Number of indexes per TSL sector (= 122)
28.2F	Unused
30	Last allocation: track number
31	Last allocation: direction
32.33	Unused
34	Number of tracks/disk (= 35)
35	Number of sectors/track (= 16)
36.37	Number of bytes/sector (= 256)
38.C3	Track Bit Map: Tracks 0. . .34; four bytes each
C4.FF	Unused space for further entries
Detail of Track Bit Map Entry	
00	Bits 7. . .0 on for Sectors 15. . .8 free
01	Bits 7. . .0 on for Sectors 7. . .0 free
02.03	Unused (= 0)

The directory usually starts in Track 17/Sector 15. Aside from the link to the next directory sector, it just consists of seven entries of 35 bytes each. See Table 7-5.

Table 7-5a. Directory Sectors (Starts at 17/15)

00	Unused
01.02	Link to next Directory sector (00 = none)
03.0A	Unused
0B.2D	Directory entry, 1st
2E.50	Directory entry, 2nd
51.73	Directory entry, 3rd
74.96	Directory entry, 4th
97.B9	Directory entry, 5th
BA.DC	Directory entry, 6th
DD.FF	Directory entry, 7th

Table 7-5b. Detail of a Directory Entry

00.01	Link to file storage (TSL)
02	File type
03.20	File name
21.22	File length in sectors

NOTE: A zero track number in the link flags entry as unused.

Table 7-5c. File Type Codes

00	Text	80	locked Text
01	Integer	81	locked Integer
02	Applesoft	82	locked Applesoft
04	Binary	84	locked Binary
08	S-type	88	locked S-type
10	Relocatable	90	locked Relocatable
20	A-type	A0	locked A-type
40	B-type	C0	locked B-type

NOTE: Relocatable files are defined in Apple's DOS Toolkit Assembler — see manual. Types S, A, and B not defined at time of this writing.

Each entry has the link to its file as the first two bytes. A zero track number in the link flags the entry as unused, so that Track Zero cannot be reached from the directory. This is why any of Track Zero cannot be released for files storage when data disks are made. When a file is DELETEd, a zero is written to the first byte of the entry to flag it as unused, the original track number is copied to the last byte of the filename, Byte $20 of the entry, and all the file sectors are freed in the Track Bit Map in the VTOC. Until these sectors are reused by

another file, all of the DELETEd file information is still available. This is the basis of DELETE recovery schemes found in some disk utility packages.

The CATALOG command displays most of the directory entry. The file length is kept in two bytes (lo/hi) and can be greater than 255; however, the CATALOG only shows the low byte value. If you have a file of 255 sectors, that will show in the catalog, but if you increase the file size to 256 the CATALOG shows zero. The information is kept in the directory, however.

Occasionally un unprintable character gets itself hidden in a file name. A dump of the directory shows it quite clearly but the CATA-LOG won't. This is annoying because you will try to reference the file without any success because you don't really know its directory name. For instance, if a file called DOODLE has a ctrl/0 hidden between the two printable "0"s, then when you type

LOAD DOODLE

you just get FILE NOT FOUND error. Whenever this happens, check the file name in the directory by dumping it with a Disk Zap or an *improved* CATALOG utility. You probably will find one or more hidden control characters.

Where seven directory entries have been made, the eighth entry must go into another sector. This is usually Track 17/Sector 14. You can confirm this by looking at the link in Track 17/Sector 15: second and third bytes. If any directory sector has a zero directory link there, it means that there are no more sectors of the directory. There may be as many as fifteen directory sectors in Track 17; these will hold 15 × 7 or 105 directory entries, maximum.

Tracks 3 to 16 and Tracks 18 to 34 are used for data storage for a total of 496 sectors. If you make a data disk without DOS, then releasing Tracks One and Two increases your files storage area to 528 sectors. For each file you create, at least two sectors are used for file storage. A small file whose data fits into one sector of less than 256 bytes uses one sector for data and one sector for an index. These data sectors are called the TSL — track sector list. Then as more sectors of data are added later, they may be linked to the file in its TSL. So, the file storage is always at least one sector larger than the size of its data. See Table 7-6.

Table 7-6. Layouts of File Sectors

Track-Sector-List (TSL) Sectors	
00	Unused
01.02	Link to next TSL sector (0 = end)
03.0B	Unused
0C.FF	Indexes to file sectors: 122 links of two bytes each

Text File Format	
00. . .	(negative-ASCII record) 8D
	(negative-ASCII record) 8D
	. . .
	(negative-ASCII record) 8D
	. . .
	00
Records delimited by CR characters; file terminated by a zero byte.	

Integer File Format	
00.01	Program length in bytes
02. . .	Tokenized program

Applesoft File Format	
00.01	Program length in bytes
02. . .	Tokenized program

Binary File Format	
00.01	Address of memory image
02.03	Length in bytes
04. . .	Binary image of memory

Each file has one, perhaps more, TSL sectors. The first TSL sector is linked from the directory entry and indexes the sequence of data sectors of the file. It consists of 122 links: each link pointing to a data sector. The first link points to the first data sector, the second link points to the second data sector, and so on. If a file has more than 122 data sectors, a second TSL sector is linked from the first by the second and third bytes of the TSL. For most files, this link will be zero, indicating no further TSL sectors. As data and TSL sectors are allocated to the file, links are written and the Track Bit Map updated in the VTOC to show them in use. Similarly, when a file is DELETEd, the Directory link is zeroed and the sectors released in the Track Bit Map. The entire file, including TSL sectors and data sectors, is maintained this way. When a file is found from the link in its directory entry, it can be scanned sequentially by simply going through the TSL from top to bottom until the end of file is reached. From each TSL link, the data sector can be accessed.

Within each file, the data varies. For each file type, the end of file is recognized in a different way. For instance data in text files consists of negative-ASCII code only with a zero byte marking the end of file. This is the simplest method. When read, text files are accessed through the DOS File Manager one record at a time; each record recognized by a carriage return character at its end. This scheme is different than that used by the other file types.

Program files of Applesoft and Integer have many different codes. They have ASCII and keyword tokens so the end of file is given by the first two bytes of the data instead. For the first sector of the data only, the first two bytes contain the length in bytes (lo/hi) of the memory image of the program. Then DOS uses this number in its LOAD command to know when the end of file has been reached.

Binary files use a similar scheme. In the first sector of data, they keep both the starting address and the length of their binary image. This scheme occupies the first four bytes of data with the memory image to be loaded following. All file types, however, keep the end of file information one way or another with the data and _not_ in the TSL.

7.1.3 Disk Format

The physical sequence of sectors on each track is not the one you normally use. For a given track, Sector One follows Sector Zero with six other sectors between them. Sector One is actually in the physical sector number seven. See Table 7-7.

To compare the logical addressed sectors with the physical positional sectors, look at the Sector Interleaving table. What this scheme of addressing does is to allow a large gap between logical sectors to give the routines time between sector accesses. The time it takes to access a sector is then available sixfold before the next sequential sector need be accessed. No space on disk is lost: all logical sectors are there in the sixteen physical spaces.

This scheme was first adopted in DOS 3.3. Before that, thirteen sector disks under DOS 3.1, 3.2, and 3.2.1 all used a different scheme that is now obsolete. The newer sixteen sector disk system introduced with the Pascal Language system uses a simple sector interleaving system just described. The DOS 3.3 scheme is compatible with the Pascal Language system, but not equivalent. That is to say, DOS 3.3 can be used to read and write disks formatted by the Pascal Language system, but the sectors have different meanings.

Table 7-7. Sector Interleaving

Physical	Logical
0	0
1	D
2	B
3	9
4	7
5	5
6	3
7	1
8	E
9	C
A	A
B	8
C	6
D	4
E	2
F	F

In Pascal, the disk is formatted like a DOS disk, but the FILER considers the disk to have 280 *blocks* of a data capacity of 512 bytes each. So, each track contains eight blocks. The Pascal block numbers, Table 7-8, allow you to convert between the two systems if you want to access one system from the other's disks. From Pascal, you will use the BLOCKREAD and BLOCKWRITE; from DOS you will use the RWTS described in this chapter.

Table 7-8. Pascal Block Numbers

Pascal Block (mod 8)	DOS sectors	Physical sectors
0	0,E	0,8
1	D,C	1,9
2	B,A	2,A
3	9,8	3,B
4	7,6	4,C
5	5,4	5,D
6	3,2	6,E
7	1,F	7,F

NOTE: Track number = Block DIV 8.

Whenever you INIT a new disk, the first thing that DOS does is to completely format the disk by writing to each and every possible location. This *completely destroys* any previous contents of the disk. Also, it creates the tracks and sectors by the method known as *soft sectoring.*

If you look at a disk, you will see perhaps a small hole near the big, central hole. On some disks that are labeled *hard sectored,* you will see a ring of these holes around the hub hole. On hard-sectored systems, these holes tell the hardware where the sectors are by *chopping* the light from a small lamp into sync pulses. Each sync pulse tells the disk controller that a new sector is coming under the read/write head. But, the Apple uses soft-sectoring instead. On soft-sectored systems, the disk is formatted so that each track has a long burst of special bytes between each pair of sectors. Then the track is simply read and these special bytes are recognized by the DOS in the data stream. So, while the hard-sectored disks have sync pulses, soft-sectored disks like the Apple uses must be read to detect *sync bytes* that were formatted on the disk earlier.

There are sixteen sectors on each track. When formatted, they are made with sync bytes that have 10 bits each instead of the usual 8. The format is 8 bits on, 2 bits off. These sync bytes partition the track into sectors of two fields each as shown in Fig. 7-3.

Each sector contains an address field and a data field. The address field has the volume, track, and sector numbers. The data field has the 256 bytes you know as its contents encoded into it. There are a few sync bytes between the two.

Each track begins with a burst of sync bytes. Then the first sector is formatted with an address field, a few more sync bytes and a data

GAPS ARE 10-BIT SYNC BYTES
FIELDS ARE 8-BIT ENCODED BYTES

Fig. 7-3. Formatted disk track gaps and fields.

field. A burst of sync bytes follow before the second physical sector is made. When the last sector is formatted, its data field overwrites the first few sync bytes of the track. This way, the sync bytes take care of small variations in speed.

Formatting is done for the entire disk, from Track Zero to Track 35. Each track is made with the given volume number, the track number, and the logical sector numbers in the address fields of each sector. All gaps between sectors and fields are filled with 10-bit sync bytes; the fields themselves are made with 8-bit bytes.

Address fields are fourteen bytes each. They start and end with special values that make sure the READ routine knows where it is at. The prefix bytes are $D5, $AA, and $96; the suffix bytes are $DE, $AA, and $EB. This leaves eight bytes between them for address information: volume, track, sector, and checksum. Each number is one byte in value and is put into two bytes each, see Fig. 7-4.

Fig. 7-4. Sector address fields.

Each byte of header data is encoded as two bytes on the disk in the address field. This is done because of restrictions in reading bytes in synchronization. What happens is that the hardware can't read any byte value; they must have their bit 7s on and they can't have two zero bits contiguously. Of the many ways data can be encoded to meet these requirements, address fields use a four-bit encoding scheme that works as follows.

Consider any byte of eight bits you wish to encode as

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

For example, encode $3B which is 00111011 in binary. The byte is mapped to two bytes having their odd bits always on as

$$1b_71b_51b_31b_1 \text{ and } 1b_61b_41b_21b_0$$

so that the $3B becomes

$$10111111 \text{ and } 10111011$$

when encoded. The $3B has been mapped to $B7 and $BB.

You can decode two bytes from an address field to its value in one byte by reversing the steps. See the hardware protocols in Section 7.2.5.

If four-bit encoding was used for data fields as well, the track would only hold ten sectors because each chunk of 256 bytes would have to be encoded to 512 bytes. So, the data fields use another encoding scheme called six bit that maps 256 bytes to 342 bytes instead. A little more complicated, the six-bit scheme gives 16 tracks per sector.

Like the address field, the data field has a prefix of three bytes, encoded contents, and a suffix of three bytes. The prefix for data fields is $D5, $AA, and $AD; the suffix is the same: $DE, $AA, and $EB. The contents are encoded as six bits of data and a checksum. See Fig. 7-5.

Here's how the sixbit encoding scheme works. The 256 bytes are converted to 342 bytes that have bits 7 and 6 in each byte as zero. These six-bit values are in the range from $00 to $3F. A table called the Write Translate Table (see Table 7-9) looks up a unique legal byte for each six-bit value. Consider the following example.

Fig. 7-5. Sector data fields.

The negative-ASCII string "HELLO" has the six values: $C8, $C5, $CC, $CC, $CF, $A0. Write in binary and regroup the bits in chunks of six each:

```
$C8  110010 00
$C5  1100 0101
$CC  11 001100
$CC  110011 00
$CF  1100 1111
$A0  10 100000
```

Write each group of six bits as a byte of two zero bits and six data bits:

```
00110010 = $32
00001100 = $0C
00010111 = $17
00001100 = $0C
```

Table 7-9. Write Translate Table

Sixbit	Code	Sixbit	Code
00	96	10	B4
01	97	11	B5
02	9A	12	B6
03	9B	13	B7
04	9D	14	B9
05	9E	15	BA
06	9F	16	BB
07	A6	17	BC
08	A7	18	BD
09	AB	19	BE
0A	AC	1A	BF
0B	AD	1B	CB
0C	AE	1C	CD
0D	AF	1D	CE
0E	B2	1E	CF
0F	B3	1F	D3
20	D6	30	ED
21	D7	31	EE
22	D9	32	EF
23	DA	33	F2
24	DB	34	F3
25	DC	35	F4
26	DD	36	F5
27	DE	37	F6
28	DF	38	F7
29	E5	39	F9
2A	E6	3A	FA
2B	E7	3B	FB
2C	E9	3C	FC
2D	EA	3D	FD
2E	EB	3E	FE
2F	EC	3F	FF

NOTE: AA and D5 are reserved codes.

$$00110011 = \$33$$
$$00001100 = \$0C$$
$$00111110 = \$3E$$
$$00100000 = \$20$$

The six values are now mapped to eight six-bit values having the same bit sequence. To get byte values that can be put on disk, use a table in RWTS called a Write Translate Table and lookup each byte to get: $EF, $AE, $BC, $AE, $F2, $AE, $FE, and $D6. These are the sixbit encoded values of the ASCII string "HELLO".

To transform raw data back to original values, the process is reversed. First, use a table called the Read Translate Table in RWTS to lookup the six-bit value for each encoded byte. Then regroup the six least significant bits from each six-bit value together to form eight-bit bytes. From 242 bytes of sixbit you will get 256 bytes of eight-bit data bytes.

Each bit is read or written to the disk in four microseconds. With the disk rotating at 300 revolutions per minute, this provides a capacity of about 50,000 bits per track. Each eight-bit byte can be accessed in 32 microseconds in this system. Although the bytes are used in the Apple with a crystal-controlled clock, remember that the bits on disk are clocked from the recorded pulses on the disk. For this reason, constant synchronization must be made.

The first method to ensure synchronization is to provide clock pulses interleaved with the data bits on the disk. Each bit is recorded as a cell consisting of one clock pulse followed by the data bit. A one bit will be represented by two pulses, clock and data. A zero bit will be represented by one pulse, clock only. Each of these bit cells is four microseconds: two microseconds for the clock pulse and two microseconds during which a bit of data is defined as pulse/no pulse. The data stream then is interleaved with clock pulses that the hardware must separate in order to read.

To separate clock and data, the hardware needs two conditions met. First, there can be no more than two zero bits consecutive. Second, all bytes must start with a one bit as bit 7.

When it starts reading a track, there is no way of knowing where it is. The first bits are arbitrarily shifted into a byte latch even though the odds are one in eight of being the start of a byte. However, if it looks for sync bytes, then it checks to see that all bits are on. If any bit is off, then it tries again with the next bit. After several tries, this synchronizes the byte reading to give true bytes.

The situation is similar when writing. A byte must be supplied each 32 clock cycles. If a longer time is taken, then zero bits are written. The one case where this is done deliberately is in the routine that creates the sync bytes. It writes an $FF, then wastes enough time for

two zero bits to get written before writing the next $FF. But normal reading and writing of bytes is done in 32-cycle loops. See Fig. 7-6.

Within the bytes, the hardware can stay synchronized with clock pulses interleaved with data bits. For longer times, the sync bytes must align the hardware to read and write bytes that agree with the sector fields.

Fig. 7-6. Example of disk data byte (value $B7).

7.2 PROTOCOLS

7.2.1 Command

The protocol you already know is the highest level you can use. You use these DOS commands in one of three ways. First, you can handle entire files at a time using BLOAD, BRUN, BSAVE, CHAIN, DELETE, EXEC, LOAD, LOCK, RENAME, RUN, SAVE, UN-LOCK, and VERIFY. Second, you can manipulate files in detail by working within their structures, their records and fields. The commands you use to manipulate text files this way are: APPEND, CLOSE, OPEN, POSITION, READ, and WRITE. And third, you have some universal *housekeeping* commands: CATALOG, FP, IN#, INIT, INT, MAXFILES, MON, NOMON, and PR#. All three types of commands are summarized in Table 7-10 which also lists their syntax.

Each command is more or less independent of the others in the sense that you can use them in turn without much more than a rough idea of what they do. The exception is the second group which is the file structure commands.

In this section, you can find each command described in alphabetic order; the file manipulation commands are described at the end in a separate group.

BLOAD f {,Aa} {,Ss} {,Dd} {,Vv}

Table 7-10. DOS Command Syntax

Command	Syntax
APPEND	f, Ss, Dd, Vv, Lj
BLOAD	f, Aa, Ss, Dd, Vv
BRUN	f, Aa, Ss, Dd, Vv
BSAVE	f, Aa, Lj, Ss, Dd, Vv
CATALOG	Ss, Dd
CHAIN	f, Ss, Dd, Vv
CLOSE	f
DELETE	f, Ss, Dd, Vv
EXEC	f, Rr, Ss, Dd, Vv
FP	Ss, Dd, Vv
IN#	s
INIT	f, Ss, Dd, Vv
INT	
LOAD	f, Ss, Dd, Vv
LOCK	f, Ss, Dd, Vv
MAXFILES	n
MON	C, I, O
NOMON	C, I, O
OPEN	f, Lj, Ss, Dd, Vv
POSITION	f, Rr
PR#	s
READ	f, Rr, Bb
RENAME	f, g, Ss, Dd, Vv
RUN	f, Ss, Dd, Vv
SAVE	f, Ss, Dd, Vv
UNLOCK	f, Ss, Dd, Vv
VERIFY	f, Ss, Dd, Vv
WRITE	f, Rr, Bb

f, g = file names
s = slot: 1 to 7
d = drive: 1 or 2
v = volume: 0 to 254
r = record: 0 to 32767
j = record size: 1 to 32767
b = byte number: 0 to j
a = start/load address (*note:* hex number prefixed $)

Loads a binary file. You can override the start address of the file with the A option; for example,

<div align="center">

BLOAD CHARACTERS,A$4000

</div>

loads the file at $4000 regardless of the file's own start address parameter. You can't alter the length, however.

What you can do is find the start address and length of the BLOADed file. Call this routine immediately after the BLOAD, before any other DOS command is used:

```
        JSR   $03DF      get File Manager parameters
        STY   $40
        STA   $41
        JSR   CROUT      output a CR
        LDY   #$08       point to address parameter
        JSR   HEXOUT     and output it.
        JSR   PRBLNK     three spaces out
        LDA   #$06       point to length parameter
        JSR   HEXOUT     and output it.
        JMP   CROUT      output a CR and return
HEXOUT  LDA   ($40),Y    low byte
        TAX
        INY
        LDA   ($40),Y    high byte
        JMP   PRNTAX     print them and return
```

The routine uses Monitor routines from Chapter Two.

BRUN f {,Aa} {,Ss} {,Dd} {,Vv}

Loads a binary file, just like the BLOAD command. When finished with the load, it transfers control to the new loaded file instead of returning. The file must have program code at its start address even if it is only the three bytes of a JMP instruction. The load address is optional, just like the BLOAD command.

BSAVE f,Aa,Ll {,Ss} {,Dd} {,Vv}

Saves a binary file. If the file is new, a directory entry is created. The file name, start address, and length are mandatory; the slot, drive, and volume are optional. The start address you give must be the first location of the binary image in memory and becomes the default load address. Remember, if you are making a binary HELLO file, you must alter the DOS as described earlier; the patch is

9E42: 34

made to DOS before the INIT command.

CATALOG {,Ss} {,Dd}

Displays most of each directory entry. As mentioned earlier, beware of hidden characters, or the CATALOG won't reveal the file names. Also, files using more than 256 sectors will show only the low byte of the file size: a number from 000 to 255 even though the directory size is larger.

CHAIN f {,Ss} {,Dd} {,Vv}

This works like a program RUN command but without clearing the variables of the current program. The idea is to link program segments together with common data when a program can't fit into memory. The program is broken up into smaller programs, or *segments,* as they are often called, and each one can (theoretically) CHAIN to any of the others without destroying the current variables' data.

In practice, it works with Integer BASIC which keeps its strings in the variables and the variables in a separate memory area. With Applesoft BASIC, chaining has bugs due to the keeping of variables in the same area as the program. Apple IIe has an improved CHAIN routine on their System Disks that can be run as a binary routine to do CHAINing.

A more satisfactory solution would be to BSAVE your data between programs. Use the information in Chapter Four. Or, you can write the data out to a data file to be read by the other programs in your system. Unfortunately, these solutions require program planning which is usually lacking in situations where CHAINing is asked for. Try and rewrite the mess as a designed, disk-based system instead.

DELETE f {,Ss} {,Dd} {,Vv}

Removes the file from disk by zeroing the track number of the link in its directory entry. The old track number is put into the last byte of the file name. The sectors used by the file are freed by turning on the bits in the Track Bit Map in the VTOC. Immediately after a DELETE, the information is not lost even though the directory and storage spaces have been released.

To *un-delete* a file, you must get to it in time with either an UNDE-LETE or a DISK ZAP utility, otherwise the space will be reused, wiping out the old file data. Find the file name by searching the directory in 17/15, 17/14, etc. When found, restore the track number from the last byte of the file name. Blank ($A0) that last byte. Then, write down the location of the TSL. Replace the directory sector and read the TSL. Copy down all the TSL entries. Read in the VTOC from 17/0 and turn off the bits in the Track Bit Map corresponding to your list. Save the rebuilt VTOC. The file should be intact if you were careful, but try and copy it to a new disk just in case. In fact, you should be able to rebuild the disk on a copy with FID for backup.

EXEC f {,Rr} {,Ss} {,Dd} {,Vv}

Generates keyboard commands by reading a data file. You can keep common command sequences in a text file, then EXEC that file instead of re-typing the sequence each time. Use a text file line editor like the one in Apple's Toolkit. Or use the commands to OPEN, WRITE, and CLOSE to create it from a program. The editor method is easiest to write and maintain.

An easy way to edit BASIC program files is by capturing them to a text file and using the features of a text line editor to maintain it. The Capture Algorithm

```
0 PRINTCHR$(4)"OPEN CFILE":PRINTCHR$(4)"WRITE CFILE":
LIST 10,32767:PRINT CHR$(4)"CLOSE":END
```

in a file called CAPTURE can be EXECuted by

```
EXEC CAPTURE
RUN
```

with the program in memory. The text file called CFILE is created which you can edit. Then, EXEC CFILE to get it back as a program in BASIC.

FP {,Ss} {,Dd} {,Vv}

If Applesoft is not currently in memory, then it will be bank-switched in an attempt to find it. If unsuccessful, DOS then tries to load and run a program in Integer called APPLESOFT from the current disk. If this fails you get a FILE NOT FOUND error.

This FP command may be implied at boot time. If the HELLO program is in Applesoft and the Apple has only Integer BASIC resident, then an FP implied command is executed. For this reason, Apple DOS Master System disks have an Integer program called APPLESOFT that loads FPBASIC into memory. When APPLESOFT is ENDed, the HELLO program is run, completing the boot.

Regardless of how it is called, the FP command finishes by cold starting Applesoft.

IN# s

This command preempts the Monitor command keyword in BASIC of the same name. It sets the input hook within DOS according to the slot number, s, as $Cn00. If s is zero, then the keyboard input is used. See Chapter Six for details on the way the hooks work.

INIT f {,Ss} {,Dd} {,Vv}

Creates slave DOS disks. You must have your greeting program in memory at the time (f) and name it, preferably as HELLO. Just before calling, you can make the patches described earlier for:

9E42: 34 for Binary HELLO
AE34: 60 to remove CATALOG pauses
BFD3: EA EA EA prevent forced BASIC loads

You get a slave disk with DOS, a VTOC, a Directory, and your HELLO program. See the layout part of the Disk Map section in this chapter for more details.

You have to BRUN MASTER CREATE to convert slave disks made with INIT to master disks. A master disk will bootstrap in old Apples that have only 16K or 32K of RAM.

INT

See also the FP command. The INT command causes DOS to switch languages to Integer BASIC if it is not already current. If Integer BASIC can't be found then you get a LANGUAGE NOT AVAILABLE error.

Once Integer BASIC is found, it is cold started, and any previous program is lost.

LOAD f{,Ss} {,Dd} {,Vv}

You use this to load a BASIC program. If the BASIC of the program isn't in current memory, then memory is switched to try and find it. If Applesoft is not found, DOS will try to run APPLESOFT in Integer BASIC from disk first.

When loaded, all pointers are reset for the language used. HIMEM and LOMEM in the case of Integer and TXTTAB and MEMSIZ in the case of Applesoft are unchanged and used to determine the load address. The length comes from the first two bytes of the program file as described in the Disk Map section.

LOCK f {,Ss} {,Dd} {,Vv}

Using this one flags the file type (bit 7) in the directory entry as protected. You see it as an asterisk − "*" − in the CATALOG display. When LOCKed, a file cannot be altered by a file managing command, either from DOS or from the File Manager directly. It won't protect the file against INIT, which will wipe out the entire disk; only a write protect tab over the notch on the side of the disk will protect it from INIT. Same goes for the RWTS writes: they can change any sector on disk regardless of what may be in one of the directory sectors.

Your best use of LOCK is in file management. For example, LOCKing a transaction file like a Cash Journal is a great way to show that the file has been posted to an accounts file(s).

MAXFILES n

A very tricky command. You can change the number of file buffers from three to another number with this one. It is known more for its restrictions which you must observe religiously:

Thou Shalt Not . . . set MAXFILES 0. Ordinary-looking commands like LOAD need a file buffer.

Thou Shalt Not . . . use MAXFILES in an EXEC file. The MAXFILES command clears all file buffers, including your EXEC buffer.

Thou Shalt Not . . . use from within Applesoft after you assign strings; they will be clobbered by an increase in the buffers space.

Thou Shalt Not . . . use from within Integer for verily thy program will be clobbered!

There is an Assembler call you can make to MAXFILES to avoid contentions with BASICs, $A251, for you to set things up from a binary HELLO program first.

MON C ,I ,O

Controls the DOS echo to the video display. As commands, inputs, and outputs are recognized at the DOS hook routine, each may be echoed to the video display at COUT1. This command lets you turn

the echoes ON, individually. The default for DOS 3.3 is for all echoes to be OFF. Using MON C, for instance, lets you see all commands that a program issues with a ctrl/D prefix. The MON I lets you see inputs to DOS from any device currently in its input hook. The MON O lets you see any outputs to DOS from your program as DOS writes them.

Use MON and NOMON when debugging DOS commands, reads, and writes to see what is getting through.

NOMOM C ,I ,O

Similar to MON, but turns OFF one or more of the three echoes. NOMON C,I,O is the normal state for DOS 3.3.

PR# s

This command preempts the Monitor command keyword in BASIC of the same name. It sets the output hook within DOS according to the slot number, s, as $Cs00. If s is zero, then the video output is used. See Chapter Six for details on the way the hooks work.

RENAME f,g {,Ss} {,Dd} {,Vv}

Renames a file in its directory entry. Looks up the old file name, f, in the directory, then replaces it with the new file name, g.

RUN {f} {,Ss} {,Dd} {,Vv}

If a file name is given, does a LOAD of that program file, then a BASIC RUN command. If no file name is given, then just a BASIC RUN for the program currently in memory is done. See the LOAD command for more details.

SAVE f {,Ss} {,Dd} {,Vv}

If program file of the type of BASIC currently active is not on file, a new directory entry is created. The BASIC program text is written to the file. If the file exists as another type, a FILE MISMATCH error results.

UNLOCK f {,Ss} {,Dd} {,Vv}

Using this one flags the file type by clearing bit 7 in the directory entry. When the file is UNLOCKed, you can write to it with any of the commands: DELETE, WRITE, and SAVE. See the LOCK command.

VERIFY f {,Ss} {,Dd} {,Vv}

This does a load of all the sectors of the file to verify that the file is readable. You can VERIFY any type of file.

APPEND f {,Ss} {,Dd} {,Vv}
OPEN f {,Ss} {,Dd} {,Vv}
CLOSE {f}
POSITION f {,Rr}
READ f {,Bb}
WRITE f {,Bb}

This is the syntax for sequential files access. Each record ends with a CR character ($8D) and has any length. The OPEN puts the position pointer at the beginning of the file while the APPEND opens the file and points to the end of the file instead. The CLOSE will close all files if no file name is given.

The POSITION works two ways. If you give it a record number, it searches forward that many records to a new file position. So, R is the key for a *relative* record number, not an absolute one. For instance, if you were at Record 34 and you gave DOS

POSITION FILEX, R4

the pointer in the file manager would advance to Record 38. If you don't use the R option, the pointer is reset to the beginning of the entire file. You do this when you want another pass through the file.

After an OPEN or a POSITION to set the pointer to zero, you can read and write the current record. Each READ or WRITE enables the DOS to trap input or output at the hooks. The B option lets you point absolutely anywhere in the file. For example,

```
1000  R$ = " ": REM GET A RECORD
1010  PRINT D$"READ"F$"",B"STR$(BN)
1020  GET A$:IF A$ = CHR$(13) THEN 1050
1030  R$ = R$ + A$: BN = BN + 1
1040  GOTO 1010
1050  BN = BN + 1: RETURN
```

where R$ is returned as the record, D$ is CHR$(4), F$ is the file name, and BN is the byte number: zero for start of file. While BN is incremented sequentially here, you can *randomly* access a sequential file, one byte at a time, by setting BN anywhere in the file you wish.

OPEN f, Lj {,Ss} {,Dd} {,Vv}
CLOSE {f}
POSITION f {,Rr}
WRITE f {,Rr} {,Bb}
READ f {,Rr} {,Bb}

This is the syntax for random access. Like sequential records, each ends with a CR character, but unlike them, all records *must* have the same length. Otherwise, the files are the same. To open, you must declare the record size as j. The record size includes *all* delimiters, especially the CR.

The POSITION works the same way as in the sequential access case; it is rarely used in random access.

Normally, READs and WRITEs use the R option without the B. This selects the record number, Record Zero being the first one. The B option points to the byte relative to the record. For instance,

PRINT D$"READ"F$"R0,B0"

or

<div align="center">

PRINT D$"READ"F$"R0"

</div>

are equivalent and point to the beginning of the file. Similarly,

<div align="center">

PRINT D$"READ"F$"R5,B1"

</div>

points to the second byte of the sixth record. This is the result of the calculation:

$$\text{pointer} = b + j*r$$

7.2.2 File Manager

Normally you use the File Manager with calls from DOS: commands like OPEN or READ. You can only OPEN and handle records in text files but by calling the File Manager directly, you can work with other file types as well. Not only will File Manager calls let you write manipulation programs like FID but you can develop your own access methods for easier file handling. The call to File Manager is provided in Page Three; you can make a simple JSR there.

The heart of your calls to the File Manager involves passing it your parameters by reference in the registers:

```
LDA  #<PARMS      address high
LDY  #>PARMS      address low
JSR  $3D6         File Manager
BCS  ERROR
```

In case of errors, the error code is in the eleventh byte (Byte $0A) of the parameter block, PARMS. If you kept a Page Zero pointer to the parameters, you could use the call:

```
LDA  ZPARM + -
     1            address high
LDY  ZPARM        address low
JSR  $3D6         File Manager
BCS  ERROR
```

Before each call, you must be sure all parameters needed are in the block you reference. Use Tables 7-11, 7-12, 7-13, and 7-14.

Table 7-11a. File Manager Syntax

Operation	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
OPEN	01	–	j	j	v	d	s	f	n	n	e	–	b	b	t	t	a	a
CLOSE	02	–	–	–	–	–	–	–	–	–	e	–	b	b	t	t	a	a
READ	03	m	r	r	y	y	l	l	k	k	e	–	b	b	t	t	a	a
WRITE	04	m	r	r	y	y	l	l	k	k	e	–	b	b	t	t	a	a
DELETE	05	–	–	–	v	d	s	–	n	n	e	–	b	b	t	t	–	–
CATALOG	06	–	–	–	–	d	s	–	–	–	e	–	b	b	–	–	–	–
LOCK	07	–	–	–	v	d	s	–	n	n	e	–	b	b	t	t	–	–
UNLOCK	08	–	–	–	v	d	s	–	n	n	e	–	b	b	t	t	–	–
RENAME	09	–	g	g	v	d	s	–	n	n	e	–	b	b	t	t	–	–
POSITION	0A	–	r	r	y	y	–	–	–	–	e	–	b	b	–	–	–	–
INIT	0B	p	–	–	v	d	s	–	–	–	e	–	b	b	–	–	–	–
VERIFY	0C	–	–	–	v	d	s	–	n	n	e	–	b	b	t	t	a	a

Table 7-11b. File Manager Syntax

aa = Address of file data buffer

bb = Address of file status buffer

d = Drive number

e = Error return code

f = File type code

gg = Address of new file name

jj = Record size(random) or $0000(sequential)

kk = Data transfer value(SINGLE) or address(RANGE)

11 = Data transfer length(RANGE)

m = Mode: SINGLE, RANGE, POSITION/SINGLE, POSITION/RANGE

nn = Address of file name

p = Page of DOS start, usually $9D

rr = Record number(random) or $0000(sequential)

s = Slot number

tt = Address of file TSL buffer

v = Volume number

yy = $0000(random) or byte offset(sequential)

Table 7-12. Read/Write Modes

Byte $01:	01 is SINGLE. One byte of data is read/written as the value in Byte $08.
Byte $01:	02 is RANGE. A range of bytes is read/written so its address is in Bytes $08.09 and its length is in Bytes $06.07.
Byte $01:	03 is POSITION/SINGLE. The current position is set to the record number in Bytes $02.03 offset by Bytes $04.05. Then one byte is read/written as the value in Byte $08.
Byte $01:	04 is POSITION/RANGE. The current position is set to the record number in Bytes $02.03 offset by Bytes $04.05. Then a range of bytes is read/written so its address is in Bytes $08.09 and its length is in Bytes $06.07.

Table 7-13. Error Return Codes

00	No error, C-flag clear
01	Unused
02	Illegal op code, Byte $00
03	Illegal mode, Byte $01
04	Write protect error
05	End of file
06	File not found
07	Volume mismatch
08	Disk I/O error
09	Disk full
0A	File locked

Table 7-14. File Type Codes

Code	File
00	Text
01	Integer
02	Applesoft
04	Binary
08	S-type
10	Relocatable
20	A-type
40	B-type

The main difference between using the File Manager through DOS's command interpreter and using it directly is in the managing of file buffers. Each file you want to access must have four buffers of various sizes for the File Manager to use. Two of these buffers are 256 bytes each: one page for a data sector and one page for a TSL sector. Then a 30-byte buffer must hold the file name and a 45-byte buffer be given to the File Manager to keep the status of the file. When it is off working with other files, this file must be remembered so that the next time you reference it, File Manager can recall its status (see Fig. 7-7). It sets up the status when you OPEN the file. So, to provide these buffers, one way is to reserve space in your program:

DATA1	DS	256	file 1 data
TSL1	DS	256	file1 TSL
STAT1	DS	45	file 1 status
NAME1	DS	30	file 1 name

And, similar declarations for any other files.

Note: See explanation of Fig. 7-7 below.

Fig. 7-7. How the file manager works.

HOW THE FILE MANAGER WORKS

Any one of your routines, or the DOS command interpreter, can call the FILE MANAGER at its Page Three JMP location at $3D6. A list of parameters must be given with its address in the Y-reg(lo) and A-reg(hi) at the time of call. This PARMS list tells FILE MANAGER what to do and gives it the pointers to the file buffer: to the DATA buffer, to the TSL buffer, and to the status location in the MISCINFO.

The FILE MANAGER reads the parameters to find out what to do. The parameters tell it where the DATA buffer, the TSL buffer, and the STATUS buffer are to be found. It uses the STATUS buffer to store its file status between calls. It accesses the disk and keeps buffers for the VTOC, the DIRECTORY, the TSL, the DATA, and for each file.

The FILE MANAGER reads and writes to the disks, one sector at a time. Interpreting its op code, it searches the disk for information, changes it, and returns the updated sectors to disk. It uses the VTOC and DIRECTORY to manage the files that it manipulates with its buffers. Each disk access is made using a Read/Write Track/Sector (RWTS) routine, described later on in this chapter.

Here's how to OPEN a file directly. With the space declared for buffers and another block of 17 bytes for your parameters, set each parameter byte:

Byte	$00	$01, the op code for OPEN
Bytes	$02.03	length of fixed record; zero otherwise
Byte	$04	volume, zero for any volume
Byte	$05	drive: $01 or $02
Byte	$06	slot: usually $06
Byte	$07	file type, use Table 7.
Bytes	$08.09	address of filename:buffer
Bytes	$0C.0D	address of status buffer
Bytes	$0E.0F	address of TSL sector buffer
Bytes	$10.11	address of data sector buffer

If you are opening a new file, then byte $07 must be set to your file type. Before calling the File Manager, set the X-reg to zero: this will

tell it to allocate a new directory entry for you. And, you must have a file name in the file-name buffer for any kind of OPEN. If you are re-OPENing an existing file, then byte $07 need not be set. You should set the X-reg to a nonzero value to inhibit the creation of a directory entry, just in case the file isn't found.

After calling the File Manager to OPEN your file, you can get the type of the previously created file you are re-OPENing from byte $07: File Manager returns it to you. If you have an error, interpret Byte $0A, the error code.

Once the file is OPENed, you should immediately do a POSITION call to point File Manager to the first byte. Use the same parameters as OPEN except for:

Byte $00 $0A, OP code for POSITION
Bytes $02.05   all zeros

Such a call is called a REWIND by analogy to the rewinding of a tape to the beginning of a file.

Similarly, you can CLOSE the file when you are finished with it. All files that were OPENed must be CLOSEd:

Byte $00 $02, the CLOSE OP code
Bytes $0C.0D   address of status buffer
Bytes $0E.0F   address of TSL buffer
Bytes $10.11   address of data sector buffer

Throughout the time a file is OPEN, you must maintain its buffer addresses for future calls, up to and including the CLOSE.

When DOS uses the File Manager, it assigns its buffers from the three at $9600.9CF8 — see the DOS memory map in Chapter Two. You can get a closer look at these three buffers in Fig. 7-8.

Each buffer is 595 ($253) bytes in size. There are two full page buffers in each for the data sector and the TSL sector of its file. The remaining 83 ($53) bytes are called MISCINFO for miscellaneous information and contain the file name and status buffers. And, MISCINFO has some pointers.

Looking at the one DOS buffer a little closer (Fig. 7-9) you can see what the MISCINFO is all about. The two small buffers for the file status (45 bytes) and the file name (30 bytes) complete the File Mana-

```
                         DOS
$9CF8 ┌─────────────────────────────┐        ▲
      │        MISC  INFO  1         │        │
      ├─────────────────────────────┤        │
      │         T/S LIST 1           │        │
      │                             │       $253
      ├─────────────────────────────┤        │
      │          DATA 1             │        │
      │                             │        ▼
$9AA6 ├─────────────────────────────┤        ▲
      │        MISC  INFO  2         │        │
      ├─────────────────────────────┤        │
      │         T/S LIST 2           │        │
      │                             │       $253
      ├─────────────────────────────┤        │
      │          DATA 2             │        │
      │                             │        ▼
$9853 ├─────────────────────────────┤        ▲
      │        MISC  INFO  3         │        │
      ├─────────────────────────────┤        │
      │         T/S LIST 3           │        │
      │                             │       $253
      ├─────────────────────────────┤        │
      │          DATA 3             │        │
      │                             │        ▼
$9600 ├─────────────────────────────┤
      │           USER              │
      │          PROGRAM            │
      └─────────────────────────────┘
```

Fig. 7-8. Three DOS buffers.

ger's needs for a single file's buffers. Then there are the four pointers at the highest addresses of MISCINFO. Three of these point to the status, TSL, and data buffers for this file. The fourth pointer is a link to the next buffer — it points to the file name of the next file buffer.

The buffers are all linked together in a chain. The first buffer is pointed to from $9D00.9D01 which is the beginning of DOS. It in turn points to the file name buffer in the next file buffer. This second buffer points in turn to the third. The third one is usually last, so it has zeros in its link pointer.

The number of buffers and the pointers in each are setup by the MAXFILES command. From assembly language you set the number of buffers at $AA57, the first pointer value at $9D00.9D01, then JSR the MAXFILES command at $A251. The first pointer must be at least 38 bytes below the highest location occupied by the buffers. It points

Fig. 7-9. One DOS buffer.

to the file name which occupies 30 bytes and is followed by four pointers in eight more bytes. To have three buffers that begin on a page boundary at $9600, the pointer in $9D00.9D01 is set to $9C

Regardless of whether you alter MAXFILES, DOS will search through the buffers beginning with the pointer at $9D00 and ending when the next file pointer is zero.

Instead of making your own file buffers, you can use the regular DOS buffers. Here's how.

There are two routines that will search the buffers. One is at $A764; it returns the address of the buffer in Page Zero at $44.45 (with $45 zero if none are free). Another uses two calls that you can make in your own search routine:

```
GETBUF  JSR   $A792      point to first
GETB1   LDY   #0
        LDA   ($42),Y    zero filename?
```

```
BEQ   FOUND        yes. . . buffer free
JSR   $A79A        no. . . point to next
BEQ   NONE         end of chain?
BNE   GETB1        no . . . loop
```

The second BEQ tests for the end of the chain of buffers: the pointer in $42.43 is zero then. Of the two exits to this routine, FOUND has the address of the free buffer in $42.43. This address is for the file name, so you can load it with yours to claim the buffer for yourself.

With the pointer in $42.43, copy your 32 byte file name into the buffer. This name should have blanks to fill up the entire 32 bytes; don't leave garbage in with your file name.

Now you can start building the parameter list for your file. Put its address in $40.41 and you can call $AF08 to set the addresses of the buffers in it. Then set your parameters in bytes $00.07 as before. The rest of the call sequence is just like the case of using your own buffers.

When you close the file, you must release the buffer back to DOS. This is easy: just zero the first byte of the file name. When the file is OPENed, you should save the file name pointer from $42.43 because the copy in the parameter list won't remain. At CLOSE then, you can recall it:

```
LDA   FILB1 + 1     get file buffer
STA   $43           pointer and put
LDA   FILB1         into $42.43
STA   $42
LDA   #0            zero first byte
TAY                 of file buffer
STA   ($42),Y       filename
```

immediately after you CLOSEd the file.

To summarize, here is what you need to open, close, and rewind files. To open a file, attach a buffer with a GETBUF type routine, copy the file name into the buffer, save the buffer pointer, copy the buffer pointers into your parameter list, and call the File Manager to OPEN the file on disk. Normal OPENs should be followed with a RE-WIND call that zeros the bytes $02.05 of the file's parameter list. Then, call File Manager to POSITION its pointer. When you close a file, call File Manager with its parameters. Include a buffer release in

your CLOSE routine, immediately after the CLOSE call to zero the first byte of the file name buffer.

You can access OPENed files with the READ and WRITE op codes. When you used DOS commands to do this, with text files, you could choose either random or sequential file access. To do the same thing with the File Manager yourself, you need to know about the *read/write mode* — a File Manager parameter — in Byte $01. By selecting the value of the mode, you can access your file either randomly or sequentially.

To read a Data file sequentially, set

Byte	$00:	$03, the READ opcode
Byte	$01:	$01, the SINGLE mode
Bytes	$02.03:	record number, reset zero by REWIND
Bytes	$04.05:	byte offset, reset zero by REWIND
Bytes	$0C.11:	addresses as set by OPEN sequence

The single byte is returned to you in Byte $08. If the *end of file* was found, the return code in Byte $0A will be $05; you must test each of your READs for this. The byte offset in $04.05 will be bumped by one count for each READ or WRITE that you make. You can buffer your reads into records by testing the text character for $8D which is the carriage return character that separates records.

To write a sequential data file, you can first search the file for the end-of-file return code in $0A with the sequential read procedure. This will APPEND your writes to the end of the current file. Assuming you have an entire record to write from the input buffer at Page Two, set

Byte	$00:	$04, the WRITE op code
Byte	$01:	$02, the RANGE mode
Bytes	$02.05:	pointers, reset by REWIND routine
Bytes	$06.07:	number of bytes to write, less 1
Bytes	$08.09:	address of bytes to write
Bytes	$0C.11:	addresses as set by OPEN sequence

Using the RANGE mode avoids having to write a loop when you want to read an entire buffer. Be careful of the number of bytes in bytes $06.07, however. You must set it to the number of bytes for READ op codes but to the number of bytes *less one* for the WRITE op codes.

The number of bytes includes the carriage return character $8D, so you must be sure that it is there. For writing sequential records, set the number of bytes in bytes $06.07 to the record length that you have without a carriage return; make sure the carriage return is appended to the record as it will be written. For example, a record of fifty bytes must be written with fifty-one bytes, fifty characters and a $8D. A fifty ($0032) must be put in bytes $06.07. The $8D is in $0232, the fifty-first character of the record.

Random access by calling the File Manager is just as easy. When OPENed, you will have declared the number of bytes in each record, including the return character $8D, in bytes $02.03. So, you can access the file to read a record by:

Byte	$00:	$03, the READ op code
Byte	$01:	$04, the POSITION/RANGE mode
Bytes	$02.03:	Record Number: 0. . .$7FFF
Bytes	$04.05:	$0000, the byte offset
Bytes	$06.07	record length
Bytes	$08.09	record buffer address, e.g., $0200
Bytes	$0C.11	addresses as set by OPEN sequence

The POSITION/RANGE mode implies a POSITION followed by a READ. This way, you only need one File Manager call to point to the record given in bytes $02.03. Since you have fixed length records, the position is easily calculated and the record is read to your given buffer. The length in bytes $06.07 is used to set the size of the record read while the position is calculated from the record length you specified back in the file's OPEN sequence. The two should be the same in most applications, but remember that File Manager uses them for these two different purposes.

Writing random is much the same. You use the same POSITION/RANGE mode but with the WRITE op code this time:

Byte	$00:	$04, the WRITE op code
Byte	$01:	$04, the POSITION/RANGE mode
Bytes	$02.03:	Record Number
Bytes	$04.05:	$0000, the byte offset
Bytes	$06.07:	record length less one
Bytes	$08.09	record buffer address, e.g., $0200
Bytes	$0C.11	addresses as set by the OPEN sequence

Notice that the value you use in bytes $06.07 is one less than the true number of bytes written. This is the case for all WRITEs, as explained before with sequential writes.

With either READ or WRITE, whenever you want random access to a record, put the record number in bytes $02.03 and *always* zero bytes $04.05 before each call. After the call, the byte offset is advanced by the File Manager to point to the next byte. Occasionally this feature may be used, but for most calls, you'll want to zero it with each call.

With the various op codes and modes directly available you can access files any way you want. In addition to the text file type, you can access files of other types using these File Manager calls. You can OPEN, READ, WRITE, and CLOSE program files or binary files in just the same way. Just keep the format of the file in mind and read the parameters when you OPEN the file.

In the case of program files — A and I types — you must first read in the beginning two bytes and store them in RAM as the length. Then you can load the file either at its normal BASIC load point or any- where else you want using the length you have as the bytes $06.07 for the READ call. You don't have return characters, $8D, to separate records in program files; refer to the description of the BASIC text in either Chapter Four or Five.

For binary file types, you can first read the address and length as the first four bytes of the file. You then can read the entire file as one range using the address and length parameters. Again, you can read it in anywhere in RAM you wish; in part or in whole.

Programs you write to read other programs into memory this way are called *loaders*. With File Manager, you can make your own cus- tom loader, perhaps as a binary HELLO program. For some applica- tions, you may want to create program or binary files. You must write two bytes as the length of your program file at the beginning. And, write the start address and length of a binary file you create as its first four bytes just like DOS does.

The File Manager has all the file handling operations in addition to the OPEN, CLOSE, READ, and WRITE. See the syntax in Fig. 7-7 for the parameters that each requires. Most of these parameters are the same ones you use when giving DOS the corresponding com- mands. Buffers are needed. You always get the carry flag to warn of error return code, and you get the code in byte $0A, just like the ones already described.

7.2.3 Read/Write Track/Sector

The Read/Write Track/Sector routines let you access the disks directly. You can read and write to individual sectors anywhere on a disk, you can format disks, and you can position the read/write head to any track for special access methods. Most usage of RWTS routines are handled for you from Page Three call sequences, so you can write programs that are independent of the DOS memory locations.

Like the File Manager, RWTS has two Page Three call sequences. One fetches the address of the parameters; the other calls the RWTS routine itself. When you write

```
JSR   $03E3      get IOB
STY   $48
STA   $49
```

you get the address of the parameter list called the *Input-Output Block* or IOB. Here it is put into $48.49 which is the same location in Page Zero that RWTS uses itself. By varying the Y-register from $00 to $10, you can set the parameters as you need for your call. Then,

```
JSR   $03E3      get IOB
JSR   $03D9      call RWTS
BCS   ERROR
```

performs the operation you specified: READ, WRITE, POSITION, or FORMAT. Like File Manager, the RWTS returns the carry flag set if it found an error. See Table 7-15 for a full summary of the IOB parameters.

When you finish using RWTS and before you return or use the Monitor again, you must zero location $48 in Page Zero. There is a conflicting usage, and the Monitor thinks a nonzero value there is a saved P-reg. You can't avoid the conflict because RWTS itself uses $48.49 as the IOB pointer.

You can of course use your own IOB instead of the one in DOS. If you do, then you must set the Y-reg to the low byte of its address and the A-reg to the high byte immediately before the call to RWTS:

Table 7-15. Input/Output Block

Byte	Contents
00	$01, always
01	Slot*16. Example $60
02	Drive: $01 or $02
03	Volume: $01 to $FE. $00 is wild.
04	Track: $00 to $22
05	Sector: $00 to $0F
06.07	Address of Device Characteristic Table
08.09	Address of Data Buffer
0A	Unused
0B	Bytes, partial sector. Full sector $00
0C	Command: 00 Position
	01 Read
	02 Write
	04 Format
0D	Return: 08 Initialization
	10 Write protect
	20 Volume mismatch
	40 Drive I/O error
0E	Volume of previous access
0F	Slot of previous access *16
10	Drive of previous access

```
LDA   #<MYIOB      get IOB
LDY   #>MYIOB
JSR   $03D9        call RWTS
BCS   ERROR
```

This is probably what you did if you used File Manager with several files, but most calls to RWTS can be handled easier if you only use the built-in IOB. This custom IOB call is only for unusual situations; use the JSR $03E3 method instead.

The thing you change for different RWTS calls is the Data Buffer address at bytes $08.09. You can keep various kinds of data in different buffers by changing this parameter. File Manager, for instance, varies this address to have separate buffers for its current VTOC, Directory sector, and up to three sets of data and TSL buffers. They all

use the same IOB; only the buffer pointer need be changed to redirect the reads and writes.

Here's how to read a sector from the disk. First, get the IOB address into $48.49 as described above. Then set the parameters as follows:

Byte	$01:	slot number times 16, normally $60
Byte	$02:	drive number, $01 or $02
Byte	$03:	volume, $00 to read any volume
Byte	$04:	track $00. . .22
Byte	$05:	sector $00. . .0F
Byte	$08.09:	Buffer address. This is where your read sector will be placed.
Byte	$0B:	Normally $00, you can set byte count for a partial sector instead.
Byte	$0C:	command code, $01 for READ

If you are using your own IOB instead of the system's, you must also set:

Byte	$00:	to $01
Byte	$06.07:	address of DCT, a Device Control Table of four bytes: 00 01 D8 EF for Disk II.
Byte	$0E:	volume number last accessed
Byte	$0F:	slot number times 16 last accessed
Byte	$10:	drive number last accessed

Upon return from RWTS, you test for error with the C-flag. If set, you can find the error in byte $0D according to Table 7-15.

Here's how to write a sector to the disk. First, get the IOB address into $48.49 as described above. Then set the parameters as follows:

Byte	$01:	slot number times 16, normally $60
Byte	$02:	drive number, $01 or $02
Byte	$03:	volume number, $00 to write any volume
Byte	$04:	track $00. . .22
Byte	$05:	sector $00. . .0F
Byte	$08.09:	Buffer address. This is the data to be written.
Byte	$0B:	normally $00, you can set the byte count for a partial sector instead.
Byte	$0C:	command code, $02 for WRITE

If you are using your own IOB instead, you must also set:

Byte	$00:	to $01
Byte	$06.07:	address of DCT
Byte	$0E:	volume number last accessed
Byte	$0F:	slot number times 16 last accessed
Byte	$10:	drive number last accessed

Upon return from RWTS, you test for error with the C-flag. If set, you can find the error in byte $0D according to Table 7-15.

Here's how to format a disk. You get a data disk with no DOS, no VTOC, and no Directory. Just 35 formatted tracks of 16 sectors each, with the sync bytes and fields written to allow RWTS to access it directly. Use this command call to make data disks for your own direct access method.

First, get the IOB address into $48.49 as described. Then set the following parameters:

Byte	$01:	slot number times 16, normally $60
Byte	$02:	drive number, $01 or $02
Byte	$03:	volume number to create: $00 . . . FE
Byte	$0C:	command code, $04 for FORMAT

If you are using your own IOB, you must set:

Byte	$00:	to $01
Byte	$06.07:	address of DCT
Byte	$0E:	volume number last accessed
Byte	$0F:	slot number times 16 last accessed
Byte	$10:	drive number last accessed

Upon return from RWTS, you test for error with the C-flag. If set, you can find the error in byte $0D according to Table 7-15.

Here's how to position the read/write head. When you want to select a given track ahead of time to speed things up or to select the track for the hardware-level reads and writes, you can position the head to that track without RWTS doing a read or write on its own. First, get the IOB address into $48.49 as described. Then set the parameters as follows:

Byte $01:   slot number times 16, normally $60
Byte $02:   drive number, $01 or $02
Byte $04:   track number to position $00. . .22
Byte $0C: command code: $00 for POSITION

If you are using your own IOB instead of the system's, you must also set:

Byte 00:   to $01
Byte $06.07:   address of DCT
Byte $0F: slot number times 16 last accessed
Byte $10:   drive number last accessed

Upon return from RWTS, you test for error with the C-flag. If set you can find the error in byte $0D according to Table 7-15.

Example 7-1 (listed earlier in the chapter) is a Disk Zap program in Integer BASIC. The routine to call RWTS is put into Page Three using Lam's method in the mainline. Another routine at $2100 displays the contents of a buffer at $2000 a half page at a time when called by the routine at line 14000. These three machine language calls, $E88A, $0300, and $2100, are represented at the bottom of the Structure Diagram in Fig. 7-10.

When run, it gives you an "INITIALIZING" message while Lam's method sets up the machine language routines at $300 and $2100. Then a routine at line 29000 displays the commands: Read, Write, List half the data buffer, Change contents of data buffer, and Quit the program. Each command has its own routine in the 10000. . .199999 line number range. A routine at 1200 parses the track and sector numbers in the Read and Write commands. The List shows the other half of the buffer, so you can see the entire buffer by repeated L commands to view alternately the $00.7F and $80.FF halves. The Change command has Monitor syntax; for instance,

C25:00 01 02

put $00 in byte $25 of the buffer, $01 in byte $26, $02 in byte $27.

It is simple to use as it stands, but you may change it for yourself. In particular, you can rewrite it in Applesoft BASIC when you key it in if you don't have Integer BASIC. This lets you simplify some expressions and eliminate the CHR$ routine at line 100.

Fig. 7-10. Disk zap call structure.

7.2.4 Nibble

The RWTS section contains all the routines to access the disk. The way it works is by having all disks formatted with their address fields. Then, any sector is read or written by searching for its address field and following it by accessing its data field. So, each newly formatted disk has 35 tracks formatted with sync bytes and 16 address fields. The RWTS write routines write the data fields one sector at a time onto the disk from your buffer. Similarly, read routines get the data field from the disk for the sector you requested and deliver it decoded to your buffer. You can see how each of the RWTS routines work to do these accesses.

The routine that writes an address field is at $BC56. You can look at it there and see how it does the write. It uses the Y-reg as the number of sync bytes to write first. Then it writes the prefix bytes $D5, $AA, $96. Next the volume, track, sector, and checksum are written, fol-

lowed by the suffix bytes $DE, $AA, $EB. The format routine in RWTS calls this one 16 times for each track.

When address bytes are written, they are packed in four bit nibbles (sometimes spelled nybbles) as described earlier in Section 7.1.3. The routine that does this packing is at $BCC4.

When the data field is written, it's a bit more complicated. Not only must the track be SEEKed, the sector must be found by reading back the address fields until the one you want is found. Then the buffer you gave to RWTS must be written in the data field in a special format. The address field is read at RDADR ($B944) and the data are transformed to disk format in two stages, see Fig. 7-11.

Fig. 7-11. Writing a data field.

First, the 256 bytes in your buffer are converted to 342 six-bit nybbles by the PRENYBBLE routine. Then, the sector is found, and the WRITE routine converts the nybbles to disk format by using the write translate table to look up the code for each of the 342 nybbles in PRI and SEC. The WRITE routine also prefixes the data field with five sync bytes, then $D5, $AA, $AD. When written, the data field gets its suffix of $DD, $AA, $EB. The result is your 256-byte buffer encoded to the data field of the specified sector.

Similarly, RWTS reads a given sector. First, the track is SEEKed and the sector address field found by using RDADR. Then the data field is read into the PRI and SEC nybble buffers by looking up the nybble six-bit values from the coded bytes using the read translate table at $BA96. See Fig. 7-12. After the READ routine at $B8DC is finished, the POSTNYBBLE routine at $B8C2 packs the 342 six-bit nybbles from PRI and SEC to your 256-byte data buffer.

All these routines are called by the RWTS main routines who see to it that registers, Page Zero pointers, and the scratchpad RAM in

Fig. 7-12. Reading a data field.

TEXT1 are given the parameters appropriate to each. RWTS gets these parameters from the IOB you gave it. You don't have to know anything about these routines to use RWTS from the IOB level, but studying how they do each separate task in converting data and accessing the disk is necessary if you want to work with raw disk dumps. You can, with practice, learn to get raw disk bytes and interpret them to recover lost data or explore disk protection schemes.

7.2.5 Hardware

By addressing the hardware you can read and write directly to the disk. Bypassing RWTS you can access the sync bytes, address fields, and data fields in each track without the automatic decoding and encoding. Here's how.

The hardware is in the DEVICE SELECT address space. For Slot Six, this is the range $C0E0.C0EF. You can address the general range $C080.C08F indexing it with 16 times the slot number instead. This lets you write slot independent routines, although the examples given here are for Slot Six only. See Table 7-16.

The addresses $C0E0.C0E7 control the stepper motor. By turning on each phase for a certain length of time in proper sequence, the read/write head can be moved radially over the disk to any desired track. The routines to do this are a bit complex, so the best way to do it is to use the SEEK command in RWTS. If you are experimenting with unusual track arrangements, you can try varying the parameters in the Device Control Table at $B7FB. See Section 7.2.3 for the call sequence.

Table 7-16. DISK II Device Addresses

Label	Address	Description
PHASOFF	$C0E0	Stepper motor Phase 0: OFF
PHAS0ON	$C0E1	Stepper motor Phase 0: ON
PHAS1OFF	$C0E2	Stepper motor Phase 1: OFF
PHAS1ON	$C0E3	Stepper motor Phase 1: ON
PHAS2OFF	$C0E4	Stepper motor Phase 2: OFF
PHAS2ON	$C0E5	Stepper motor Phase 2: ON
PHAS3OFF	$C0E6	Stepper motor Phase 3: OFF
PHAS3ON	$C0E7	Stepper motor Phase 3: ON
MOTOROFF	$C0E8	Drive motor: OFF
MOTORON	$C0E9	Drive motor: ON
DRIVE1	$C0EA	Select Drive One
DRIVE2	$C0EB	Select Drive Two
Q6L	$C0EC	Strobe data latch
Q6H	$C0ED	Load data latch
Q7L	$C0EE	Prepare data latch to read
Q7H	$C0EF	Prepare data latch to write

NOTE: For controller card in Slot Six

You can turn the drive motor on and off easily. After a SEEK, you must turn it on to keep the disk spinning. Then it is your responsibility to turn it off when finished. The instructions are:

```
BIT  $C0E9        drive motor ON
BIT  $C0E8        drive motor OFF
```

You can only access one drive at a time. A pair of addresses switch between the two, so you can select the drive directly by:

```
BIT  $C0EA        engage Drive One
BIT  $C0EB        engage Drive Two
```

The write-protect tab on the disk is detected by a microswitch mounted inside the drive. You can read this switch indirectly with this sequence,

```
BIT  $C0ED
BIT  $C0EE
BMI  PROTECT
```

where PROTECT is the address you want to execute if the write-protect tab was found on the disk. This sequence switches two transistors: Q6 high and Q7 low to read the protect switch to the high-order bit of the data latch. If the high bit is set, then the disk is being protected. You must detect this and avoid writing to the disk because the hardware won't protect the disk.

Other combinations of Q6 and Q7 let you do reads and writes with the hardware data latch. To read a byte:

```
            BIT  $C0EE        input mode
    VALID   LDA  $C0EC        strobe data latch
            BPL  VALID
```

To be valid, the high-order bit must be on, hence the BPL. The resulting byte in the A-reg can be tested for $FF with a CMP instruction if you are looking for sync bytes.

To write, you must observe a few precautions. First, you have to check for a write-protect tab as described above. Then,

```
    BIT  $C0EF        output mode
```

given to prepare for output. After that, you need a 100 microsecond delay before the first write can take place. Writing data must be done in a 32-cycle loop so that all bytes are written 32 cycles apart. The A-reg is written by,

```
    STA  $C0ED        to data latch
    ORA  $C0EC        strobe data latch
```

within the loop. Study the examples that RWTS uses for itself.

There is little need to write directly, however. The direct read may be necessary to get data from an unreadable format where RWTS returns drive errors. Once captured, the data may be extracted by recognizing the data fields and using the read translate table, POSTNIBBLE routine, and modifications to decode them. This is heavy stuff; be prepared to spend some time.

The Track Dump program of Example 7-2 lets you examine the raw bytes from any of the 35 tracks on disk. After loading, type

Example 7-2.

```
SOURCE FILE: EXAMPLE 7.2
0000:            1 *****************************
0000:            2 * EXAMPLE 7.2              *
0000:            3 *                          *
0000:            4 *   T R A C K   D U M P    *
0000:            5 *                          *
0000:            6 * USE FROM MONITOR WITH THE *
0000:            7 * CTRL/Y COMMAND:          *
0000:            8 * [TRACK] < [START].[END]  *
0000:            9 *                          *
0000:           10 *****************************
0000:           11 *
0000:           12 *
0000:           13 TRACK   EQU  $00
0002:           14 START   EQU  $02
0004:           15 END     EQU  $04
003C:           16 A1      EQU  $3C        MONITOR PAR
MS
003E:           17 A2      EQU  $3E
0042:           18 A4      EQU  $42
0048:           19 IOB     EQU  $48
0000:           20 *
0000:           21 * MONITOR CALLS
0000:           22 *
FBDD:           23 BELL1   EQU  $FBDD
FF69:           24 MONZ    EQU  $FF69
0000:           25 *
0000:           26 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 7.2.OBJ0

0300:           27        ORG  $0300
0300:           28 *
0300:A9 4C      29        LDA  #$4C        THE JMP OPC
ODE
0302:8D F8 03   30        STA  $03F8       CTRL/Y VECT
OR
0305:A9 12      31        LDA  #>DUMP
0307:8D F9 03   32        STA  $03F9
030A:A9 03      33        LDA  #<DUMP
030C:8D FA 03   34        STA  $03FA
030F:4C 69 FF   35        JMP  MONZ
0312:           36 *
0312:           37 * FIRST GET PARAMETERS FROM
0312:           38 * THE CALLER AND IOB POINTER
0312:           39 * FROM DOS.
0312:           40 *
0312:A5 42      41 DUMP   LDA  A4          GET TRACK P
ARM
0314:85 00      42        STA  TRACK
0316:A5 3C      43        LDA  A1          GET START A
DDRESS
0318:85 02      44        STA  START
031A:A5 3D      45        LDA  A1+1
031C:85 03      46        STA  START+1
031E:A5 3E      47        LDA  A2          GET END ADD
RESS
0320:85 04      48        STA  END
```

Example 7-2 Cont.

```
0322:A5 3F     49          LDA   A2+1
0324:85 05     50          STA   END+1
0326:20 E3 03  51          JSR   $03E3     GET IOB POI
NTER
0329:84 48     52          STY   IOB
032B:85 49     53          STA   IOB+1
032D:          54 *
032D:          55 * SECOND SET IOB AND CALL
032D:          56 * RWTS TO SEEK THE TRACK.
032D:          57 *
032D:A0 04     58          LDY   #$04
032F:A5 00     59          LDA   TRACK
0331:91 48     60          STA   (IOB),Y   SET TRACK N
UM
0333:A4 0C     61          LDY   $0C
0335:A5 00     62          LDA   $00       SEEK COMMAN
D
0337:91 48     63          STA   (IOB),Y   SET
0339:20 E3 03  64          JSR   $03E3     GET IOB ADD
RESS
033C:20 D9 03  65          JSR   $03D9     CALL RWTS T
O SEEK
033F:A9 00     66          LDA   #0
0341:85 48     67          STA   $48       MONITOR FIX
!
0343:90 03     68          BCC   DUMP1
0345:4C DD FB  69          JMP   BELL1     ERROR EXIT
BEEPS
0348:2C E9 C0  70 DUMP1    BIT   $C0E9     KEEP MOTOR
ON
034B:          71 *
034B:          72 * THIRD SEARCH FOR BEGINNING
034B:          73 * OF A FIELD ON THE DISK.
034B:          74 *
034B:A2 00     75          LDX   #0
034D:2C EE C0  76          BIT   $C0EE     SET LATCH T
O READ
0350:AD EC C0  77 DUMP2    LDA   $C0EC     STROBE LATC
H
0353:10 FB     78          BPL   DUMP2     UNTIL BYTE
IS VALID
0355:C9 FF     79          CMP   #$FF      SYNC BYTE?
0357:D0 F7     80          BNE   DUMP2     NO..TRY AGA
IN
0359:AD EC C0  81 DUMP3    LDA   $C0EC     LOOK FOR SE
COND
035C:10 FB     82          BPL   DUMP3     SYNC BYTE
035E:C9 FF     83          CMP   #$FF
0360:D0 EE     84          BNE   DUMP2
0362:AD EC C0  85 DUMP4    LDA   $C0EC     WE HAVE TWO

0365:10 FB     86          BPL   DUMP4     SYNC BYTES!
IGNORE
0367:C9 FF     87          CMP   #$FF      ANY FURTHER
ONES.
```

Example 7-2 Cont.

```
0369:F0 F7    88          BEQ   DUMP4
036B:         89 *
036B:         90 * READ DISK TO MEMORY RANGE
036B:         91 * GIVEN BY THE CALLER.
036B:         92 *
036B:D0 05    93          BNE   DUMP6       (ALWAYS)
036D:AD EC C0 94 DUMP5    LDA   $C0EC       READ A BYTE

0370:10 FB    95          BPL   DUMP5
0372:81 02    96 DUMP6    STA   (START,X)   PUT BYTE
0374:E6 02    97          INC   START
0376:D0 F5    98          BNE   DUMP5       BUMP PIONTE
R
0378:E6 03    99          INC   START+1     UNTIL LAST
PAGE
037A:A5 03    100         LDA   START+1
037C:C5 05    101         CMP   END+1
037E:D0 ED    102         BNE   DUMP5
0380:         103 *
0380:         104 * ALL DONE: CLEAN UP.
0380:         105 *
0380:2C E8 C0 106         BIT   $C0E8       MOTOR OFF
0383:4C 69 FF 107         JMP   MONZ        BACK TO MON
ITOR
```

*** SUCCESSFUL ASSEMBLY: NO ERRORS

300G

to the monitor, to set the ctrl/Y jump.

To get any track, you type a monitor command with the ctrl/Y. For instance, if you want Track 17, choose a chunk of memory to contain it — say, $1000.3FFF. This command is

11<1000.4000(ctrl/Y)(CR)

where $11 is the track number. In general,

{track} < {start}.{finish}(ctrl/Y)

Another one is given in Example 7-3. This command fetched Track 17 ($11) to memory $4000.5FFF. Here is what it shows.

The first three bytes, $D5, $AA and $96, are the prefix to an address field. The next eight bytes then contain the volume, track, sector, and checksum. Then at $400B you can see the suffix bytes of $DE, $AA, and $EA. This last byte, $EA, is nominally $EB but is not verified so it can vary somewhat in actual value.

Example 7-3.

```
*11<4000.6000

*4000.43FF

4000- D5 AA 96 EE EA AA BB AB
4008- AA EF FB DE AA EA D2 FF
4010- F3 FC FF FF FF FF D5 AA
4018- AD A6 AF 9D AE B3 AF A7
4020- AE AE 9D AE AD B3 9D 9B
4028- D7 D9 D9 F2 B5 D6 D9 97
4030- D6 F2 F2 D6 9B DD F5 D6
4038- F7 A7 96 A7 AE AE 9D AE
4040- 96 A7 A7 A7 A7 AE AE 97
4048- D7 D6 9B 9B 9A 9A EF 9A
4050- B7 EF B7 EF B6 B7 B5 D6
4058- 96 96 AE 9D A7 A7 96 A7
4060- 9D AE 9D AE 96 A7 A7 A7
4068- AC B2 B2 D7 DA ED B6 CE
4070- FB A6 A6 A6 9B 96 96 96
4078- 96 96 9F 9D DA BA 97 97
4080- 9E 9E CD BE 9B 9F 9F BF
4088- BD 9B A6 9A CF 96 96 96
4090- 96 96 96 96 96 96 96 96
4098- 96 96 96 FC B9 9E A6 9A
40A0- F3 9F 9A 9D 9E BE CE 97
40A8- 9F A6 9D 9E CD 96 96 96
40B0- 96 96 96 96 96 96 96 96
40B8- 96 96 96 96 96 96 E9 9D
40C0- 9E 9D 97 EF 96 9F 9E 9E
40C8- CD BD 9A 9A 9D 9D 96 9E
40D0- 9D 9E CD 96 96 96 96 96
40D8- 96 96 96 96 96 96 96 96
40E0- 96 E6 9A 9E 9F 9B EF 96
40E8- 9F 9E 9E CD CD 9F 9D A6
40F0- 9A A6 CD 96 96 96 96 96
40F8- 96 96 96 96 96 96 96 96
4100- 96 96 96 96 EA 9E 9D 9F
4108- 9A F2 97 9B 9E 9E 9E CD
4110- CD 9F 9D A6 9A A6 CD 96
4118- 96 96 96 96 96 96 96 96
4120- 96 96 96 96 96 96 96 D6
4128- A7 9D 9F 9A EF 97 A6 97
4130- 96 CE BD 9B 9E 9B A6 96
4138- 97 9A BE 96 96 96 96 96
4140- 96 96 96 96 96 96 96 96
4148- 96 96 E5 97 9D 9E 97 F2
4150- 96 9F 9D 9E CD BD 96 9D
4158- 9E BE 96 96 96 96 96 96
4160- 96 96 96 96 96 96 96 96
4168- 96 96 96 96 96 E6 9A 96
```

Look at the contents of the address field. Each two bytes encode one byte of information as follows:

$$1b_7 1b_5 1b_3 1b_1 \quad 1b_6 1b_4 1b_2 1b_0$$

The algorithm to decode is to shift left the first byte then AND the two together. Doing it by hand, you pick out the bits in order of significance and write them:

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

For instance, take the volume encoded as $EE, $EA. Expanding in binary

$$11101110\ 11101010$$

then pick out the informative bits,

$$-1-0-1-0\ \ -1-0-0-0$$

then rearrange in order of significance,

$$11001000$$

which is $C8, decimal 200. The volume number is 200. You know the track number is 17 ($11), so you can decode the next two bytes, $AA and $BB, yourself and check your result.

After the address field, you can see some "garbage" and $FF sync bytes before the data field at $4016. The three bytes, $D5, $AA, and $AD, prefix the data. The data is encoded sixbit, so you must use a translate table in DOS to convert the codes to sixbit. For instance, you will notice a lot of $96 codes in the data. The $96 is the code for $00.

The data field ends with the three bytes, $DE, $AA, and $EB, at $4170. The next sector follows at $4187 with the prefix to its address field. And on it goes.

If you decode the sector numbers in these two address fields, you will find that the first sector is $02 and the second is $03. There is no way of knowing which sector the command will read first from the track; you can consider it as a roulette game. In this case it started with Sector $0B, followed by Sector $09. See the Sector Interleaving in Table 7-7; Sector $0B is physical sector number $02.

CHAPTER EIGHT

Input/Output

8.1 BUILT-IN I/O

You can use the built-in I/O alone in the case of the speaker, or keyboard. For video display, you need a monitor connected from outside. Similarly for the cassette recorder and various devices that connect to the games socket: joysticks, relays, pushbuttons, etc. The advantage to using built-in I/O is in not having to use a peripheral card.

8.1.1 Cassette Tape

You can store up to four disks' worth of files on one C-60 cassette tape. Considering the difference in cost between the two media, tape is the best choice for archival storage. Use good quality tapes to reduce the chance of *dropout* in your recordings. Dropout is missing bits on the tape due to uneven magnetic oxide coating; it won't be noticed in listening but it causes errors in digital reads. Choose a standard bias, low noise, C-30 or C-60 tape like the Maxell UD-30 or UD-60.

As far as a tape recorder is concerned, you can use any one you happen to own as long as you can get it to work after adjusting the volume and tone. If you are buying a new one specifically for the Apple, then choose a plain mono type with a counter. Since the Apple was first used, two of the most trouble-free models have been the Panasonic RQ-2309 and the Sony TCM-737. People have used other brands as well, but try any new recorder out first. Some machines

won't work; they can't be adjusted to the Apple's volume and tone requirements. Recording and playing back an Apple file is the only way to be sure.

Accessories you need to complete the cassette recorder include two miniature phone plug cables, Radio Shack 42-2420. Also, you should have cotton swabs and isopropyl alcohol to clean the heads and capstan. A head demagnetizer should be used at least every couple of months or so. If you depend on the machine, have it cleaned and adjusted by a technician once a year to maintain reliability.

To use, you must find the volume and tone control settings. On a compatible recorder, this will be about a third volume and near full tone. Listen to a pre-recorded tape and adjust to get a clear tone at the beginning of each recording. This is the header record and is a constant tone lasting about ten seconds.

Connect the IN jack to the recorder's MON output jack using one of the cables. Connect the OUT jack to the recorder's MIKE input jack. Now you can attempt to record and play back a file to confirm the volume and tone settings.

To record, mount a new tape. Rewind. Without changing the controls from the settings you made by ear, enter the following command to the Apple Monitor:

D000.FFFFW

without a RETURN. Then put the recorder in RECORD mode. With the tape running, press RETURN to enter the tape write command. When finished, the Monitor's asterisk (*) returns. Stop the recorder.

To play back, rewind. Enter the command

1000.3FFFR

to the Monitor, again without a RETURN. Put the recorder in PLAY mode, then press RETURN with the tape running. If all is well, the Monitor will return with an asterisk when the playback is finished. If it cannot READ anything, it will just do nothing — looking for the recording forever. If it reads but finds errors, you get an ERR message on the screen.

If the playback was read successfully, compare the copy at $1000.3FFF with the original at $D000.FFFF using the verify command in the Monitor. They should match exactly.

If you cannot record and playback successfully, try another setting. Repeat the procedure until successful. Then mark the volume and tone settings on the recorder using a dab of nail polish or typewriter correction fluid.

If you want to hear the playback, unplug the MON jack temporarily. To make sure it is getting through to the Apple, leave it plugged in and play back using the following routine

```
0300:   JSR   $FCFD        read a bit
        LDA   $C030        toggle speaker
        JMP   $0300
```

instead of the READ command. This will sound the speaker with the incoming bit stream so you can hear it.

You can read and write BASIC programs to and from tape easily. To save a BASIC program, type

WRITE

then start the recorder. With the tape running, press RETURN. When finished you will see the BASIC prompt, a] or >. Similarly, use the

READ

command to load a BASIC program from tape. If it cannot read, it will wait forever or give you an ERR message.

Remember not to use a file name like you do with DOS commands. A READ or WRITE command without a file name will be accepted as a tape command.

To save and load binary files, enter the Monitor first. Then use the R and W monitor commands with *start* and *end addresses*. The procedure in giving commands and starting the recorder is the same regardless of file type. One point you should keep in mind when working with tape is that you must provide start and end addresses for reads as well as writes. With disk, you can BLOAD the file into the same memory that it was BSAVEd from; with tape this is not automatic. You have to remember where the file resides in memory.

An exception to this is the SHLOAD command in Applesoft. The binary file is read by this command, but Applesoft gives the start

address for you. When the tape is read, the binary file is put below the
MEMSIZ address and the start address of the file is given to you at
$E8.E9 in Page Zero. Applesoft uses this address for all its shape table
commands; see Chapter Six for more information on shape tables.

To read and write binary files from an assembler program, you set
the start address in location $3C.3D and the end address in $3E.3F.
Then use a JSR to the READ routine at $FEFD or to the WRITE rou-
tine at $FECD. With the READ routine, you will need a method to
detect errors. One way is to save CH ($36) in Page Zero before making
the READ. This is the horizontal screen cursor and will change during
the read if an error occurs. This is because the READ routine will
print an ERR message to the screen, advancing the cursor as it does
so. Upon return from the READ, you can compare CH to its previous
value. If it has changed, then you know an error has occurred. Such
an error detector might look like

```
        LDA   CH            get cursor
        STA   OLDCH
        JSR   READ          read tape to (A1.A2)
        LDA   CH            new cursor
        CMP   OLDCH         same?
        BNE   ERROR         no. . .ERR in READ
        . . .               yes. .no error in READ
```

where OLDCH is any RAM location.

A binary file has a format on tape like that of Fig. 8-1. For BASIC

Fig. 8-1. Taped binary file.

files, a more complicated format is used; see Fig. 8-2. The READ and
WRITE routines handle these formats for us so we don't normally get

Fig. 8-2. Taped BASIC file.

involved in the formats. Knowing the details is needed if you want to write special tape routines, like an Apple tape loader for another computer.

Here's how the Apple II reads and writes tapes. There are two kinds of files built-in to the tape routines, BASIC and binary. Binary files are the simplest. To locate any file on a tape, you must listen for it or use the tape counter. Some people use the recorder's microphone to record voice cues between files. Once located, a steady tone lasting about ten seconds marks the beginning of the file.

This tone is called the *header*. It consists of a 770-Hz square wave, 1300 microseconds each cycle. At the end of the header is a special sync bit lasting 450 microseconds. The sync bit is followed by the data itself at 1500 baud (bits per second). One last byte contains the checksum. The header, sync bit, data, and checksum byte make one binary file.

BASIC files are kept as two binary files, one immediately following the other. That is why you hear two beeps for BASIC files and only one beep for binary files. The first binary file is fixed. It contains two bytes (Integer) or three bytes (Applesoft) and tells the BASIC the length of the program to load. The second binary file contains the program itself as data. It can be any length, and the BASIC loader knows that length from the first file.

Two bytes are always needed in the length file. Integer BASIC files have two bytes, but Applesoft has three. The third byte is read into Page Zero at $D6. The value normally written is $55. If a value greater than $7F was written, then the Applesoft will RUN the program after the READ is finished and disable immediate execution. Unless you can get into the Monitor and change location $D6 back to $55, you cannot use Applesoft to LIST the program or do anything else. This "feature" is not used by the normal WRITE command but some commercially distributed tapes might. A Standard Apple Monitor will RESET to the Monitor command interpreter where you can change location $D6 to its *safe value* of $55 if you experience this problem.

Bits are created by toggling the OUT jack during the recording session. For instance, the HEADR routine creates a square wave by toggling each 650 microseconds to give a period of 1300 microseconds. At the end of its count, it toggles after 200 microseconds and again after 250 microseconds to create the sync bit. The sync bit therefore lasts 450 microseconds but is not symmetric. When writing data, each bit is symmetric but has one of two different periods. A *one* bit is en-

coded as a 1000 microsecond-cycle and a *zero* bit as a 500-microsecond cycle. This gives a frequency of 1000 Hz for ones and 2000 Hz for zero. Such encoding schemes are often called *frequency shift keying* or FSK for short. With FSK of 1000 Hz and 2000 Hz the average gives the transmission rate for recording and reproducing the data — 1500 Hz as 1500 baud.

Bits are read and written by two monitor routines called WRBIT ($FCD6) and RDBIT ($FCFD). They address the IN and OUT hardware addresses at $C060 and $C020.

Table 8-1 summarizes the addresses used by tape routines; see Chapter Two for further notes on each location.

Table 8-1. Summary of Tape Addresses

Label	Address	Contents
CHKSUM	$002E	Checksum EORed during READ and WRITE
CH	$0036	Cursor changed by READ if ERR
A1	$003C	Start address for READ and WRITE
A2	$003E	End address for READ and WRITE
	$00D6	Inhibits Applesoft when > $7F
SHADDR	$00E8	Shape table start address
	$C020	Cassette "OUT" port
	$C060	Cassette "IN" port
HEADR	$FCC9	Writes header tone and sync bit
WRBIT	$FCD6	Writes one bit
RDBIT	$FCFD	Reads one bit
WRITE	$FECD	Writes binary file from (A1) to (A2)
READ	$FEFD	Reads binary file into (A1) from (A2)

8.1.2 Games Socket

Many of the built-in I/O features of the Apple II are collected in the "games" socket on the motherboard. This is a 16-pin DIP socket in the right rear area, designated as J-6 on the Apple II. This is where the game paddles plug in with a 16-pin DIP header plug. In addition to paddles, you can plug in other devices using the various pinouts provided.

Most device hookups require the 5 volts on pin 1 and ground lines on pin 8. Up to 4 game paddles (or 2 joysticks), 3 switches, and 4 TTL outputs called annunciators are available. In addition, one line called a strobe can be brought low during Phase Zero for a cycle by address-

ing $C040 with a read instruction. Although seldom used, this line can be useful in enabling or gating special TTL circuits you may build.

On the IIe model, an adaptor cable from the games socket provides the switches, paddles, and power lines to a DB-9 connector on the back of the case. This is a good idea, and if you have another Apple II model, you may want to add this cable yourself to upgrade your Apple. The game paddles and switches are the most often used lines on devices plugged in there. The 16-pin DIP socket is fragile unless you are used to handling hardware and don't plug and unplug the cable often. When the connection is on the DB-9 at the back, you can plug and unplug your joystick/game paddles much easier and reliably. The connections are shown in Table 8-2.

Table 8-2. Games Socket Pinouts

16-pin DIP	Name	Address	Description	DB-9 connector
1	5 VOLTS		Maximum 300 mA	2
2	SW0	C061	Switch ON if > 127	7
3	SW1	C062	Switch ON if > 127	1
4	SW2	C063	Switch ON if > 127	6
5	STB	C040	Strobes when addressed	
6	PDL0	C064	Game paddle resistance	5
7	PDL2	C066	Game paddle resistance	8
8	GND		Signal ground	3
9	N.C.		No connection	
10	PDL1	C065	Game paddle resistance	4
11	PDL3	C067	Game paddle resistance	9
12	AN3	C05E.C05F	Clear/set annunciator	
13	AN2	C05C.C05D	Clear/set annunciator	
14	AN1	C05A.C05B	Clear/set annunciator	
15	AN0	C058.C059	Clear/set annunciator	
16	N.C.		No connection	

The switches are used extensively. On game paddles and joysticks, SW0 and SW1 are connected to the pushbuttons where you can use them for graphics control such as scaling and pen up/down functions. On the IIe model, they are read by the RESET routine to determine which of the many RESET routines that the model IIe has will be executed. In particular, SW0 is connected to the OPEN-APPLE key just left of the spacebar and SW1 is connected to the SOLID-APPLE

key just right of the spacebar. On earlier Apples, you may connect the SW2 to the SHIFT key on the keyboard for use by a lowercase routine. This modification is described later in this chapter, in Section 8.1.4.

When designing custom interfaces, remember that the switches are simple LS inputs and the annunciators are LS outputs with little fan-out; use buffers.

If you plan to use the annunciators at all, it's a good idea to make a *state tester* first. Take a 16-pin DIP header and connect four LEDs to it with series resistors to pins 12, 13, 14, and 15. Return the cathodes to ground at pin 8. Then, you can see the LEDs toggle when you address them in the $C058.C05F range.

Use the annunciators to drive relays like the Clare 1896. See Fig. 8-3. A 2N2222 makes a low cost current amplifier to drive the winding

Fig. 8-3. Annunciator output relay circuits.

from the TTL output. Always protect relay input circuits with a diode like the 1N914 across the winding; the back EMF can clobber solid-state quite easily. For simple secondary circuits like a cassette recorder motor switch, a small resistor value in series wiht a 0.01 μF capacitor is usually sufficient. You can control a tape recorder by switching the recorder off when not writing to tape so that you don't create long *dead* sections. Be sure to delay after switching the motor on and be-fore recording again, to allow the tape to reach operating speed.

Other output circuits are possible; relays, SCRs, and triacs may be used. The 2N2222 buffer is a simple buffer that you may have to adapt. Don't try to use current devices without amplification; the LS output just won't have the source current to make most of them work.

You can use game paddles or a joystick on each pair of analog in-puts. Most connect to PDL0 and PDL1. To use the built-in routines, you should use paddles or joysticks having a full resistance of 140

kilohms each. Whenever a paddle is read, this resistance gives a value of zero to 255 depending upon the setting of the knob.

To read the value and therefore the position of the knob, use the PDL(n) function in BASIC where n is the paddle number, 0, 1, 2, or 3. Normally, PDL(0) and PDL(1). If you are assembling, the routine is in the Monitor at $FB1E, returning the value in the Y-reg, $00 to $FF. In either case, you need a resistance of zero to 140 kilohms to get values from zero to 255.

Once you have read one paddle resistance this way, you must delay before reading the paddle again, or even reading any other paddle. What happens is that reading the paddle discharges four capacitors through the four resistances of the paddles (as connected to the games socket). Then the routine counts until your paddle has discharged the capacitor and clears bit 7 at its address. Once this is finished, the capacitor is recharged for the next call. If you call too soon, the capacitors don't have enough time to recharge and you get a too-low reading. This interference between successive readings is simply a function of time; use a delay loop if your routine has nothing else it can do.

If you look at the PREAD routine at $FB1E in the Monitor, you can see how it works. First, the $C070 address is referenced to fire a special timer chip (acts like four 555's). This chip discharges four 0.022 μF capacitors through any conducting paths connected to the games socket pins PDL0, PDL1, PDL2, and PDL3 to the +5-volt line. The higher the resistance of each path, the longer the discharge will take. By testing location $C064,X where X is 0, 1, 2, or 3, the routine detects one of these time-outs. By counting with the Y-register once each twelve cycles, the routine has its return value when the time-out occurs. To count all the way to 255 the resistance must be at least 150 kilohms; if you use smaller resistance values, the range of return values will be correspondingly less. You call PREAD with the paddle number in the X-reg, and pickup the returned value from the Y-reg.

If you write your own paddle routine, you would do much the same thing. First, use a read instruction with $C070 to start the four timers. Then loop, counting and testing for a time-out on one or more paddles. Finally, return with all counts as the paddle values reflecting the knob position(s). Remember, you are working in real time: use the times given in Table 8-3 to design your loops.

The addresses of the paddles and their routine are summarized in Table 8-4.

Table 8-3. Joystick/Paddle Times

Resistance	Time
500kΩ	11,000 μs
200kΩ	4400 μs
150kΩ	3300 μs
100kΩ	2200 μs
50kΩ	1100 μs
20kΩ	440 μs
10kΩ	220 μs
5kΩ	110 μs
2kΩ	44 μs
1kΩ	22 μs

NOTE: Time constants for discharging 0.022 μF capacitor through given resistances.

Table 8-4. Joystick/Paddle Addresses

Label	Address	Contents
PDL0	$C064	Game paddle time-out
PDL1	$C065	Game paddle time-out
PDL2	$C066	Game paddle time-out
PDL3	$C067	Game paddle time-out
PTRIG	$C070	Game paddle trigger
PREAD	$FB1E	Game paddle read routine

There are several reasons for writing your own joystick routine. A joystick hs two paddle resistances, and both can be read at the same time even though the PREAD routine calls will interfere with each other without an intervening delay. Joysticks can be easily found at good prices with resistance values other than 150 kilohms. The PREAD routine needs a *resolution* of 256; you must be able to make one of 256 different settings to use each control. Because of aging, noise, and the small angle you have in joysticks, this is almost impossible to maintain. Your own routines can deal with the resolution problem by reducing it and scaling. You can handle resistances other than 150 kilohms. And once started, your timers can be counted simultaneously to return both X and Y direction values from only one call.

As an example, look at the joystick schematic in Fig. 8-4. This one uses a 100-kilohm joystick from Radio Shack (271-1705) with a pair of pushbuttons (275-8077) mounted in a case (270-231). A 75-cm

Fig. 8-4. Joystick schematic.

(30-inch) length of ribbon cable with a 16-pin DIP header plug provides the Apple interface. The four resistors are ¼-watt each and limit the current to each device. The joystick resistance is chosen largely on availability and price. Such a choice reduces the range of values from PREAD or the PDL(n) function by about a third. See the table of Joystick/Paddle Times in Table 8-3.

Table 8-3 is just a list of selected values for the times taken to discharge a 0.022 μF capacitor through various resistances. For the design value of 150 kilohms, this is 3.3 milliseconds, while for the 100 kilohms of the Radio Shack joystick this is 2.2 milliseconds or about two-thirds of the design value. The general formula for capacitor discharge time is

$$t = RC$$

where t is in milliseconds, R is in kilohms, and C is in microfarads (μF). The discharge time is the time the paddle circuit takes from the $C070 reference until the bit 7 changes at the paddle address from on to off.

The JOYSTICK routine of Example 8-1 works with the 100-kilohm

Example 8-1.

```
SOURCE FILE: EXAMPLE 8.1
0000:               1 ******************************
0000:               2 * EXAMPLE 8.1              *
0000:               3 *                          *
0000:               4 *   J O Y S T I C K        *
0000:               5 *                          *
0000:               6 * READS JOYSTICK IN GAMES  *
0000:               7 * SOCKET.  BOTH RESISTANCES *
0000:               8 * READ SIMULTANEOUSLY, BUT  *
0000:               9 * WITHOUT 256 RESOLUTION.   *
0000:              10 *                          *
0000:              11 * RETURNS VALUES IN X-REG AND*
0000:              12 * Y-REG.  A-REG CLOBBERED.  *
0000:              13 ******************************
0000:              14 *
0000:              15 *
C064:              16 XTOUT   EQU   $C064     X-VALUE TIM
EOUT
C065:              17 YTOUT   EQU   $C065     Y-VALUE TIM
EOUT
C070:              18 PTRIG   EQU   $C070     PADDLES TRI
GGER
0000:              19 *
0000:              20 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 8.1.OBJ0

0300:              21          ORG   $0300
0300:              22 *
0300:              23 * A TEST CALL SEQUENCE.
0300:              24 *
0300:20 08 03      25          JSR   JOY
0303:86 00         26          STX   $00
0305:84 01         27          STY   $01
0307:60            28          RTS
0308:              29 *
0308:              30 *
0308:              31 *
0308:A2 00         32 JOY      LDX   #0
030A:A0 00         33          LDY   #0
030C:A9 80         34          LDA   #$80      RESOLUTION
030E:38            35          SEC
030F:2C 70 C0      36          BIT   PTRIG
0312:2C 64 C0      37 JOY1     BIT   XTOUT
0315:10 03         38          BPL   JOY2
0317:E8            39          INX
0318:D0 02         40          BNE   JOY3
031A:EA            41 JOY2     NOP
031B:EA            42          NOP
031C:2C 65 C0      43 JOY3     BIT   YTOUT
031F:10 03         44          BPL   JOY4
0321:C8            45          INY
0322:D0 02         46          BNE   JOY5
0324:EA            47 JOY4     NOP
0325:EA            48          NOP
0326:E9 01         49 JOY5     SBC   #1
0328:B0 E8         50          BCS   JOY1
032A:60            51          RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

joystick. Two paddles are counted using the X-reg and the Y-reg for each of PDL0 and PDL1 ($C064 and $C065). The A-reg counts for the entire loop, guaranteeing a fixed number of tests of both time-out addresses. For the 100-kilohm case, a value of $80 is quite sufficient. Each of the resulting 128 times through the loop results in a test of each time-out by using the BIT instruction. The BIT won't change any register value and is needed because all three registers are used as counters. Each of the two tests within the loop is an IF-THEN-ELSE having either a count or a short delay as an action. The delay, done with NOP instructions, ensures that both paths take the same time — eleven cycles each, including the BIT. So, the two tests take 22 cycles within the loop. Adding the loop overhead gives 27 cycles for each loop with a possible count in the X-reg, Y-reg, or both on each.

The longest time for a time-out with the 100-kilohm joystick is about 2200 microseconds. With 27 microseconds for each count, this gives $2200 \div 27$ or about 70 as the largest count value. So, the routine will return a value of zero to 70 in each of the X- and Y-registers as its result.

With a low resolution scheme like this, you avoid several hassles as mentioned earlier, but you have to program a bit more in exchange. For instance, if you want to move a cursor around a screen with the X- and Y- values from the JOYSTICK routine, you have to do some scaling.

For LORES coordinates, you can convert the joystick readings to screen positions by

$$XP = INT(0.5714*XJ)$$
$$YP = INT(0.5714*YJ)$$

where XJ and YJ are the joystick values and

$$40/70 = 0.5714 \text{ (approx)}$$

The idea here is that you want a value of zero to 39 for each position given joystick values from zero to 69. Joysticks vary, so you will have to adjust the 0.5714 factor by experiment.

For HIRES coordinates, you need two scales. One scale for fine positioning, perhaps 0.5 or 0.25 depending on the *feel* you prefer in the application. The other scale for coarse positioning lets you reach

any area on the screen using the joystick before switching scales for fine setting. Given the noise susceptibility of low cost joysticks, this approach is better than the unit factor normally used. To change from zero to 69 to the range of zero to 255, the factor is

$$255 \div 70 = 3.642857$$

giving scale equations of

$$XP = INT(3.6429*XJ)$$
$$YP = INT(3.6429*YJ)$$

for coarse positioning. You can use one of the pushbuttons for scaling to make the cursor positioning easier to handle.

If you have a joystick that has a spring return to center position, you should remove the springs to do position encoding as just described. Alternately, you can use this feature to do velocity encoding instead.

Velocity encoding lets you use low resolution without switching scales. Rather than using the joystick values as positions, you use them to change the current position to a new value. Releasing the joystick knob so that it returns to center stops cursor motion at the current position. It is simple to use and only a little tricky to program.

You call JOYSTICK at the beginning of your program to determine the center position values in X and Y. The user must allow the joystick handle to rest in its center position while this reading is made. You then use these values to calculate several parameters:

$$DL = 0.85 * CJ \qquad \text{called dead low}$$
$$DH = 1.15 * CJ \qquad \text{called dead high}$$
$$SL = 0.40 * CJ \qquad \text{called speed low}$$
$$SH = 1.60 * CJ \qquad \text{called speed high}$$

where CJ is the reading at the center of the joystick. They should be close for both X and Y.

In the cursor move loop of your program, you get the joystick readings and then range test them with DL, DH, SL, and SH. If between DL and DH, don't do anything; this is called the *dead band*. If between DL and SL, decrease the position coordinate slightly. If less than SL, decrease the position coordinate considerably more.

Similarly, increase the position coordinate slightly if the joystick value is between DH and SH and increase it a lot if greater than SH. For example, here's how it might be done for the X-coordinate in HIRES:

```
1000  REM Velocity encode X-coord of cursor
1010  IF XJ > DL AND XJ < DH THEN RETURN
1020  IF XJ > CJ THEN 1040
1022  IF XJ < SL THEN 1026
1024    XC = XC – 1 : GOTO 1028
1026    XC = XC – 8 :
1028     IF XC < 0 THEN XC = 0:
1030  RETURN
1040  IF XJ > SH THEN 1044
1042    XC = XC + 1 : GOTO 1046
1044    XC = SC + 8 :
1046     IF XC > 279 THEN XC = 279
1048  RETURN
```

Similarly, a routine using YJ can velocity encode the Y-coordinate of the cursor position, YC.

The relationship between the velocity of the cursor and the position of the joystick is given in Fig. 8-5. See the dead band in the center, where the cursor doesn't move. See the two speeds in each of the two directions to give it a nice *feel* to the operator.

Fig. 8-5. Velocity encoding a joystick.

8.1.3 Speaker

The simplest and most common way to use the built-in speaker in the Apple is with the monitor routine. This is easily done by typing a ctrl/G on the keyboard followed by a RETURN. This results in a *beep* sound called the *bell*. From an Applesoft program, you can PRINT a CHR$(7) to do the same thing: the ASCII code 7 is called BEL for this purpose.

When writing in machine language you can use the output hook to do the same ctrl/G beep. Load the A-reg with $87 — the negative-ASCII code for BEL — and JSR COUT. Whatever device is acting as video terminal will get the ctrl/G code. Normally, this will be the built-in Apple video at COUT1; in fact by JSR COUT1 instead of COUT you can be sure that the Apple built-in speaker routine will get the ctrl/G if you have a special setup in the output hook you don't want disturbed. There are two other Monitor calls that you can use as well. BELL will load the ctrl/G for you and jump to COUT. The actual routine in the video routines of the Monitor that makes the sound is called BELL1 and you can use it directly. BELL for terminal use; and BELL1 for always ensuring an Apple beep, are usually the best choices of beep routines to call.

The speaker itself is driven by a transistor circuit because the LS TTL just can't supply the kind of power and low impedance a common eight-ohm speaker needs. The transistor circuit is driven from a TTL address decoder so that a read command from the processor at $C030 will toggle the speaker. Two reads will produce one cycle if they are far enough apart. Don't use write instructions because you will get two toggles very close to each other. The speaker and its circuit can only respond to audio frequencies; closely-timed toggles won't be realized, much less heard.

There are many sounds you can make with simple control of the interval between successive toggles. Among these often wanted are ticks, tones, staccato, and trills. Each requires you to toggle the speaker in various kinds of real-time loops. For sounds, frequencies between 100 Hz and 2000 Hz are best. Now, 100 Hz has a period of 10 milliseconds and 2000 Hz one of 0.5 millisecond. Toggling twice each period means that a 100 Hz tone must delay between each toggle so that the time from one toggle to the next is only 5 milliseconds. For the 2000 Hz tone, the time between toggles is 0.25 millisecond or 250 microseconds. These times are easy to get with a fast 1.023 MHz clock (0.9778 microsecond).

An easy way in fact is the WAIT routine at $FCA8. By loading the A-register before the JSR, you can control the delay time this routine takes to return to you; it does nothing except decrement the A-reg to zero. Fig. 8-6 plots the equation

$$t = 0.4889(26 + 27a + 5a^2)$$

where t is the delay time in microseconds and a is the value of the A-register passed to WAIT. Some delay times are shown in Table 8-5 as a result of this equation.

VALUE PASSED IN A-REG

Fig. 8-6. Execution of a WAIT routine.

To make the speaker tick, you toggle it twice with a delay between. Delays between 0.25 millisecond and 5 milliseconds work OK; just try them and choose the one that sounds right for what you want. For instance, a click made with a 1.0 ms interval can be heard by running

```
BIT    $C030              toggle speaker
LDA    #$12               for 1.0 ms(approx)
```

Table 8-5. WAIT Routine Intervals

A-reg	Time (ms)
1	0.028
2	0.049
5	0.140
10	0.389
20	1.255
50	6.79
100	25.8
200	99.1
255	162.0

```
JSR   WAIT
BIT   $C030           toggle speaker again
RTS
```

Now, vary the delay parameter from $12 to get the click to sound the way you want.

Steady tones are done with loops. For each time through the loop, you toggle the speaker at half the period as just explained above. A 1.0 kHz tone is toggled each 0.5 ms to work:

```
TONE    LDX   #$FF          duration of tone
TONE1   LDA   #$0C          for 0.5 ms(approx)
        JSR   WAIT
        DEX
        BNE   TONE1
```

The duration is controlled by setting the number of half-cycles in the X-reg. Notice that the length of time depends on both the duration (X-reg) and half-period (A-reg) defined for the loop. To have the period and duration of the tone independent of each other as parameters is tricky but there is a short routine called Lutas' algorithm that will do just that. With Example 8-2 set $0300 to the period and $0301 to the duration. You can even play musical tunes; see Table 8-6 for the notes.

Staccato sounds use loops. Each staccato has a simple unit, usually a tick or click sound, but a tone can be used for special effect. Simply repeat the unit sound in the loop.

Trills are performed by alternately sounding two tones usually close

Example 8-2.

```
SOURCE FILE: EXAMPLE 8.2
0000:              1 ****************************
0000:              2 * EXAMPLE 8.2              *
0000:              3 *                          *
0000:              4 *    LUTAS' ALGORITHM FOR  *
0000:              5 *    MAKING SPEAKER TONES. *
0000:              6 *                          *
0000:              7 * FROM BASIC:              *
0000:              8 *    POKE 768, PERIOD      *
0000:              9 *    POKE 769, DURATION    *
0000:             10 *    CALL 770              *
0000:             11 *                          *
0000:             12 ****************************
0000:             13 *
0000:             14 *
0000:             15 *
----- NEXT OBJECT FILE NAME IS EXAMPLE 8.2.OBJO

0300:             16          ORG   $0300        FOR 768
0300:             17 *
0300:             18 *
0300:             19 PERIOD   DS    1
0301:             20 DURATN   DS    1
0302:             21 *
0302:             22 *
0302:AD 30 CO     23 TONE     LDA   $C030        TOGGLE SPEA
KER
0305:88           24 TONE1    DEY
0306:DO 05        25          BNE   TONE2
0308:CE 01 03     26          DEC   DURATN
030B:FO 09        27          BEQ   TONE3        FINISHED
030D:CA           28 TONE2    DEX
030E:DO F5        29          BNE   TONE1
0310:AE 00 03     30          LDX   PERIOD
0313:4C 02 03     31          JMP   TONE
0316:60           32 TONE3    RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

to each other in pitch. Each tone must have a short, often equal, duration. A value of $10 or so makes a good starting point when trilling with Lutas' alogrithm. If you want to have a continuous sound without the break in pitch, you can write a gliding tone. Like the trill, it works between two pitches (frequencies), but sounds all the pitches between. The so-called phasor zap sound is a glide.

8.1.4 Built-In Terminal

In the simplest case of built-in terminal usage, you just connect a cable from the video output jack to a video monitor. Alternately, an R.F. modulator is connected between the video output and a television

Table 8-6. Notes for the Tone Routine

Note	Octave 1		Octave 2		Octave 3	
	DEC	HEX	DEC	HEX	DEC	HEX
C	225	E1	113	71	056	38
C#	213	D5	106	6A	053	35
D	201	C9	100	64	050	32
D#	189	BD	095	5F	047	2F
E	179	B3	089	2D	045	2D
F	169	A9	084	54	042	2A
F#	159	9F	080	50	040	28
G	150	96	075	4B	038	26
G#	142	8E	071	47	035	23
A	134	86	067	43	033	21
A#	126	7E	063	3F	032	20
B	119	77	060	3C	030	1E
C	113	71	056	38	028	1C

Use the first two octaves, and only the third octave when really needed. Here are the durations to use.

Length	Duration	
Half	255	FF
Quarter dot	192	C0
Quarter	128	80
Eighth dot	096	60
Eighth	064	40
Sixteenth dot	048	30
Sixteenth	032	20

set. If an 80-column display is needed, you should use a good-quality cable like those sold for video recorders; otherwise, you could lose information and be unable to read the characters on the screen. In such a case, you will use a monitor with at least 8-MhZ bandwidth because a tv just doesn't have the ability to handle 80 columns of text.

The Apple IIe model provides an auxiliary socket where you can plug in an 80-column text card. This converts the built-in display to 80-columns from 40-columns by using soft switches; see Chapter Two. In the IIe monitor, many routines are provided to allow the 80-column extension to be a true extension of the original 40-column display. If you don't have a IIe, then you can plug a regular peripheral card into Slot Three.

In models previous to the IIe, the way to get 80-columns is to use a card like the Videx. In addition to an 80-column display, this card gives you lowercase and keyboard goodies. In particular, you can use

ctrl/A as a shift key. By using the input and output hooks described in Chapter Six, this card replaces the Monitor's routines completely to give you a complete terminal using the built-in keyboard and external video monitor. It is the easiest way to go when you need lowercase and an 80-column display.

If you just need lowercase but don't want to use an 80-column card, then there are a few schemes to do this.

The simplest and cheapest way to get lowercase is to use a cascade in the output, as described in Section 6.1. The LCOUT routine given there converts uppercase to inverse and lowercase displays normally. Apple Computer Inc., markets a text editor called Applewriter that uses this scheme. It is an inexpensive and effective method to have lowercase.

Another way is to buy a lowercase adaptor. You install it yourself or the dealer will do it for you. For Apples of Revision 7 or greater, it is inexpensive and simple. A more complicated adaptor is needed for the earlier (than Revision #7) models, so make sure you get the right one for you. Revision 7 and later models have a slide switch added to interlock the RESET key to the CTRL key, located just under the front opening when the cover is removed. The adaptor may have a cable to the games socket; this lets the SHIFT key switch SW2 for software detection.

Finally, you can do it yourself. The hardware you need is simple, but you need a character set in ROM. One scheme described by Don Lanacaster in *Son of Cheap Video* uses a Motorola 6674 character generator and modifies the Apple. Another scheme replaces the 2513 ROM with a 2716 EPROM that you must program. See *Apple Orchard*, Vol.1, No.1 (Mar/Apr. 1980) for hardware details and firmware listing.

If you have an old Apple, earlier than Revision 7, then you may have a *live* RESET key. Unless modified, this key will work by itself. Modify it so that anyone wanting to do a RESET must also press the CTRL key. Otherwise, you will cause a RESET sometime when you only mean to press the nearby RETURN key.

Here's how. Remove the cabinet from the steel base plate and unplug the keyboard. Remove the keyboard. Look at the CTRL key; you will see two unused pins. Connect them to a pair of wires; AWG30 wire-wrap wire will do fine. With this pair, connect the RESET key in series with the CTRL key. Keep leads dressed so as not to catch on anything during reassembly. Replace keyboard and plug back into the

motherboard. Replace cabinet. RESET should not work now unless CTRL is also held down.

So, if you want more than the built-in terminal offers, install a card or make the keyboard modifications you need. Lowercase adaptor schemes are available alone or on 80-column cards. The alternative to 80-column cards is a serial card in Slot Three to use an external, stand-alone terminal. This scheme is sometimes used for full screen graphics from the Apple video output with the text being exchanged with the user on the external terminal.

8.2 PERIPHERAL I/O

8.2.1 The Apple Bus

In Chapter One, the use of peripheral cards was discussed and Fig. 1-1 gave the locations of the slots where they are plugged in. Here, the card itself is discussed. In particular, this section deals with how to use the Apple bus to design and build your own cards. Even for a beginner, this is not too difficult, provided the interface needed is simple.

The Apple bus is defined as the pinout on the seven slots that accept peripheral cards. Each pin is labeled by number and name as shown in Fig. 8-7. To make your own peripheral interface, you use a Hobby/Prototyping Board like that of Fig. 8-8 and make connections by soldering, wire-wrapping, or both. Wire wrap is probably the easiest; you can modify your work until you are satisfied with its performance.

The first thing you can get from the bus onto your card is power. The + 5 volt and ground lines on Pins 25 and 26 are connected to the buses on your card. You just install decoupling capacitors between ICs and then connect these lines to feed those ICs. If you need other voltages, they are on Pins 33, 34, and 50. Current limits per card are determined by the connecting paths. For the entire set of peripheral cards, the maximum current for + 5 V is 500 ma, for − 5 V is 200 ma, for − 12 V is 200 ma, and for + 12 V is 250 ma. These figures are given for the IIe model; others, especially clones, may be different. The total power dissipation for any card should not exceed 1.5 watts, regardless.

On the bottom pins, towards the front, lies the data bus from Pin 42 (D_0) to Pin 49 (D_7). You can connect to a MOS memory or something

TOP VIEW
BACK OF P.C. BOARD

Fig. 8-7. Peripheral connector output
pins. (Courtesy Apple Computer, Inc.)

GND	26		25	+5 V
DMA IN	27		24	DMA OUT
INT IN	28		23	INT OUT
$\overline{\text{NMI}}$	29		22	$\overline{\text{DMA}}$
$\overline{\text{IRQ}}$	30		21	RDY
$\overline{\text{RES}}$	31		20	I/O STROBE
$\overline{\text{INH}}$	32		19	N.C.
−12 V	33		18	R/W
−5 V	34		17	A15
N.C.	35		16	A14
7M	36		15	A13
Q3	37		14	A12
Φ1	38		13	A11
USER 1	39		12	A10
Φ0	40		11	A9
DEVICE SELECT	41		10	A8
D7	42		9	A7
D6	43		8	A6
D5	44		7	A5
D4	45		6	A4
D3	46		5	A3
D2	47		4	A2
D1	48		3	A1
D0	49		2	A0
+12 V	50		1	I/O SELECT

FRONT OF P.C. BOARD

LOCATIONS J2 TO J12

Fig. 8-8. Hobby/Phototyping board A2B000IX. (Courtesy Apple Computer, Inc.)

else that presents a light load. If you need more than one LS load, then use a buffer like the LS245, which is tri-state and bi-directional.

The address bus is on the top pins, towards the front. The entire bus can be accessed from Pin 2 (A_0) to Pin 17 (A_{15}) and Pin 18 (R/W). Only memory and processor cards usually need the entire address bus. Common peripheral interfaces use Pin 41 (DS) to select hardware and peripheral chip registers and Pin 1 (I/O select) to select on-board memory. A separate select line on Pin 20 (I/O strobe) can enable additional on-board memory in the $C800.CFFF range. The use of these selects is shown in the following sections.

Many peripheral chips are designed to use interrupts. In addition, you may use them directly to get a special job done like step and trace debugging, taking memory *snapshots,* forcing RESETs, and so on. Each interrupt line − IRQ, NMI and RES − is available on Pins 30, 29, and 31. In addition, the bus gives you a special feature called daisy-chain interrupts.

This scheme is designed to handle interrupt contention among several cards. The line enters from the next highest slot on Pin 28 and must leave for the next lowest slot on Pin 23. If not used, connect Pin 28 to Pin 23. If used, Pin 28 signals allowed interrupts and Pin 23 allows interrupts by lower cards. Connect Pin 28 to enable your interrupts; connect Pin 23 to disable further interrupts whenever you generate an IRQ (or NMI).

Daisy-chain interrupts are seldom used. To allow for their use by other cards in your system, connect Pins 23 and 28 together.

Timing is available on Pins 35, 36, 37, 38, and 40. Of these, Phase Zero on Pin 40 is most often used because it goes low at the right time to enable data transfers. You simply connect it to the enables on your buffers, peripheral chips, or whatever needs a ground-enabled input to transfer data with the data bus. The exact phase of this line varies between the old Apple IIs and the Apple IIe so production boards must be tested in both. The old Apple II bus gives a slightly later falling edge. The other clock lines provide for special timing.

In addition to several clock lines, there are other special features on the Apple bus. A daisy chain from Slot 7 down to Slot 1 called DMA for direct memory access will inhibit the 6502 from accessing the memory by signaling the cards of lower priority, just like the interrupt daisy chain. The DMA line at Pin 22 is used when this occurs. To ignore this feature, you should connect Pin 27 (DMA in) to Pin 28 (DMA out). Don't connect anything to Pin 22 (DMA).

The RDY line from the 6502 appears at Pin 21. It will halt the processor during Phase One when pulled low. Rarely used. The inhibit line

on Pin 32 (INH) disables motherboard memory access. Also rarely used. These two lines are used by fancy cards having on-board processors and memory.

Pin 39 is interesting. On Apples before the IIe, it is called USER1 and connects to the LS138 that performs the I/O select of slot-dependent memory. Rarely used, it would disable the memory selection when pulled low. A jumper called USER1 on the motherboard can connect or disconnect the line between the LS138 and the peripheral bus. Rarely used, it appears to serve little purpose.

On the Apple IIe, the Pin 39 line carries S.O. from the processor. Known as *sync output,* S.O. goes high whenever the 6502 does an operand fetch. It can be used to generate an interrupt, preferably an NMI, each instruction to provide a "bullet-proof" single-step debugger. You need some switching on your card to do this; the Sym-1 uses this method very nicely. On the old Apple, S.O. is hardwired to ground and is just not available.

Finally, Pin 19, which is not connected on Slots 1 to 6, carries the video sync signal to Slot 7. It has a fan out of two LS loads. The fan outs and fan ins of the bus are given for all other pins in Table 8-7.

8.2.2 Simple I/O Ports

One of the simplest I/O port systems you can make on a peripheral card is shown in Fig. 8-9. Eight output ports are connected to LEDs and eight input ports are connected to switches. By throwing the switches and observing the LEDs, you can see the ports work with simple software routines. When you get everything working okay, you can then replace the switches and LEDs with other I/O devices as you wish.

To build it, use the parts list of Table 8-8. Wire wrap is the easiest method to use. Some soldering will be needed as well.

Mount the sockets on the board with Silicone Seal™ or hot glue. Install the 0.05 μF capacitors between the ICs and across the power bus for decoupling. One of these should be as close to the plug pins as possible. Using wire wrap, connect the ground and five-volt lines to the pins on the IC sockets as given in Table 8-9. Jumper the Apple bus daisy chains: connect Pin 24 and Pin 27 together; connect Pin 23 and Pin 26 together.

The remainder of the wiring appears on the schematic of Fig. 8-9. Use it to complete the wiring. You may have to test your LEDs for

Table 8-7. Peripheral Loading and Driving Rules
(Courtesy Apple Computer, Inc.)

Pin Number	Name	Required Drive	Maximum LSTTL Load
1	I/O SELECT	N/A	10
2-17	A_0-A_{15}	Tri-State Buffer	5
18	R/$\overline{W}$	Tri-State Buffer	10
19	N/C	N/A	N/A
20	I/O STROBE	N/A	2
21	RDY	Open Collector	N/A
22	DMA	Open Collector	N/A
23	INT OUT	4 LSTTL	N/A
24	DMA OUT	4 LSTTL	N/A
25	+5V	N/A	N/A
26	GND	N/A	N/A
27	DMA IN	N/A	4
28	INT IN	N/A	4
29	NMI	Open Collector	N/A
30	IRQ	Open Collector	N/A
31	RES	N/A	2
32	INH	Open Collector	N/A
33	−12V	N/A	N/A
34	−5V	N/A	N/A
35	N/C	N/A	N/A
36	7M	N/A	2
37	Q3	N/A	2
38	Ø1	N/A	2
39	USER 1	N/A	N/A
40	Ø0	N/A	2
41	DEVICE SELECT	N/A	10
42-49	D_0-D_7	Tri-State Buffer	1
50	+12V	N/A	N/A

polarity before wiring them in if you don't know them already. Otherwise, wiring is straightforward.

Here's how it works.

The data from the Apple bus is buffered by the bi-directional tri-state buffer, 74LS245. The direction is controlled from the R/W line and it is enabled by the DS line. This means that the bus is connected only when an address of $C0nx is given, where n is the slot number plus eight and x is any number, $0 to $F. Given such an address, if the R/W line is low then the direction of the 74LS245 is right to left; if it is high, the direction is left to right. This chip isolates the Apple data bus from the board's data bus that connects to the 74LS75s and 74LS244.

Fig. 8-9. Simple I/O port schematic.

Table 8-8. Parts List for Simple I/O Ports

Quantity	Description
1	74LS245 quad bus transceiver
1	73LS138 decoder: 3 to 8
2	74LS75 quad latch
1	74LS244 octal tri-state bus driver
1	74LS04 hex inverter
2	20-pin DIP wire-wrap sockets
5	16-pin DIP wire-wrap sockets
1	14-pin DIP wire-wrap socket
1	16-pin DIP header
1	8xSPST DIP switch
8	Standard red LEDs
8	220 ohm resistors
8	2200 ohm resistors
6	10.05 μF disk capacitors
1	Apple hobby/prototyping board

NOTE: See Fig. 8-9.

Table 8-9. TTL Power Pinouts on the Simple I/O Ports

Type 74LS	Ground pin	+5 volt pin
04	7	14
75	12	5
138	8	16
244	10	20
245	10	20

This internal data bus won't load the Apple bus. You can connect several loads to it in LS type chips if you wish. Here, we have a set of eight inputs and a set of eight outputs connected. The inputs come from a 74LS244 that is a unidirectional buffer. Here, it works from right to left. The inputs are switches but could be any other device that is capable of driving LS logic. The switch level is gated through to the internal data bus whenever Pins 1 and 19 on the 74LS244 go low.

The internal bus also carries output data to the D-latches in the two 74LS75s. Whenever Pins 4 and 13 go high, the inputs to the latches are used to set or clear the outputs. By using latches, the output appears

constant even after the data from the bus disappears. A LED is lighted whenever an output goes low; dark whenever it goes high.

Address decoding is done with 74LS138 chips. On the motherboard, one of these feeds the peripheral I/O slots by decoding the address bus to one of seven lines. Each line is a separate DS for each slot. This way, each slot can use DS to enable its own hardware with very little further decoding to be done.

On this card, the 74LS138 completes the address decoding. It is enabled whenever DS goes low. Another enable is connected from an inverter on clock Phase One to provide timing. The three lines on Pins 1, 2, and 3 of the 74LS138 provide a three bit address to bring one of eight output lines low. To get the address, the R/W line and address lines A^0 and A^1 are used here. This generates different outputs for reads and writes for the various combinations of A_0 and A_1. Only two of these eight combinations are used; one of these selects the output port and one selects the input port. The actual addresses that must be used to reach the ports are summarized in Table 8-10.

Table 8-10. Simple Port Device Selection

Instruction	A1	A0	R/W	Yn	Pin
LDA $C080,X	H	H	H	0	15
STA $C080,X	H	H	L	1	14
LDA $C081,X	H	L	H	2	13
STA $C081,X	H	L	L	3	12
LDA $C082,X	L	H	H	4	11
STA $C082,X	L	H	L	5	10
LDA $C083,X	L	L	H	6	9
STA $C083,X	L	L	L	7	7

There are eight possible combinations of R/W, A_0, and A_1 lines. Each combination brings a different 74LS138 line low; if a routine has sixteen times the slot number in the X-reg, $s0$, then the instructions given will address the board. Each address selects a different Yn, 0 to 7, as the enable output from the 74LS138. Y0 appears on Pin 15 and selects the output port in the schematic. Y1 appears on Pin 14 and selects the input port. Y2 to Y7 are unused here, so $C081.C083 will have no effect.

Look at a couple of test examples. Suppose you put the board in Slot Five. The addresses $C0D0.C0DF *belong* to that slot since $C0D0 is $C080 + $50. If you write to the board with a Monitor command

C0D0:AA

you should see the LEDs make a pattern with every second one lighted. $AA is 10101010 in binary. Write other patterns to test the output port.

Similarly, test the input port by reading from the Monitor with:

C0D0

The byte read from that location should match the bit pattern of the eight switches on the board. Convert the hex number to binary and match up the pattern. Make different patterns and test until you are sure the input port reads exactly what you expect.

Instead of switches and lights, you can connect other devices to your I/O port. Make sure that it works okay with the switches and lights first. Then, you can either replace them with lines from the device or you can use the remaining enables from the 74LS138.

Replacement is the simplest. If you have an input device like an analog-to-digital converter, simply unhook the switch lines from the 74LS244 and wire up your new lines. For output disconnect the LEDs from the 74LS75s and connect any output lines you want latched. If you don't want latching on output, use the internal board data bus — Pins 2 to 8 on the 74LS245. Without latching, be sure to use the enable from Pin 15 of the 74LS138 to strobe it at the enable-low pin of your device. To enable high, use Pin 4 of the 74LS04 inverter.

For cases where a strobe is required to enable the external device, use one of the other lines on the 74LS138 as indicated in Table 8-10. This way you can leave the LEDs and switches at $C080,X alone and assign one of the remaining three addresses. Simply use the internal data bus from the 74LS245 Pins 2 to 9 for data.

8.2.3 Peripheral Interface Adaptor

Another way of making a simple I/O port is to use a peripheral interface adaptor chip, PIA for short. The most common of these is the Motorola 6821. Equivalent types you may see in the literature are the 6820 and the 6520. What this chip does is provide two ports of eight data bits and two control bits each. It has logic you can control by addressing. It provides processor bus interfacing with eight bits of data and two IRQ lines. In addition, it has several enables, allowing it

to be selected by several addressing schemes. By coming in a single, inexpensive package, it is often the way to go when making up simple interface cards for peripheral devices.

Hookup of the 6821 PIA to the Apple bus is quite simple. Fig. 8-10 shows the connections. If you have only the one IC to connect to the

Fig. 8-10. The 6821 to Apple II interface.

data bus on the card then you don't need an internal data bus; simply connect the data lines directly as indicated by the dashed lines. However, if you plan to add other data bus devices later, such as memory, then you should buffer with a 74LS245 to provide an internal data bus, just like the simple I/O port of Fig. 8-9 shown earlier. Use R/W on the direction pin and enable it to ground.

The rest of the pinout is straightforward. There are two IRQs from the 6821, IRQA and IRQB, each from its corresponding port. Simplest thing to do is connect both to the Apple II IRQ line to make them available for future programming. The reset line (RES) must be connected since it clears the registers in the PIA at power up. Without resetting, the PIA may generate unintentional interrupts!

Addressing is used to select one of four register locations at Pins 35 and 36 on the 6821. The device select (DS) line connected to the chip select (CS3) addresses these registers at the four locations given in Table 8-11. Each of the two ports has two locations, data and control.

Table 8-11. PIA Register Select Addresses

Address	Register
$C080,X	Data register A
$C081,X	Control register A
$C082,X	Data register B
$C083,X	Control register B

NOTE: Where X-reg contains 16 times slot number.

By reading and writing to these locations from the Apple, you can send and receive data from the ports, and control such things as interrupts and control lines handshaking protocol.

Like the simple I/O port, you can connect devices. Use either Port A or Port B data lines for most simple applications. Each line on each port can be set independently for input or output. However, if you plan to use the control lines in future for handshaking, use Port A for input and Port B for output. While the two ports are identical in their data handling logic, they differ in their control logics. See the references for details on use of control lines, interrupts, and handshaking.

Here, you can see how to control the transfer of data with a PIA.

Each of the two data locations belongs to one port. And, each of the two control locations contains the control register for each port. See Fig. 8-11 for PIA data and control registers for Port A. There are

ORA

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

DDRA

DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0

CRA

IRQ1	IRQ2	OUT	NEG	IRQE	DDR	NEG	IRQE

CA2   CA1

Fig. 8-11. PIA data and control registers for port A.

two registers for data, ORA and DDRA. Data are transferred in ORA and in the direction specified by DDRA. ORA is called the output register of Port A while DDRA is called the data direction register of Port A. Both ORA and DDRA reside at the same location, but more about that a little later.

Upon reset, the data location contains ORA. All lines of data, PA0 to PA7, are connected to ORA as inputs. By reading at the data location — $C800,X — you get the eight bits representing the eight data input lines at the time of the read. Input is not latched; it follows the data lines as they change from cycle to cycle.

The reason all bits of ORA are inputs is because the other data register, DDRA, was cleared to zero by the reset. If any bit in DDRA is changed to a one, then the corresponding bit in ORA becomes an output. For example, if you changed bit 3, DA3, in DDRA to a one, then bit 3, PA3, in ORA would become an output bit. The line PA3 would be an output line. The remaining lines would not be affected; they would remain as inputs.

To change bits in the data direction register, you have to switch the data location from ORA to DDRA. Then, you can change the DDRA contents. Immediately afterwards, you would want to switch the data location back from DDRA to ORA to access the port. This switching is done in the control register. You use bit 2, called DDR, in CRA. Here is the code:

```
LDA  $C081,X    get CRA
ORA  #$04       turn on Bit 2 (DDR)
STA  $C081,X    replace CRA
LDA  $C080,X    get DDRA
ORA  #$08       turn on Bit 3 (output)
STA  $C080,X    replace DDRA
LDA  $C081,X    get CRA
AND  #$FB       turn off Bit 2 (DDR)
STA  $C081,X    replace CRA
```

Remember, the X-reg contains sixteen times the slot number.

When a line is set to output, it is latched by ORA, the output register. So, after you write to the data register, the output appears on any output lines and remains there until you change it again.

To summarize, inputs are zero in the DDR and unlatched in the output register. Outputs are the ones in the DDR and latched in the output register.

To use Port B, the rules are the same. Use $C082,X for
ORB/DDRB and $C083 for CRB. The layout is given in Fig. 8-12;

ORB

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0

DDRB

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0

CRB

IRQ1	IRQ2	OUT	NEG	IRQE	DDR	NEG	IRQE

CB2   CB1

Fig. 8-12. PIA data and control registers for port B.

compare to Fig. 8-11. As an example, here is how to set all eight data
lines of Port B for output:

```
LDA   $C083,X     get CRB
ORA   #$04        turn on Bit 2 (DDR)
STA   $C083,X     replace
LDA   #$FF        turn on all bits
STA   $C082,X     in DDRB
LDA   $C083,X     get ORB again
AND   #$FB        turn off Bit 2
STA   $C083,X     replace
```

To use the ports, then, you could read from Port A by

LDA $C080,X

and write to Port B by

STA $C082,X

assuming you initialized Port B for output as just shown. Always set
the X-reg to sixteen times the slot number before using any of these
statements.

8.2.4 Peripheral Memory

While you can get peripherals working from routines in main memory, there are cases where you need memory right on the same card as the interface circuitry itself. The most obvious is the need to determine the slot number of the device. An on-card routine can do this easily. Another is the need to use Apple's input/output protocol, the hooks. You can send and receive byte streams easily from BASIC or many software packages if the routine is at C\underline{s}$00, where $\underline{s}$ is the slot number. Review Section 6.1 for how the hooks work.

If you decide to add memory to your peripheral card, the question is, RAM or ROM? Traditionally, ROM is used on cards so that the routine is permanent. No initial loading is needed and any system can pass bytes with no need to have the routine on a special disk. Choose an EPROM like the 2316. You will need a PROM programmer; use an Apple card. Several are available from manufacturers. You can save a bit by making your own eraser.

To make a PROM eraser, get an 18-inch fluorescent lamp that produces short ultraviolet. They are sold by lamp suppliers as germicidal lamps. Don't use so-called "black light" tubes. They are cheaper, but put out long wavelength ultraviolet, not the short wavelength the EPROM needs to erase. The lamp should be labeled as producing 253.7 nanometers, a mercury line in the short ultraviolet. Fig. 8-13 gives the circuit.

Fig. 8-13. EPROM eraser circuit.

If you are modifying an existing fixture, you have to do two things. First, put the fixture in a "bottomless" box that rests on a table or floor so as to cover and illuminate the EPROMs. Locate the start switch outside the box so that you can start it without looking at the light itself. The short ultraviolet is dangerous to eyesight. Next, use a

normally open pushbutton to turn the lamp off whenever the box is lifted from its resting surface. This is your safety switch to prevent accidental eye exposure to the lamp.

Such a homemade eraser lamp is safer and cheaper than most commercially available lamps.

Alternately, you can use RAM on a card. The advantages include speed and ease of programming and no initial costs for PROM programming equipment. You have to load the card before using it, but you can also modify your routines when necessary.

The easiest RAM to use is a static RAM. A low-cost static RAM is the type 2114. A circuit that adapts two of these chips to the Apple bus is shown in Fig. 8-14. The buffer used to isolate the internal data bus is not shown, but is the same as described. The 74LS245 buffer can be enabled permanently by grounding Pin 19 so that it will work for both device addresses and I/O (memory) addresses. Connect the direction switch at Pin 1 to Apple R/W line.

Regardless of RAM or ROM, you will write the routine starting at $Cs00 to either input or output one byte. You have 256 bytes of program space there to do that. For simple devices, that is usually sufficient.

If you want RAM storage for information between calls, use the scratchpad memory in SCREEN1 ($0400.07FF) as assigned to your slot. See Chapter Two for a breakdown. This is where you keep setup parameters the first time they are called, and use them on subsequent calls. If the slot number is kept in the Y-reg, then you address this scratchpad as $0478,Y and $04F8,Y and so on. See Table 2-3 for the others.

Remember, you need sixteen times the slot number in the X-reg to address the device hardware as described in Sections 8.2.2 and 8.2.3.

Here is how to get the slot number of a routine running in $Cs00.CsFF:

```
JSR    $FF58        a known RTS
TSX
LDA    $0100,X      gets our PC-high
AND    #$0F         isolates s
TAY                 slot number in Y-reg
ASL    A
ASL    A
```

Fig. 8-14. Static RAM on a peripheral card.

```
ASL   A
ASL   A
TAX              $10*slot in X-reg
```

Do this after pushing the registers onto the stack at the beginning of the routine. The slot number in the Y-reg can reach the proper scratch-pad RAM and the X-reg can reach the proper device by $C08n,X addressing.

By making your own special-purpose peripherals and by programming their routines on-board, you can transform your Apple into any number of different "custom-built" computers. Where does your imagination lead you at this point? Make it on your Apple!

APPENDIX A

Bibliography and Notes

In addition to the material in this book, you may wish to consult the following on particular points. They can give you more detail on specific topics whenever you must dig a little deeper on any project.

Apple Computer, Inc., publish definitive reference material on the Apple II. In particular, you may wish to consult one of the following:

Apple II Reference Manual (1978 ed.). A collection of engineering notes including the Standard Monitor and Sweet16 source listings, very clear schematics and the instruction set for Integer BASIC as it appears in Chapter Five of this book. Out of print now.

Apple II Reference Manual (1979 ed.). A *real* reference manual, this is the standard reference for Apple II before the IIe model — Standard and Autostart Monitors listed.

Apple IIe Reference Manual (1982 ed.). Two volumes, the second containing the listings of the IIe Monitor and 80-column firmware. Very thorough.

Applesoft BASIC Programmer's Reference Manual (1982 ed.). 2 volumes, *for IIe only*. An expanded version of the earlier, excellent Applesoft manual, this one emphasizes IIe features. With care, it can be used on the older Apple II models — try out any feature first.

Apple II Product Specification — Hobby/Prototyping Board (Product Code A2B0001X). This useful collection of bus inter-facing information comes with to bare board from Apple. Nice to have if you design peripheral cards.

Dougherty, *The Apple II Monitors Peeled* (latest ed.) is a complete description of the Standard and Autostart Monitors. Very exhaustive.

Programmer's Aid #1 Installation and Operating Manual expands on the features you can use with Integer BASIC configurations. Main features are highlighted in Chapter Five in this book.

Apple Software Bank — Contributed Programs Volumes 3-5 is a collection of freebee software documentation. Includes File Cabinet, a LISPer, and other goodies that were distributed by Apple. Try your local Apple user group; they may have the software in their library.

Disk Operating System Instructional and Reference Manual (latest version 3.3). Highlighted in Chapter Seven, but contains more use-ful material, especially at the command level.

You can get Apple publications through local dealers. The address of the *orchard* head office is

Apple Computer, Inc.,
20525 Mariani Avenue
Cupertino, CA 95014

Apple Pugetsound Program Library Exchange is a user group with a large mail-in membership. Their magazine, which is called A.P.P.L.E., is distributed to members. They distribute software and documentation; in particular:

The Wozpak II and Other Assorted Goodies. Supplied from Apple, it contains original Wozinak material on the Apple II and Integer BASIC goodies developed in the *early years* of the Apple. A must for Integer BASIC freaks, it comes with software.

Program Line Editor written by Neil Konzen. This manual and software is a must for anyone doing extensive BASIC program-ming. Integer and Applesoft versions come on disk.

Write them for current membership information. There are many more useful utilities you can get from them:

Apple Pugetsound Program Library Exchange
6708 39th Avenue SW,
Seattle, WA 98136

Crossley, John, "Applesoft Internal Entry Points," *Apple Orchard,* pp 12-14, published by International Apple Core, P.O. Box 976, Daly City, CA 94017. Vol. 1, no. 1 (Mar., Apr., 1980). A collection of Applesoft locations and call descriptions. The first large collection published, this is the one Applesoft books are largely based upon.

Coan, James A., *Basic APPLE BASIC,* Hayden Book Company, Inc., Rochelle Park, New Jersey. For anyone who has little or no Applesoft programming experience, this book takes you through BASIC from the beginning. May be used for both Applesoft and Integer BASIC instruction.

Gayler, Winston D., *The Apple II Circuit Description,* Howard W. Sams & Co., Inc. (1983). Very effective reference if you expect to do much work with Apple II hardware. Also great for troubleshooting clones.

Intel Component Data Catalog (latest edition), good source for PROM data. Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051.

Irwin, Paul, "Amper Jump & TSort," *Nibble,* vol. 2, no. 6 (1981). A simple method for using several routines with ampersand calls is given. Also, a tag sort for strings is given and described in detail.

Ibid, "Amp-L-Soft", *Nibble,* vol. 3, no. 7 (1982). More ampersand call goodies with notes on loading ampersand routines to run with Applesoft. Tones, an INPUT anything, and a fast substring search are included.

Lancaster, Don, *TTL Cookbook,* Howard W. Sams & Co., Inc. (1974). Excellent introduction to TTL chips for would-be hardware hackers.

Ibid, *Son of Cheap Video,* Howard W. Sams & Co., Inc. (1980). The last two chapters describe a lower-case video and keyboard scheme.

Leventhal, Lance, *6502 Assembly Language Programming,* Osbourne/McGraw-Hill, Inc. (1979). Although the unexplained Assembler directives will discourage the beginning programmer, this is quite a thorough reference for anything you may care to look up in the way of 6502 features and routines.

Luebbert, William F., *What's Where in the Apple?* (latest edn.) from Micro Ink, Inc., 34 Chelmsford Street, P.O. Box 6502, Chelmsford, Mass. 01824, the people who publish *Micro* magazine. This is a large gazetteer of the Apple II that covers both Applesoft and Integer configurations.

MC6500 Microcomputing Family Programming Manual, Jan. 1976. Published jointly by MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401 and by Synertek, P.O. Box 552, MS/34, Santa Clara, CA 95052. When it comes to programming the 6502, this is the definitive work by the designers themselves.

MC6500 Microcomputing Family Hardware Manual, 1976. Also published jointly by MOS and Synertek, this describes and explains how to design with 6500 series hardware: 6502 and 6520 chips in particular. Remembr, the 6520 can be had in a later product called the 6821.

Motorola Microprocessors Data Manual, latest edition, from Literature Distribution Center, Motorola Semiconductor Products Inc., P.O. Box 20924, Phoenix, AZ 85036. This gives the 6800 family of processors and peripherals. These peripheral chips work on the Apple II bus.

Pump, Mark, "DOS Internals: An Overview," *Call − A.P.P.L.E.,* (Feb. 1981). One of the best dissections of DOS ever written, it covers versions 3.1, 3.2, and 3.3.

Radio Shack, *Semiconductor Reference Guide* (current ed.). A collection of data on the products carried by Radio Shack. Many

popular items — memory, transducers, transistors, TTL, and so forth. Radio Shack Cat. #276-4006.

Synertek (current year) *Data Catalog,* Synertek, P.O. Box 552 MS/34, Santa Clara, CA 95052. Lots of 6500 series data, memories, especially the type 2114 static RAM.

The TTL Data Book for Design Engineers, latest edition, from Semiconductor Group, Texas Instruments, Inc., P.O. Box 225012, Dallas, TX 75265. This is the *bible* of the industry — look up any of the TTL chips you may likely use here. If you don't know TTL, get Lancaster's book as well.

Lechner, Pieter, Worth, Don, *Beneath Apple DOS,* (1981), from Quality Software, 6660 Resenda Blvd., Resenda, CA 91335. Just about everything you wanted to know about DOS.

APPENDIX B

Apple II Programmers' Reference Card

MONITOR COMMANDS SUMMARY

Examine Memory

addr	Displays single location.
addr1.addr2	Displays a block of memory.
(*return*)	Displays next 8 locations.
*addr1<addr2.addr3*V	Verifies that block (*addr2.addr3*) equals the block beginning at (*addr1*).
*addr*L	List (disassemble) locations from (*addr*).
L	List beginning at next location.

Change Memory

addr:byte byte	Change contents starting at (*addr*).
:byte byte byte	Change contents starting at next location.
*addr1<addr2.addr3*M	Move block from (*addr2.addr3*) to (*addr1*).

NOTE: To set a block to all single byte value (e.g., all zeros), use two commands as follows.   *addr1:byte*
      (*addr1 + 1*)<*addr1.*(*addr2-1*)M

Cassette Tape

*addr1.addr2*W	Write block of memory to tape.
*addr1.addr2*R	Read block from tape into memory.

Apple Video Display

N	Set to normal white-on-black.
I	Set to inverse black-on-white.

Calculate Hexadecimal

byte1 + byte2	Add hex numbers.
byte1 − byte2	Subtract hex numbers.

HEX	HEX	HIGH	Applesoft	ASCII	OP CODE	FLAGS	HEX
5D	93	23808		REM	EOR m,X	-V-BDI-C	5D
5E	94	24064	↑	LET	LSR m,X	-V-BDIZ-	5E
5F	95	24320	←	GOTO		-V-BDIZC	5F
60	96	24576	`	IF	RTS	-VI-----	60
61	97	24832	a	PRINT	ADC (z,X)	-VI----C	61
62	98	25088	b	PRINT		-VI---Z-	62
63	99	25344	c	PRINT		-VI---ZC	63
64	100	25600	d	POKE		-VI--I--	64
65	101	25856	e	,	ADC z	-VI--I-C	65
66	102	26112	f	COLOR =	ROR z	-VI--IZ-	66
67	103	26368	g	PLOT		-VI--IZC	67
68	104	26624	h	,	PLA	-VI-D---	68
69	105	26880	i	HLIN	ADC #v	-VI-D--C	69
6A	106	27136	j	,	ROR A	-VI-D-Z-	6A
6B	107	27392	k	AT		-VI-D-ZC	6B
6C	108	27648	l	VLIN	JMP (m)	-VI-DI--	6C
6D	109	27904	m	,	ADC m	-VI-DI-C	6D
6E	110	28160	n	AT	ROR m	-VI-DIZ-	6E
6F	111	28416	o	VTAB		-VI-DIZC	6F
70	112	28612	p	=	BVS	-VIB----	70
71	113	28928	q	=	ADC (z),Y	-VIB---C	71
72	114	29184	r	)		-VIB--Z-	72
73	115	29440	s			-VIB--ZC	73
74	116	29696	t	LIST		-VIB-I--	74
75	117	29952	u	,	ADC z,X	-VIB-I-C	75
76	118	30208	v	LIST	ROR z,X	-VIB-IZ-	76
77	119	30270	w	POP		-VIB-IZC	77
78	120	30464	x	NODSP	SEI	-VIBD---	78
79	121	30976	y	NODSP	ADC m,Y	-VIBD--C	79
7A	122	31232	z	NOTRACE		-VIBD-Z-	7A
7B	123	31488	{	DSP		-VIBD-ZC	7B
7C	124	31744	|	DSP		-VIBDI--	7C
7D	125	32000	}	TRACE	ADC M,X	-VIBDI-C	7D
7E	126	32256	~	PR#	ROR m,X	-VIBDIZ-	7E
7G	127	32512	DEL	IN#		-VIBDIZC	7F

HEX	HEX	HIGH	Applesoft	ASCII	OP CODE	FLAGS	HEX
80	128	32768	END	NUL	STA (z,X)	N-------	80
81	129	33024	FOR	SOH		N------C	81
82	130	33280	NEXT	STX		N-----Z-	82
83	131	33536	DATA	ETX		N-----ZC	83
84	132	33792	INPUT	EOT	STY z	N----I--	84
85	133	34048	DEL	ENQ	STA z	N----I-C	85
86	134	34304	DIM	ACK	STX z	N----IZ-	86
87	135	34560	READ	BEL		N----IZC	87
88	136	34816	GR	BS	DEY	N---D---	88
89	137	35072	TEXT	HT		N---D--C	89
8A	138	35328	PR#	LF	TXA	N---D-Z-	8A
8B	139	35584	IN#	VT		N---D-ZC	8B
8C	140	35840	CALL	FF	STY m	N---DI--	8C
8D	141	36096	PLOT	CR	STA m	N---DI-C	8D
8E	142	36532	HLIN	SO	STX m	N---DIZ-	8E
8F	143	36608	VLIN	SI		N---DIZC	8F
90	144	36864	HGR2	DLE	BCC	N--B----	90
91	145	37120	HGR	DC1	STA (z),Y	N--B---C	91
92	146	37376	HCOLOR =	DC2		N--B--Z-	92
93	147	37632	HPLOT	DC3		N--B--ZC	93
94	148	37888	DRAW	DC4	STY z,X	N--B-I--	94

95	149	38144	XDRAW	NAK	STA z,X	N--B-I-C	95
96	150	38400	HTAB	SYN	STX z,Y	N--B-IZ-	96
97	151	38656	HOME	ETB		N--B-IZC	97
98	152	38912	ROT =	CAN	TXA	N--BD---	98
99	153	39168	SCALE =	EM	STA m,Y	N--BD--C	99
9A	154	39424	SHLOAD	SUB	TXS	N--BD-Z-	9A
9B	155	39680	TRACE	ESC		N--BD-ZC	9B
9C	156	39936	NOTRACE	FS		N--BDI--	9C
9D	157	40192	NORMAL	GS	STA m,X	N--BDI-C	9D
9E	158	40448	INVERSE	RS		N--BDIZ-	9E
9F	159	40704	FLASH	US		N--BDZIC	9F
A0	160	40960	COLOR =	SP	LDY #v	N-1-----	A0
A1	161	41216	POP	!	LDA (z,X)	N-1----C	A1
A2	162	41472	VTAB	"	LCX #v	N-1---Z-	A2
A3	163	41728	HIMEM:	#		N-1---ZC	A3
A4	164	41984	LOMEM:	$	LDY z	N-1--I--	A4
A5	165	42240	ONERR	%	LDA z	N-1--I-C	A5
A6	166	42496	RESUME	&	LDX z	N-1--IZ-	A6
A7	167	42752	RECALL	'		N-1--IZC	A7
A8	168	43008	STORE	(	TAY	N-1-D---	A8
A9	169	43264	SPEED =	)	LDA #v	N-1-D--C	A9
AA	170	43520	LET	*	TAX	N-1-D-Z-	AA
AB	171	43776	GOTO	+		N-1-D-ZC	AB
AC	172	44032	RUN	,	LDY m	N-1-DI--	AC
AD	173	44288	IF	–	LDA m	N-1-DI-C	AD
AE	174	44544	RESTORE	.	LDX m	N-1-DIZ-	AE
AF	175	44800	&	/		N-1-DIZC	AF
B0	176	45056	GOSUB	0	BCS	N-1B----	B0
B1	177	45312	RETURN	1	LDA (z),Y	N-1B---C	B1
B2	178	45568	REM	2		N-1B--Z-	B2
B3	179	45824	STOP	3		N-1B--ZC	B3
B4	180	46080	ON	4	LDY z,X	N-1B-I--	B4
B5	181	46336	WAIT	5	LDA z,X	N-1B-I-C	B5
B6	182	46592	LOAD	6	LDX z,Y	N-1B-IZ-	B6
B7	183	46848	SAVE	7		N-1B-IZC	B7
B8	184	47104	DEF FN	8	CLV	N-1BD---	B8
B9	185	47360	POKE	9	LDA m,Y	N-1BD--C	B9
BA	186	47616	PRINT	:	TSX	N-1BD-Z-	BA
BB	187	47872	CONT	;		N-1BD-ZC	BB
BC	188	48128	LIST	<	LDY m,X	N-1BDI--	BC
BD	189	48384	CLEAR	=	LDA m,X	N-1BDI-C	BD
BE	190	48640	GET	>	LDX m,Y	N-1BDIZ-	BE
BF	191	48896	NEW	?		N-1BDIZC	BF
C0	192	49152	TAB	@	CPY #v	NV------	C0
C1	193	49408	TO	A	CMP (z,X)	NV-----C	C1
C2	194	49664	FN	B		NV----Z-	C2
C3	195	49920	SPC(	C		NV----ZC	C3
C4	196	50176	THEN	D	CPY z	NV---I--	C4
C5	197	50432	AT	E	CMP z	NV---I-C	C5
C6	198	50688	NOT	F	DEC z	NV---IZ-	C6
C7	199	50944	STEP	G		NV---IZC	C7
C8	200	51200	+	H	INY	NV--D---	C8
C9	201	51456	–	I	CMP #v	NV--D--C	C9
CA	202	51712	*	J	DEX	NV--D-Z-	CA
CB	203	51968	/	K		NV--D-ZC	CB
CC	204	52224		L	CPY m	NV--DI--	CC
CD	205	52480	AND	M	CMP m	NV--DI-C	CD
CE	206	52736	OR	N	DEC m	NV--DIZ-	CE
CF	207	52992	>	O		NV--DIZC	CF
D0	208	53248	=	P	BNE	NV-B----	D0

Test and Debug

*addr*G	Go, executes routine at *(addr)*.
ctrl/Y	User, executes routine at $03sF8.
ctrl/E	Examine registers. Change them with a colon on the next command — : *(A) (X) (Y) (P) (S)* — for as many registers as wanted
*addr*T*	Trace routine at *(addr)* until BRK (op code $00) is executed.
*addr*S*	Single step by executing one instruction only at *(addr)*. Step following instructions by "S" only for each.

NOTE: Trace and step are available in Old Monitor only.

Mini-Assembler

CALL – 151	Enter Monitor from Integer BASIC only.
F666G	Enter Miniassembler from Monitor.
addr:instruction	Assembles *instruction* with mnemonic at *(addr)*.
(space)instruction	Assembles *instruction* at next location.
$*command*	Executes any Monitor command.
$FF69G	Return from Miniassembler to Monitor.

HEX	LOW	HIGH	ASCII	INTEGER BASIC	OPCODE	FLAGS	HEX
00	0	0	NUL	start line	BRK	--------	00
01	1	256	SOH	end line	ORA (z,X)	-------C	01
02	2	512	STX			------Z-	02
03	3	768	ETX	:		------ZC	03
04	4	1024	EOT	LOAD		-----I--	04
05	5	1280	ENQ	SAVE	ORA z	-----I-C	05
06	6	1536	ACK	CON	ASL z	-----IZ-	06
07	7	1792	BEL	RUN		-----IZC	07
08	8	2048	BS	RUN	PHP	----D---	08
09	9	2034	HT	DEL	ORA #v	----D--C	09
0A	10	2560	LF	,	ASL A	----D-Z-	0A
0B	11	2816	VT	NEW		----D-ZC	0B
0C	12	3072	FF	CLR		----DI--	0C
0D	13	3378	CR	AUTO	ORA m	----DI-C	0D
0E	14	3584	SO	,	ASL m	----DIZ-	0E
0F	15	3840	SI	MAN		----DIZC	0F
10	16	4096	DLE	HIMEM:	BPL	---B----	10
11	17	4352	DC1	LOMEM:	ORA (z),Y	---B---C	11
12	18	4608	DC2	+		---B--Z-	12
13	19	4864	DC3	–		---B--ZC	13
14	20	5120	DC4	*		---B-I--	14
15	21	5376	NAK	/	ORA z,X	---B-I-C	15
16	22	5632	SYN	=	ASL z,X	---B-IZ-	16
17	23	5888	ETB	#		---B-IZC	17
18	24	6144	CAN	>=	CLC	---BD---	18
19	25	6400	EM	>	ORA m,Y	---BD--C	19
1A	26	6656	SUB	<=		---BD-Z-	1A
1B	27	6912	ESC	<>		---BD-ZC	1B
1C	28	7168	FS	<		---BDI--	1C
1D	29	7424	GS	AND	ORA m,X	---BDI-C	1D
1E	30	7680	RS	OR	ASL m,X	---BDIZ-	1E
1F	31	7936	US	MOD		---BDIZC	1F
20	32	8192	SP	^	JSR m	--I-----	20

21	33	8448	!		AND (z,X)	--I----C	21
22	34	8704	"	(		--I---Z-	22
23	35	8960	#	,		--I---ZC	23
24	36	9216	$	THEN	BIT z	--I--I--	24
25	37	9472	%	THEN	AND z	--I--I-C	25
26	38	9728	&	,	ROL z	--I--IZ-	26
27	39	9984	'	,		--I--IZC	27
28	40	10240	(	"	PLP	--I-D---	28
29	41	10496	)	"	AND #v	--I-D--C	29
2A	42	10752	*	(	ROL A	--I-D-Z-	2A
2B	43	11008	+			--I-D-ZC	2B
2C	44	11264	,		BIT m	--I-DI--	2C
2D	45	11520	–	(	AND m	--I-DI-C	2D
2E	46	11776	.	PEEK	ROL m	--I-DIZ-	2E
2F	47	12032	/	RND		--I-DIZC	2F
30	48	12288	0	SGN	BMI	--IB----	30
31	49	12544	1	ABS	AND (z),Y	--IB---C	31
32	50	12800	2	PDL		--IB--Z-	32
33	51	13056	3			--IB--ZC	33
34	52	13312	4	(		--IB-I--	34
35	53	13568	5	+	AND z,X	--IB-I-C	35
36	54	13824	6	–	ROL z,X	--IB-IZ-	36
37	55	14080	7	NOT		--IB-IZC	37
38	56	14336	8	(	SEC	--IBD---	38
39	57	14592	9	=	AND m,Y	--IBD--C	39
3A	58	14848	:	#		--IBD-Z-	3A
3B	59	15104	;	LEN(		--IBD-ZC	3B
3C	60	15360	<	ASC(		--IBDI--	3C
3D	61	15616	=	SCRN(	AND m,X	--IBDI-C	3D
3E	62	15872	>	,	ROL m,X	--IBDIZ-	3E
3F	63	16128	?	(		--IBDIZC	3F
40	64	16384	@	$	RTI	-V------	40
41	65	16640	A		EOR (z,X)	-V-----C	41
42	66	16896	B	(		-V----Z-	42
43	67	17152	C	,		-V----ZC	43
44	68	17408	D	,		-V---I--	44
45	69	17664	E	;	EOR z	-V---I-C	45
46	70	17920	F	;	LSR z	-V---IZ-	46
47	71	18176	G	;		-V---IZC	47
48	72	18432	H	,	PHA	-V--D---	48
49	73	18688	I	,	EOR #v	-V--D--C	49
4A	74	18944	J	,	LSR A	-V--D-Z-	4A
4B	75	19200	K	TEXT		-V--D-ZC	4B
4C	76	19456	L	GR	JMP m	-V--DI--	4C
4D	77	19712	M	CALL	EOR m	-V--DI-C	4D
4E	78	19968	N	DIM	LSR m	-V--DIZ-	4E
4F	79	20224	O	DIM		-V--DIZC	4F
50	80	20480	P	TAB	BVC	-V-B----	50
51	81	20736	Q	END	EOR (z),Y	-V-B---C	51
52	82	20992	R	INPUT		-V-B--Z-	52
53	83	21248	S	INPUT		-V-B--ZC	53
54	84	21504	T	INPUT		-V-B-I--	54
55	85	21760	U	FOR	EOR z,X	-V-B-I-C	55
56	86	22016	V	=	LSR z,X	-V-B-IZ-	56
57	87	22272	W	TO		-V-B-IZC	57
58	88	22528	X	STEP	CLI	-V-BD---	58
59	89	22784	Y	NEXT	EOR m,Y	-V-BD--C	59
5A	90	23040	Z	,		-V-BD-Z-	5A
5B	91	23296	[	RETURN		-V-BD-ZC	5B
5C	92	23552	\	GOSUB		-V-BDI--	5C

D1	209	53504	<	Q	CMP (z),Y	NV-B---C	D1
D2	210	53760	SGN	R		NV-B--Z-	D2
D3	211	54016	INT	S		NV-B--ZC	D3
D4	212	54272	ABS	T		NV-B-I--	D4
D5	213	54528	USR	U	CMP z,X	NV-B-I-C	D5
D6	214	54784	FRE	V	DEC z,X	NV-B-IZ-	D6
D7	215	55040	SCRN(	W		NV-B-IZC	D7
D8	216	55296	PDL	X	CLD	NV-BD---	D8
D9	217	55552	POS	Y	CMP m,Y	NV-BD--C	D9
DA	218	55808	SQR	Z		NV-BD-Z-	DA
DB	219	56064	RND	⊏		NV-BD-ZC	DB
DC	220	56320	LOG	\		NV-BDI--	DC
DD	221	56576	EXP	⊐	CMP m,X	NV-BDI-C	DD
DE	222	56832	COS	↑	DEC m,X	NV-BDIZ-	DE
DF	223	57088	SIN	—		NV-BDIZC	DF
E0	224	57344	TAN	\	CPX #v	NV1-----	E0
·E1	225	57600	ATN	a	SBC (z,X)	NV1----C	E1
E2	226	57856	PEEK	b		NV1---Z-	E2
E3	227	58112	LEN	c		NV1---ZC	E3
E4	228	58368	STR$	d	CPX z	NV1--I--	E4
E5	229	58624	VAL	e	SBC z	NV1--I-C	E5
E6	230	58880	ASC	f	INC z	NV1--IZ-	E6
E7	231	59136	CHR$	g		NV1--IZC	E7
E8	232	59392	LEFT$	h	INX	NV1-D---	E8
E9	233	59648	RIGHT$	i	SBC #v	NV1-D--C	E9
EA	234	59904	MID$	j	NOP	NV1-D-Z-	EA
EB	235	60160	↑	k		NV1-D-ZC	EB
EC	236	60416		l	CPX m	NV1-DI--	EC
ED	237	60672		m	SBC m	NV1-DI-C	ED
EE	238	60928		n	INC m	NV1-DIZ-	EE
EF	239	61184		o		NV1-DIZC	EF
F0	240	61440		p	BEQ	NV1B----	F0
F1	241	61696		q	SBC (z),Y	NV1B---C	F1
F2	242	61952	error	r		NV1B--Z-	F2
F3	243	62208	messages	s		NV1B--ZC	F3
F4	244	62464		t		NV1B-I--	F4
F5	245	62720		u	SBC z,X	NV1B-I-C	F5
F6	246	62976		v	INC z,X	NV1B-IZ-	F6
F7	247	63232		w		NV1B-IZC	F7
F8	248	63488		x	SED	NV1BD---	F8
F9	249	63744		y	SBC m,Y	NV1BD--C	F9
FA	250	64000		z		NV1BD-Z-	FA
FB	251	64256	↓	{		NV1BD-ZC	FB
FC	252	64512		|		NV1BDI--	FC
FD	253	64768	(	}	SBC m,X	NV1BDI-C	FD
FE	254	65024	(	~	INC m,X	NV1BDIZ-	FE
FF	255	65280	(	DEL		NV1BDIZC	FF

UNIQUE 6502 INSTRUCTIONS

MNEMONIC	OP CODE	ADDRESSING	FLAGS
branch			
BCC	90	relative	------
BCS	B0	relative	------
BEQ	F0	relative	------
BMI	30	relative	------
BNE	D0	relative	------
BPL	10	relative	------
BVC	50	relative	------
BVS	70	relative	------
p-register bit			
CLC	18	implied	-----C
CLD	D8	implied	--D---
CLI	58	implied	---I--
CLV	B8	implied	-V----
SEC	38	implied	-----C
SED	F8	implied	--D---
SEI	78	implied	---I--
program flow			
BRK	00	implied	---I--
JMP	4C	absolute	------
JMP	6C	indirect	------
JSR	20	absolute	------
NOP	EA		------
RTI	40	implied	stack*
RTS	60	implied	------
transfer			
TAX	AA	implied	N---Z-
TAY	A8	implied	N---Z-
TSX	BA	implied	N---Z-
TXA	8A	implied	N---Z-
TXS	9A	implied	-----
TYA	98	implied	N---Z-
stack			
PHA	48	implied	------
PHP	08	implied	------
PLA	68	implied	N---Z-
PLP	28	implied	stack*

*restored from stack

ACCUMULATOR, MEMORY, AND INDEX INSTRUCTIONS

	ACCUMULATOR/IMPLIED*	IMMEDIATE	ZERO PAGE	ZERO PAGE, X	ABSOLUTE	ABSOLUTE, X	ABSOLUTE, Y	INDIRECT, X	INDIRECT, Y	FLAGS
ADC	–	69	65	75	6D	7D	79	61	71	NVZC
AND	–	29	25	35	2D	3D	39	21	31	N-Z-
ASL	0A	–	06	16	0E	–	–	–	–	N-ZC
BIT	–	–	24	–	2C	–	–	–	–	76Z-
CMP	–	C9	C5	D5	CD	DD	D9	C1	D1	N-ZC
CPX	–	E0	E4	–	EC	–	–	–	–	N-ZC
CPY	–	C0	C4	–	CC	–	–	–	–	N-ZC
DEC	–	–	C6	D6	CE	DE	–	–	–	N-Z-
DEX	CA*	–	–	–	–	–	–	–	–	N-Z-
DEY	88*	–	–	–	–	–	–	–	–	N-Z-
EOR	–	49	45	55	4D	5D	59	41	51	N-Z-
INC	–	–	E6	F6	EE	FE	–	–	–	N-Z-
INX	E8*	–	–	–	–	–	–	–	–	N-Z-
INY	C8*	–	–	–	–	–	–	–	–	N-Z-
LDA	–	A9	A5	B5	AD	BD	B9	A1	B1	N-Z-
LDX	–	A2	A6	B6#	AE	–	BE	–	–	N-Z-
LDY	–	A0	A4	B4	AC	BC	–	–	–	N-Z-
LSR	4A	–	46	56	4E	5E	–	–	–	N-Z-
ORA	–	09	05	15	0D	1D	19	01	11	N-Z-
ROL	2A	–	26	36	2E	3E	–	–	–	N-ZC
ROR	6A	–	66	76	6E	7E	–	–	–	N-ZC
SBC	–	E9	E5	F5	ED	FD	F9	E1	F1	NVZC
STA	–	–	85	95	8D	9D	99	81	91	----
STX	–	–	86	96#	8E	–	–	–	–	----
STY	–	–	84	94	8C	–	–	–	–	----

*implied
#zero page, Y

N negative
V overflow
Z zero

C carry
6 V if bit 6
7 N if bit 7

Index

SAMS

More Books for Apple Owners!

INTRODUCING THE APPLE® MACINTOSH™
Introduces you to the design philosophy and physical structure of the Macintosh™, and explores its displays, keyboard, mouse, software, accessories, and more. By Connolly and Lieberman. 192 pages, 8 x 9¼, softbound. ISBN 0-672-22361-9. © 1984.
No. 22361 ....**$12.95**

APPLE® IIe PROGRAMMERS' REFERENCE GUIDE
An outstanding reference guide specifically for the IIe that makes needed facts, applications, and other technical information readily available at your fingertips. By David L. Heiserman. 416 pages, 5½ x 8½, comb-bound. ISBN 0-672-22299-X. © 1984.
No. 22299 ....**$19.95**

APPLESOFT FOR THE IIe
A detailed Applesoft programmer's reference manual written specifically for the IIe and covering all aspects of IIe syntax and programming techniques. By Blackwood and Blackwood. 368 pages, 6 x 9, comb-bound. ISBN 0-672-22259-0. © 1983.
No. 22259 ....**$19.95**

APPLE® PROGRAMMER'S HANDBOOK
This single-volume coverage of essential Apple data contains dozens of tested "stock" routines organized by topic, a detailed memory map, and much more. By Paul Irwin. 480 pages, 5½ x 8½, comb-bound. ISBN 0-672-22175-6. © 1984.
No. 22175 ....**$21.95**

BASIC TRICKS FOR THE APPLE®
From a seasoned professional comes this collection of ideas, examples, and special Applesoft subroutines to use or modify as part of your own Apple programs. By Allen L. Wyatt. 160 pages, 5½ x 8½, softbound. ISBN 0-672-22208-6. © 1983.
No. 22208 ....**$8.95**

APPLE® II FOR KIDS FROM 8 TO 80
Whatever your age, you'll think you're at Computer Camp as these enjoyable and easy to follow, beginner-level BASIC programming instructions help you quickly begin writing your own Apple II-compatible programs. By Michael P. Zabinski and Frank Mazzola. 160 pages, 8½ x 11, softbound. ISBN 0-672-22297-3. © 1984.
No. 22297 ....**$10.95**

ENHANCING YOUR APPLE® II, Volume 1 (2nd Edition)
Lets you mix text, low-res, and high-res together anywhere on-screen; have 3-D graphics, overlapping single-line colors, and other special effects; tear apart and understand somebody else's machine-language program, and much more. By Don Lancaster. 256 pages, 8½ x 11, softbound. ISBN 0-672-21822-4. © 1984.
No. 21822 ....**$15.95**

THE APPLE® II CIRCUIT DESCRIPTION

Gives you a detailed circuit description of all revisions of the Apple II and Apple II+ motherboard, including the keyboard and power supply. By Winston D. Gayler. 176 pages plus foldouts, 8½ x 11, comb-bound. ISBN 0-672-21959-X. © 1983.

No. 21959 ....$22.95

DISKS, FILES, AND PRINTERS FOR THE APPLE® II

Provides basic-to-advanced details for using disks, files, and printers with an Apple II, plus hard-to-find advice on programming with sequential-access, random-access, and executive files. By Blackwood and Blackwood. 216 pages, 6 x 9, comb-bound. ISBN 0-672-22163-2. © 1983.

No. 22163 ....$15.95

APPLE® II APPLICATIONS

Gives you a broad spectrum of tested programming and board-level interfacing applications, including serial and parallel I/O boards, EPROM or E^2PROM boards, remote data acquisition, and more. By Marvin L. De Jong. 256 pages, 5½ x 8½, softbound. ISBN 0-672-22035-0. © 1983.

No. 22035 ....$13.95

APPLESOFT LANGUAGE (2nd Edition)

New material quickly introduces you to Applesoft syntax and programming, including advanced programming techniques, graphics, color commands, sorts, searches, and more. By Blackwood and Blackwood. 288 pages, 6 x 9, comb-bound. ISBN 0-672-22073-3. © 1983.

No. 22073 ....$13.95

MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE® II, Book 1

Twenty-eight Applesoft programs, including a telephone dialer, digital stopwatch, a spelling test, house-buying guide, gas mileage calculator, and many more. By Howard Berenbon. 160 pages, 8½ x 11, comb-bound. ISBN 0-672-21789-9. © 1980.

No. 21789 ....$13.95

MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE® II, Book 2

More fascinating BASIC programs, including three dungeons, eleven household programs, seven on money or investment, two that test your level of ESP, and more — 32 in all! By Howard Berenbon. 224 pages, 8½ x 11, comb-bound. ISBN 0-672-21864-X. © 1981.

Ask for No. 21864 ....$12.95

INTERMEDIATE LEVEL APPLE® II HANDBOOK

Hard-to-find, practical info that uses ROM-based Integer BASIC to lead you into Apple 6502 machine and assembly language programming. By David L. Heiserman. 328 pages, 6 x 9, comb-bound. ISBN 0-672-21889-5. © 1983.

No. 21889 ....$16.95

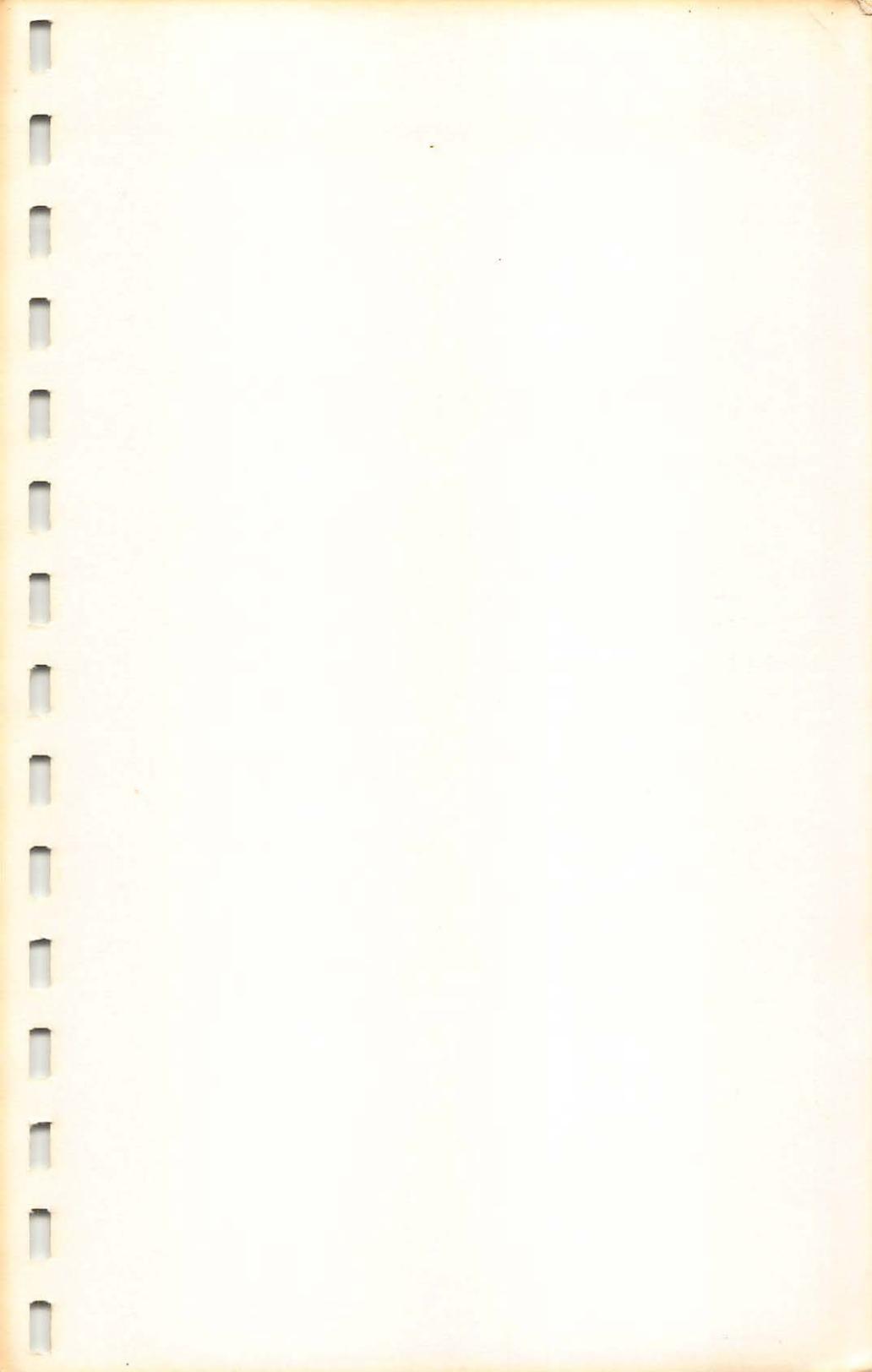## INTIMATE INSTRUCTIONS IN INTEGER BASIC

Includes much to help you build Integer programs that run smoothly and take full advantage of that dialect's rapid-running characteristics. By Blackwood and Blackwood. 160 pages, 5½ x 8½, soft. ISBN 0-672-21812-7. © 1981.

No. 21812 ....$8.95

These and other Sams Books and Software products are available from better retailers worldwide, or directly from Sams. Call 800-428-SAMS or 317-298-5566 to order, or to get the name of a Sams retailer near you. Ask for your free Sams Books and Software catalog!

Prices good in USA only. Prices and page counts subject to change without notice.

Apple is a registered trademark of Apple Computer, Inc. Macintosh is a trademark of Apple Computer, Inc.
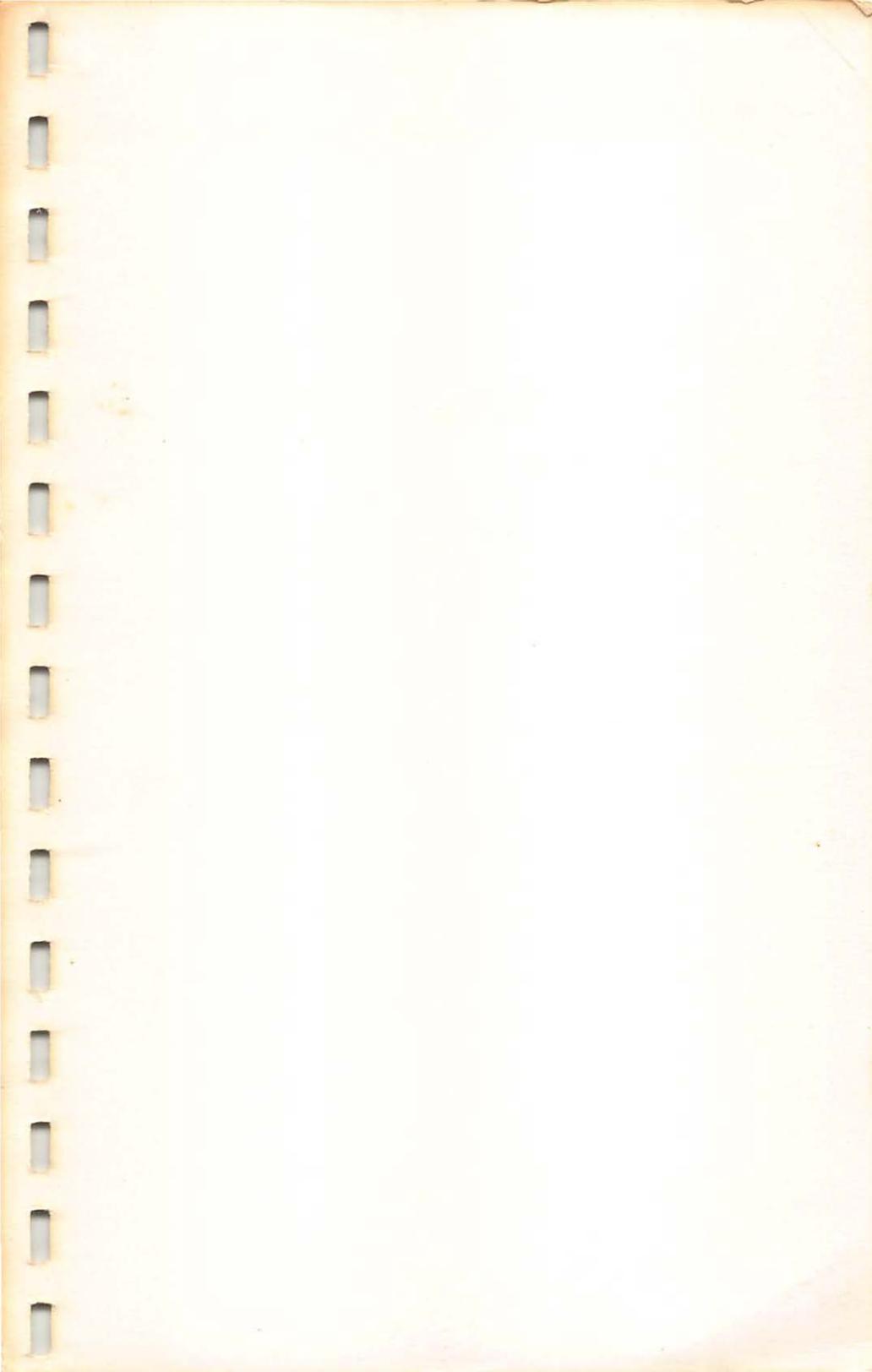
SAMS

Apple® Programmer's Handbook

The Apple computer has emerged as a true "Open System." The built-in peripheral bus, along with Apple published Monitor source listings and schematics, allows anyone to configure his own Apple into a custom computer. From word processing to video games, the Apple can be any computer you want by adding reasonably priced peripheral boards.

This book is for the people who have these customized Apples. If you need specific information on these custom features you can find just what you want to know quickly and easily in this book. Each topic is treated with specific examples. The *Apple Programmer's Handbook:*

- Tells you what peripherals you need to make your Apple II into any custom computer system
- Explains Assembler programming
- Provides maps and locations most often needed by the Assembler programmer
- Gives information on Integer and Applesoft BASIC
- Shows you hardware projects you can build
- Lists newly developed state-of-the art applications for your Apple
- Presents information so clearly written the book can be used for self study by the Apple II user

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

SAMS

Apple® Programmer's Handbook

The Apple computer has emerged as a true "Open System." The built-in peripheral bus, along with Apple published Monitor source listings and schematics, allows anyone to configure his own Apple into a custom computer. From word processing to video games, the Apple can be any computer you want by adding reasonably priced peripheral boards.

This book is for the people who have these customized Apples. If you need specific information on these custom features you can find just what you want to know quickly and easily in this book. Each topic is treated with specific examples. The *Apple Programmer's Handbook:*

- Tells you what peripherals you need to make your Apple II into any custom computer system
- Explains Assembler programming
- Provides maps and locations most often needed by the Assembler programmer
- Gives information on Integer and Applesoft BASIC
- Shows you hardware projects you can build
- Lists newly developed state-of-the art applications for your Apple
- Presents information so clearly written the book can be used for self study by the Apple II user