

WIDTH.40 and WIDTH.80 These commands select 40 and 80 column modes on IIC and IIC only. HTAB will not work past 40 columns (use FORK 1403.POS as Apple recommends). Once a mode is selected, it will be the default for the TEXT command. HGR text will always be 40 columns.

ERR.OFF Turns off ONERR GOTO

HANDLE.ERR Turns off ONERR and fixes ONERR bug. Should be used to start the module that handles errors.

INDEX

41, 7,8,10,29
 Accelerator cards, 25
 Apperand (4), 2,7,10,32
 Applesoft, 1,5,7,10,11
 Autonom, 9
 BOX, BOXFILL, 17,41
 CASB, 12,40
 CHAIM, 27,42
 Character Editor, 35,36
 COLLECT, 20,42
 COMPILER, 7,39
 Compiler, 1,25
 Control structures, 2,6,39,40
 DEFINE-FINISH, 5,39
 DEL.ARRAY, 20,42
 DISK, 19,42
 DRAW.USING, 5,17,40
 EDIT, 9,10,42
 Editor, 2,4,5,7-10
 Errors, Error handling, 20,30,31
 FILE, 13,41,20
 FLASH, 16
 GRT, 19,41
 GOTO, GOSUB, 7,9,11
 Graphics, 2,4,16,17
 Hardcopy, see PRt, see LIST, 5
 HGR, 16,40
 HOME, 6,16,40
 HSCRN, 18,41
 IN#, 29
 INKEY, 19,41
 INLINE, 19,41
 INSTR#, 24,41
 Interpreter, 1,25
 INVERSE, 16,40

Library routines, see Modules, 13
 LIST, 5,9,41
 LIST 5,10,41
 Local variables, 27,28
 LOOP-EXITWHEN-ENDLOOP, 11,39
 Memory usage, 2,32,33
 MERGE, 13,41
 Modules, 12,13,39
 Newsletter, 3
 NORMAL, 16,40
 PERFORM, 5,66,13,39
 PR#, 5,10,19
 PRINT.USING, 20,42
 RAM drives, 25
 RANDOMIZE, 20,42
 RENUM, 9,41
 REPEAT-UNTIL, 12,39
 Reset, 7,10
 RESTORE.HERR, 15,42
 REVERSE, 16,17,40
 SEARCH, 24,42
 Self-addressed-stamped-envelope, 3
 SHRINK/EXPAND/SHORTEN, 36,37
 SORT, 21-23,42
 SOUND, 18,42
 Structured programming, 2,3,11-15
 Structured Programming With BASIC, 3,35
 SWAP, 19,42
 TAB#, 16
 TAB#, 6,7,16,40
 VECTOR, 28,42
 WHEN-ELSE-ENDWHEN 5,6,11,12,40
 WHILE-ENDWHILE, 12,39
 WIDTH.40, WIDTH.80, 16,33,43
 Windows, 25-27

Blankenship BASIC

FOR THE APPLE II^e, II^e, II⁺, AND III

Copyright January 1984
 by John Blankenship
 (diskette and manual modified 1985, 1986, 1987)

P.O. BOX 47934 Atlanta GA 30362

No part of this document may be reproduced in any form. Apple and Applesoft are trademarks of Apple Computer Inc. You may freely give unaltered copies of BASIC to others but . . .

PLEASE do not distribute this documentation.

I believe LOW prices are the way to beat piracy.
 HELP PROVE ME RIGHT

CHAPTER	TITLE	PAGE
1	Introduction to BASIC	1
2	Tutorial	4
3	The Editor	8
4	Modular and Structured Programming	11
5	Text, Graphics, and Sound	16
6	Faster and Easier	19
7	Power Users	25
8	Handling Errors	30
9	Technical Specifications	32
10	Other BASIC Products	35
11	Commercial Programs	38
12	Summary of Commands	39

DISCLAIMER
 BASIC has been thoroughly tested, but programs of this size and complexity can always have errors. If you find an error you can duplicate, please send me a diskette copy of your program (include your version of BASIC) and a description of the problem and I will attempt to fix it.

I assume no responsibility for any damages, including lost profits or other incidental or consequential damages, arising from the use of, or inability to use, BASIC. My sole responsibility will be to replace the product or refund the purchase price, whichever is appropriate.

It is possible that there will be updates, additions, and utilities available for BASIC. Information about these topics, as well as helpful hints, can be found in the BASIC newsletters. This update of the manual includes the information found in the first six newsletters. Registered owners can receive free newsletters by keeping a self-addressed **STAMPED** envelope on file with me as described in Chapter 1.

CHAPTER 1 INTRODUCTION TO BBASIC

Congratulations on your purchase of BBASIC. It is one of the most powerful and easy to use languages for the Apple II line of computers. When the Apple was first introduced in the mid 70's, there was only one high level language available - Applesoft. Now, more than a decade later, the choices are so numerous that it can sometimes be confusing. As you begin your quest for the perfect language, you must first decide if you want an interpreter or a compiler.

Interpreters (like Applesoft) are very easy to use because they are interactive. You can make changes to your program and see the results almost immediately. When an error occurs, an interpreter lets you print the value of the variables to help you find the problem. If you grew-up on Applesoft, you probably take these things for granted.

A compiler is not nearly as friendly. The programming environment for a compiler typically consists of an editor, a compiler, an assembler, and a linker. You start by creating your source code with the editor and saving it to disk. When the compiler is run, it reads the source file and creates an ASM file for the assembler. Once the assembler is loaded, it can translate the ASM file into relocatable object code. Only when the linker has finished its work will you have a machine language program (that may be twice as large as the equivalent interpreted version) that is ready to run. If an error occurs anytime during the process, this lengthy sequence must be repeated from the beginning.

After reading the above discussion, you might wonder why anyone would choose a compiler over an interpreter. The answer is speed. Even though it takes much longer to develop and debug a program using a compiler, the final code often runs several times faster than an interpreter version. After their first confrontation with a compiler, though, many people decide that the time it takes a program to execute is often far less important than the time it takes to write it.

DESIGN OBJECTIVES

After buying my first Apple computer and programming with Applesoft for a while, I started looking for a programming language that offered the best features of both compilers and interpreters. I wanted to maintain the interactive simplicity of an interpreter. Since Applesoft was simple and widely used, I wanted as much compatibility as possible. I also wanted a modern structured language with NAMED procedures and enough control structures to eliminate the nasty GOTO statement entirely. Finally, I wanted a variety of new commands that would (1) make programming easier and (2) increase the execution speed to the point where an interpreter could be used for many applications that would normally require a compiler.

These objectives seemed very lofty when I first bought my Apple. By 1980, though, many of my desires were already available as Applesoft ampersand extensions, so I knew my expectations were achievable. Unfortunately, in order to satisfy even a subset of my goals, I would have had to purchase numerous packages at \$30 or \$40 each. Even if I chose to buy them, the packages would require most of the available memory, and even if there was enough room for all of them, there would almost certainly be compatibility problems between them.

I finally decided that if I wanted the perfect Apple language, I was going to have to write it myself. As I began to design the language it quickly became apparent that it would have to be more than a program. In order to achieve both power and simplicity, I needed a system capable of integrating many utilities into one compatible, easy to use,

package. Because of memory limitations, this new language would have to be coded very efficiently and its capabilities chosen very carefully. In order to further conserve memory, existing ROM routines would have to be used whenever possible.

I decided to start with Applesoft and expand it using the ampersand vector. The first step was to add modern control structures (similar to those in PASCAL) to permit programs to flow more logically. Proper structures not only make life easier for the experienced programmer, they also keep beginners from developing bad habits. Although my control structures made designing programs easier, the ampersands cluttered up the listings. To solve this problem I added an editor that automatically translated each new command into its ampersand equivalent. A new LIST command converted the commands for output and formatted the listings. Since this process was totally invisible to the user it provided a truly professional-looking way to expand Applesoft.

I was so enthused with my new system that I wanted to add dozens of additional features, but memory limitations soon forced me to pick my extensions carefully. Although BBASIC could not be all things to all people, I am very pleased with the final product. Here is a summary of the features that made it to the present version.

GRAPHICS

1. VTAB, HTAB, HOME, INVERSE PRINT, etc. work in HIRSH
2. DRAW, USING provides fast sexy IBM type shapes.
3. HSCRM determines if HIRSH points are ON or OFF.
4. BOX and BOXFILL for HIRSH

EDITOR

1. Automan and Reman
2. Insert and delete
3. List and edit procedures by name.

COMPATIBILITY

1. Existing Applesoft programs LOAD and RUN normally. KR2 is the only command not supported.
2. Variable storage is identical to Applesoft. (all local variables)
3. The 3.3 version works with DAVID DOS, Diversal DOS, etc.

CONVENIENCE

1. PRINT, USING makes formatting easy.
2. RANDOMIZE provides a random seed.
3. INDEX and INLINK provide INPUT alternatives.
4. SOUND and BELL aid your audio.
5. FREE newsletter available to registered owners.
6. And all this is in one integrated package. Quit trying to get one one firm's editor to work with another's sort or another's.....

PERFORMANCE

1. Built in commands like SORT, SMAP SEARCH, and INSTR make programs FAST and programming EASY.
2. Procedure addresses are compiled to decrease overhead.
3. CHAIN programs of unlimited size.
4. Improved garbage collection.
5. Procedures may be FILED and VERGED

STRUCTURE

1. DEFINE and PERFORM named procedures
2. REPEAT-UNTIL and WHILE-ENWHILE loops
3. Multi-line IF-THEN-ELSE-ENDIF (WHEN)
4. Listings are indented automatically
5. Procedures may be FILED and VERGED

All of the above features fit into 8K of memory. That may not impress you until you discover that your favorite editor or sort utility requires 5K by itself. BBASIC doesn't use just any 8K of memory, though. Most of it resides below the HIRSH screen in an area normally inaccessible to Applesoft (when your program is moved above the screen). This means you only give up 2K of memory to get 8K of new code. And since BBASIC uses all of Applesoft as a subroutine, you now have an 18K BASIC for your Apple.

STRUCTURED PROGRAMMING

Modular design and modern control structures are very important programming innovations. They make programs easier to design, debug, and maintain because they allow us to communicate with the computer using the same terms we use for logical thinking. Poorly structured systems force programmers to adjust their thinking to match the limited capabilities of the language.

The chapters that follow will examine each of the capabilities of

BBASIC. Since BBASIC is a highly structured language and because structured programming requires a different philosophy, there will be numerous examples to aid you in your understanding. This manual is not, however, designed to teach you how to program. It assumes that you are reasonably proficient with AppleSoft. (All of the new BBASIC commands are discussed, but none of the original AppleSoft commands are explained.) If you find you need more help (either because you are not a programmer or because you find structured programming frightfully unconventional), then you may want to purchase *Structured Programming With BBASIC* as described in Chapter 10. It is available separately so that purchasers of BBASIC have to buy only what they need. I am very dedicated to making BBASIC available at the lowest possible price. As a teacher, I am especially interested in providing a low cost solution for schools that want to use a modern language without sacrificing their investment in hardware.

ANSWERS TO YOUR QUESTIONS

This manual, like BBASIC, is very compact. Don't let the small size fool you, though. If you take the time to read the entire manual, I think you will find that it contains the answers to most of your questions. Actually, the small size is a reflection of how easy BBASIC is to use. If you have problems, you have several alternatives. To Registered owners of BBASIC are eligible for free newsletters. To receive them, you need only send in your registration form and keep a SELF-ADDRESSED STAMPED ENVELOPE on file with me. (I suggest you send several at once. Place the number of each newsletter desired in the lower left-hand corner and indicate the *last one* so you will know when to send in additional envelopes.) I use the newsletters to provide helpful hints and answer questions I feel will be of general interest. Much of this manual (this is the 2nd edition) is a result of your questions as it contains the information provided by the first six newsletters. Drop me a note when you have a question and I'll try to address it in the next newsletter. If you want an individual reply, you must include a self-addressed stamped envelope.

I realize that occasionally you might have a question that needs an immediate answer. After you have thoroughly examined the manual, feel free to call me. My home number is 404-491-3151. Because of my teaching schedule, I may or may not be at home. (BBASIC is far from profitable enough to allow it to be my full time occupation.) If I am not home, you can leave a 2 minute message on my machine. Briefly describe your problem and offer two suggestions for when I can call you back COLLECT. If I am home I will try to answer your question on the spot. If you catch me at a bad time, I may suggest a time for you to call back. I regret that I can't offer full-time support, but my low prices just don't permit it. You can help increase my services by encouraging your friends to become registered owners (see Chapter 10).

Now that you have been introduced to BBASIC, let's move along and discover why BBASIC MAKES PROGRAMMING FUN AGAIN.

MAKE SURE YOU
KEEP AN ON
ENVELOPE ON
FILE FOR THE
NEXT NEWSLETTER

Even though I suggest that you read the entire manual before you use BBASIC extensively, I know that most people will want to get started as quickly as possible. To that end, this chapter provides a summary of the information that you are most likely to need. Even if you are an experienced programmer, I encourage you not to skip this chapter. It sets the stage for things to come.

The first thing you should do is make a backup copy of your BBASIC master diskette. The exact procedure will be different depending on whether you use 3.3 DOS or PRODOS, so consult your DOS manual. Both of these versions of BBASIC look alike to the user, but there are significant differences inside the two programs.

THE BBASIC STARTUP MENU

When you boot your BBASIC master diskette, a menu will appear. The first two choices in the menu allow you to select one of the two versions of BBASIC that are on your diskette. The HIRRES Graphics version allows you to use HGR and other HIRRES graphics commands. When your application does not use any HIRRES graphics, you may want to choose selection 2. The non-graphics version is smaller (mostly because you don't need the HIRRES screen), so you will have nearly 10K of additional memory for your program.

Another option in the menu is DOCUMENTATION. It only provides a small portion of the information found in this manual and explains how BBASIC can be ordered. Actually, this documentation could be more accurately described as an advertisement. I believe in BBASIC and feel that anyone that tries BBASIC will recognize its value. Because of my confidence, I allow BBASIC to be freely given (not sold) to your friends as long as you include my advertisement. You may not, however, distribute the written documentation in any way. I decided to use this innovative method of distributing BBASIC because I believe you should get to try software before you shell out your hard-earned money. Please confirm my trust by encouraging your friends to become registered if they decide to use BBASIC. The conventional sales of BBASIC often fail to pay the advertising costs, so it is essential that I receive secondary registrations in order to stay in business (and provide future support). The first two years have been very encouraging, so I look forward to many more. I appreciate the fact that many of you have written to say that the newsletters alone are worth the registration fee.

Another menu item lets you transfer BBASIC to a previously initialized diskette. This is an easy way to let your friends try BBASIC. This option only transfers the BBASIC system itself. It does not copy any of the demonstration programs. The 3.3 DOS version of BBASIC has an additional selection that will initialize a new diskette for you (and transfer BBASIC). If you have PRODOS BBASIC you may RUN the program PRODOS.DOCUMENT. It lists a few minor places where PRODOS BBASIC differs from the 3.3 version.

JUMP RIGHT IN

Boot your BBASIC disk and choose Option 1 from the menu to get into the graphics version. When the flashing cursor appears, type in the following line but do not press RETURN.

THIS IS A TEST

Use the LEFT and RIGHT arrow keys to move the cursor along the line. If you type a character, it will be INSERTED at the cursor position with the rest of the line shifting to the right. If you wish to delete the character to the left of the cursor you can use CTL P (hold down

the control key and press P) or CTL Z. If you have a *file*, *lrc*, or *gs*, you may use the **DELETE** KEY. When you get the line the way you want it, press **RETURN** to send it to **BBASIC**. You do not have to have the cursor at the end of the line. The entire line will be sent, no matter where the cursor is. Of course, if you press **RETURN** now, **BBASIC** will respond with a **SYNTAX ERROR** since our example line is not a valid **BBASIC** statement. If you wish to abort the line, press **ESC**. There are many more editing features discussed in Chapter 3, but this will be enough to get you started.

Type **CATALOG** (or **CAT** for **Prodos**) and **RETURN** to see a list of programs on your diskette. If you press **CTL F** for **FILES**, the **BBASIC** editor will type **CATALOG** for you. When the catalog appears, notice that one of the programs is named **NESTING EXAMPLE**. The name will be **NESTING.EXAMPLE** on the **PRODOS** disk since **PRODOS** does not allow spaces in a program name. Load the program by typing in the following line.

LOAD NESTING EXAMPLE

Take a look at the program by typing **LIST**. If you press **CTL L**, **BBASIC** will type **LIST** for you. You may start and stop the listing from scrolling by using almost any key. I recommend the space bar. When you have the listing in a pause state, you can abort by pressing either **ESC** or **RETURN**.

HARDCOPY LISTINGS

If you would like a printout of this program (and you have a normal printer interface in **SLOT 1**), type **LIST** and **RETURN**. You should not issue a **PR#1**. When you use **LIST**, **BBASIC** will automatically turn on the printer. **LIST** the program, and turn the printer off.

If you examine the listing you will see that most of the commands look like normal **Applesoft** commands. In fact, you may use every **Applesoft** command except for **RGR2**. A few of these old commands have been enhanced and act a little differently. The new **LIST** command, for example, automatically formats the program. **loops** and **WHEN** (multiline **IF-THEN-ELSE**) statements are indented to show their actions more clearly. **Defined modules** (subroutines) are indented and separated by a blank line. These modules begin with **DEFINE**, end with **FINISH**, and are called with **PERFORM**.

You can also use the **list** command to begin listing the program at a specific module. Type in the following line.

LIST "HORIZ.BAT"

When you press **RETURN** the program will list starting with line 1500, which is the first line of the module "HORIZ.BAT". The list will not stop until the end of the program. Use the space bar and **ESC** to control the listing as you see fit.

RUN THE PROGRAM

You run **BBASIC** programs just like you do with **Applesoft**. Type **RUN** now to run the program previously loaded into memory. You should see a short message telling you that the program will draw 4 types of electronic components on your screen. It will also ask you how many total components you want. Answer 25 and press **RETURN**. You will see 25 components drawn on the **HIBES** screen. At the bottom of the screen will be the question **DO YOU WANT TO SEE IT AGAIN (Y/N)?** Answer **N** to end the program.

This program shows how easily **BBASIC** handles graphics. Let's try a few immediate mode commands to demonstrate this point. Type **HOME** to clear the screen and position the cursor at the top of the page. Now type the following line.

AS="RRJJJJJJRR"

Although it will appear that you are in the text mode, remember that the program ended with your **Apple** in the graphics mode and we have not issued a **TEXT** command. To prove your **Apple** is still in the graphics mode type the following command.

HPLOT 0,0 TO 50,100

A line will appear on your screen. Now type this statement.

DRAW USING AS

DRAW USING AS that draws a shape specified by a string, in this case **AS**. Refer to our definition of **AS** above. The **R** indicates movement to the right and **I,J,K**, and **M** are used for the

GIVES 5

diagonals. These movements define **AS** to be the symbol for a resistor. (Refer to Chapter 5 for complete details on the **DRAW USING** command.)

When the resistor appears on the screen it starts at the last point plotted, which is the end of the line. The resistor will be very small. Type in the following line.

DRAW USING "2"AS"4"AS"8"AS

Three more resistors will appear. Each starts at the last point plotted and will be twice as large as the previous resistor. The "numbers" in the string control the size of the shape. After a "2" for example, each **R** will move 2 dots to the right instead of 1. Type in the following line.

DRAW USING "DDD LLL D JMKI"

This will draw a line down, left, down a little, and then a diamond. All movements will be of size "8" because the default is the last size used. The size may be from "1" to "9" and spaces are ignored so you may use them to improve the readability of string.

You may want to experiment with the **DRAW USING** command before you continue. Type **HOME** to clear the screen. (Notice that the **BBASIC HOME** command works for both the **TEXT** and **HIBES** screens.) You may use **HPLOT** to determine where the shapes will appear. If you don't plot any points after a **HOME**, the shapes will start in the center of the screen. Don't feel intimidated if you feel that you need more information about the **DRAW USING** command. It has many additional features and they will all be explored later. Remember, the purpose of this chapter is just to get you started.

CONTROL STRUCTURES

Now that you know a little about the **DRAW USING** statement, let's look at the program again and see how it works. Type **LIST**. **BBASIC** will automatically enter the text mode before the listing begins. This is necessary because the text on the graphics screen cannot scroll. Look at the listing (or at the hardcopy you made earlier).

The first command in the program is **COMPILE**. It is always required in any **BBASIC** program that uses **PERFORM** statements. **PERFORM** is very similar to the **Applesoft GOSUB** statement except that you can **PERFORM** named procedures instead of **GOSUB**ing to line numbers. Names make your programs much easier to understand. The readability of the program is also improved because **BBASIC** provides modern control structures. Control structures are used to determine the order that program statements execute. You should already be familiar with the **Applesoft** control structures **FOR-NEXT** and **IF-GOTO**. One advantage of having a full complement of control structures is that the **GOTO** can be eliminated completely. Two new structures are used in this program.

The first of these new structures is a **REPEAT-UNTIL** loop which starts in line 1010 and ends at line 1220. Notice that all the lines in between are indented to make the body of the loop more obvious. Everything in the loop will be repeated until **AS="N"**.

Another new structure is the **WHEN-THEN-ELSE-ENDWHEN**. Look at the example below.

```
100 INPUT X
110 WHEN X>100 THEN
120 PRINT "X is very large"
130 PRINT "In fact it is over 100"
140 ELSE
150 PRINT "X is less than 100"
160 ENDWHEN
```

If **X>100** then lines 120 and 130 will be executed and everything between the **ELSE** and the **ENDWHEN** will be skipped. If **X<=100** then only the lines between the **ELSE** and the **ENDWHEN** will be performed. The indenting makes it easier to follow the logic of this decision-making structure. **BBASIC** will indent the lines properly no matter how you enter them. The only time indenting will not be handled properly is when a control structure is left out. (For example, if you had an **UNTIL** without a **REPEAT**.) In fact, you can often discover where you have forgotten a structure with only a quick look at the listing. If you forget an originating structure (like **FOR**), then the listing will

be moved to the left toward the line numbers. The listing will drift to the right when a terminating structure (like NEXT) is omitted.

BASIC-AND THE AMPERSAND

As mentioned in Chapter 1, BASIC uses the Applesoft ampersand (&) vector to add the new commands. Press RESET (CTL RESET on some Apples) and then LIST our example program. You will notice that all the new BASIC commands have been prepended. COMPILER, for example, has become & STOR. REPEAT is & CONT. BASIC follows the & with standard Applesoft key words so they can be tokenized to save memory space.

I only mention this because I don't want you to have a heart attack if you press RESET and try to list your program. Normally, you will never have to deal with anything but the BASIC commands themselves. When you type in a BASIC command the BASIC editor will automatically convert it to its appropriate & version. When you LIST a program, BASIC checks for &'s and converts each instruction back to the more readable form.

Although Applesoft ignores spaces when it examines a line, the BASIC editor is not so forgiving. BASIC commands will not be recognized (or translated) if they contain imbedded spaces. If you type LIST, for example, the editor will simply pass it on to Applesoft which will do a normal list.

When you press RESET the BASIC editor is disconnected and Applesoft's editor takes control. Don't worry, though, as there are many ways to put the BASIC editor back in charge. Often, all you have to do is RUN your program. Since the & commands do not require the editor to be functioning in order to execute (only to be entered), they will operate properly. Several commands (such as TEXT) reconnect the editor.

Your first thought might be that TEXT is an Applesoft command, not a BASIC command. Actually, TEXT was an Applesoft command, but if you press RESET and use Applesoft to LIST line 1250 you will see that TEXT is actually a new BASIC command. The new command was necessary because BASIC needs to move between the 80 column, 40 column and HGR modes in a much more complicated manner than does Applesoft. In later chapters you will see that there are several Applesoft commands that will act differently if they are entered under control of the BASIC editor.

&I Another way of returning to BASIC after a RESET, is to type &I and press RETURN. This command will reconnect the BASIC editor and leave your program intact. You can tell which editor is in effect by looking at the cursor. BASIC's editor uses a flashing cursor, even on the graphics screen.

APPLESOFT PROGRAMS

I wish to emphasize that you may LOAD Applesoft programs into BASIC and they will RUN properly (as long as they do not use HGR2). The only difference you will see is that the BASIC editor will be in effect for INPUT statements and you can eliminate that by pressing RESET before running your program. You must be very careful, though, if you use the BASIC editor to alter lines in an Applesoft program. Any BASIC command (such as HGR) will be converted to its ampersand equivalent and the program will operate differently. (You will have mixed text and graphics instead of the split Applesoft HIRTS screen with four lines of text at the bottom.)

BASIC supports GOTO and GOSUB statements only to permit you to run Applesoft programs. These statements should never be used when writing new BASIC programs.

This chapter has provided only a brief encounter with BASIC. The purpose was to introduce you to the new environment as quickly as possible. Now that you know how to operate BASIC you should have no trouble digesting the rest of this manual.

CHAPTER 3 THE EDITOR

Many first-time users of BASIC may find the editor a little unusual. If you give it a chance, though, I think you will discover that the things that seem the most peculiar at first will soon become some of your favorite features. The reason for this is that BASIC is very different from Applesoft. A structured environment requires a new way of thinking about programming. As you adapt to this new way of thinking you will find that the BASIC editor has been specifically designed to simplify modular programming.

The BASIC editor is always in effect. You may use it when entering a new line or editing an old one. All of the features are available even during normal INPUT statements. Here is a summary of the primary editing commands.

CURSOR MOVEMENTS

- CTL B - moves cursor to the beginning of the line
- CTL E - moves cursor to the end of the line
- CTL R - moves cursor left one character
- left arrow - moves cursor right one character
- right arrow - moves cursor left five characters
- up arrow - moves cursor left five characters
- CTL K - same as up arrow for II+ users
- down arrow - moves cursor right five characters
- CTL J - same as down arrow for II+ users

DELETING CHARACTERS

- DELETE - erases character to the left of the cursor
- CTL P - same as DELETE (the II+ does not have DELETE)
- CTL Z - same as DELETE (preferred by left handers)
- CTL W - deletes the word to the left of the cursor

WHEN YOU ARE THROUGH

- RETURN - sends the entire line to BASIC (also used to stop a LIST)
- CTL Q - sends only the portion of the line that is left of the cursor (quit at cursor)
- ESC - aborts the present line without making any changes (also used to stop a LIST, stop editing, and to abort a long CATALOG)
- CTL X - same as ESC (because Applesoft used it)

ENTERING CONTROL CHARACTERS

- CTL V - Enters the next character typed, even if it is a control character (verbatim). Control characters are displayed in inverse in the text mode and as special characters in the graphics mode. (Note: If you enter the monitor with a CALL-151, you will have to use CTL V to enter a CTL C to return to BASIC.)

The following characters have special meaning if they are typed as the first character on the line.

- SPACE - prints a RETURN so that a space will continue a long CATALOG just like it always has with Applesoft
- ESC - Aborts a long CATALOG or a program LIST. If used during INPUT it ends the program. (ESC aborts most everything)
- CTL I - types a new line number equal to the last line number plus 1
- CTL C - passes directly through editor to stop program
- CTL R - types EDIT for you
- CTL L - types LIST for you
- CTL F - types CATALOG for you (FILERS)

INITIALIZATION

- &I - reconnects the BASIC editor after a RESET

* NUMBERING STATEMENTS

EDITING vs LISTING
One peculiar characteristic about the editor is that it will not list one line for you. If you type LIST 100, for example, the program will begin listing at line 100 and continue to the end of the program. As it turns out, you never need to LIST one line in BASIC. Think about this for a moment. The only reason that you ever listed a single line with Applesoft was to BSC up to it and modify some portion of it. If you wish to modify line 100 with BASIC just type EDIT 100 and the line will be presented for you to modify.

I think the easiest way to learn anything on a computer is to try it, so type in the following lines.

```
NEW
100 COMPILR
101 FOR I=1 TO 10
102 PERFORM "HELLO"
103 NEXT I
```

If you are like me, you hate typing in line numbers. BASIC will type the next line number for you if you press CTL I as the first character on the line (or TAB on a IIc, IIc, or GS). Try it as you type in the following lines.

```
104 END
105 DEFINE "HELLO"
106 PRINT "HI THERE"
107 FINISH
```

Type LIST and notice that BASIC has formatted the program to make it easier to read. As you look at the listing, you are probably wondering why I increment the line numbers by 1. The answer won't be obvious or perhaps even believable until you use BASIC for a while and accept the fact that line numbers have very little importance. They are only used for editing. A properly coded BASIC program will never use GOSUB or GOTO so line numbers will never appear in the body of the program. This means that a program can be renumbered by changing only the line numbers themselves. Type RENUM and notice how quickly you get the cursor back. Even very large programs will renumber almost instantaneously. List the program again. It should look like this.

```
1000 COMPILR
1010 FOR I = 1 TO 10
1020 PERFORM "HELLO"
1030 NEXT I
1040 END
```

```
1050 DEFINE "HELLO"
1060 PRINT "HI THERE"
1070 FINISH
```

RENUM always starts with the number 1000 to make sure the left side of the lines will line up correctly. If we started with line number 10, the listing would be out of line at 100 and again at 1000. RENUM also always uses an increment of 10. For structured programs, that usually will be plenty of space for inserting lines (especially with the auto-increment of 1). If you find you need more space just RENUM again.

Your first thoughts might be that you will never be able to keep track of your line numbers if you are constantly renumbering. The nice thing is that with BASIC you don't have to keep track of your line numbers. You will see in later chapters that properly designed programs will be made up of many small DEFINED modules. Let's assume that we want to edit something in the module "HELLO". With Applesoft you would need to know the line numbers so that you could list it on the screen. With BASIC just type EDIT "HELLO". Do so now and you will see the first line appear for you to edit.

If the line needs editing you may use any of the commands listed earlier. When you are through (or if no changes are necessary), you may press RETURN (or CTL Q) to enter the line. The next line will automatically appear for editing. This will continue until you press BSC (or CTL X). You might find this strange at first, but I can almost

guarantee that you will grow to love it in a very short time. The real advantage is that you no longer have to use line numbers when you edit your programs. (Note: EDIT will also work with a line number. You will usually use this option when an ERROR is reported in a specific line.)

LIST operates in a similar manner. You may LIST 1050 or LIST "HELLO". Both of these commands will start listing at line 1050 and continue until the end of the program. Use the space bar to start and stop the LIST, and BSC or RETURN to abort the list (while it is stopped).

RESTRICTIONS

The editor is responsible for converting new BASIC commands into their Appersand (A) equivalent. In order for this conversion to work properly, there can be only one BASIC command per line. If you enter two BASIC commands on the same line then only one of them will be converted. (You can still use the colon to separate multiple Applesoft commands.) If you must use a BASIC command and an Applesoft command on the same line, the BASIC command must be the first statement on the line or it will not LIST and indent properly. This simply means that you should generally use only single statement lines in BASIC programs. Actually, it is a good idea anyway because it makes the program easier to read. A special utility is available (see Chapter 10) that can compress your program if you need more space.

Another restriction is that the BASIC editor will only let you enter 79 characters per line. Actually, this is not much of a restriction if you use only one statement per line. The only situation where you might require more characters is a long PRINT statement. When you find it absolutely necessary to use more than 79 characters per line, you can trick BASIC into letting you enter them. To do so, type in your line but do not leave a space between the line number and the first character. When you hear the warning beep, indicating that the line is full, press RETURN. Type HOME to clear the screen and then EDIT your line. You will then be able to enter the same number of characters as you can with Applesoft. The only problem you will have with this long line is that it cannot be edited at the bottom of the screen (which is why BASIC tries to keep you from creating long lines). Doing so will cause continuous scrolling. (If this occurs, just press ESC, type HOME, and EDIT the line at the top of the screen.) The reason for the scrolling is that BASIC does not edit on the screen. All editing is done in the buffer and the line is printed over and over in the same spot on the screen. This makes it possible for one simple subroutine to handle editing for the 40 column, 80 column, and HRES graphic modes.

Since the editor continually prints to the screen, you must never issue a PR#1 to turn on the printer while in the BASIC immediate mode. Use LIST when you wish to get a hardcopy and always turn the printer on and off inside your program with the DISK command (see chapter 6).

Sometimes you might need to turn on the printer while in the immediate mode (to print a catalog for example). When you do, just press RESET to return you to the Applesoft editor before typing PR#1. When you are finished, use KT to reconnect the BASIC editor.

Occasionally, the BASIC editor might conflict with DOS if a FILE name contains a BASIC reserved word (such as SORT). If this happens you can RSR# and then perform your command. For example:

```
LOCK SORT.TEST
You can then use KI to reconnect the editor. Another way of avoiding this problem (short of not using reserved words in file names), is to use the DISK command in the immediate mode as follows.
DISK "LOCK SORT.TEST"
```

CHAPTER 4 MODULAR AND STRUCTURED PROGRAMMING

If you have never used a structured language before, you may have a little trouble adjusting to programming without using GOTO. Structured programming is not just a way to write programs; it is a way of thinking. Old habits die hard, so don't get discouraged. Once your mind makes the "flip" (and it will), you'll wonder how you ever organized a program using unstructured techniques.

MAKING DECISIONS

Let's start by recognizing why GOTO commands are used in Applesoft. The first reason is to create an IF-THEN-ELSE decision. The WHEN control structure in BASIC allows this to be done without a GOTO as shown below.

<pre> Applesoft version 10 IF NOT(X>Y) THEN 40 20 PRINT "X IS LARGER" 30 GOTO 50 40 PRINT "Y IS LARGER" 50 REM REST OF PROGRAM </pre>	<pre> BASIC version 10 WHEN X>Y THEN 20 PRINT "X IS LARGER" 30 ELSE 40 PRINT "Y IS LARGER" 50 ENDWHEN </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------

If the decision following the WHEN is true, then all the lines between the WHEN will be executed and all the lines between the ELSE and the ENDWHEN will be skipped. If the decision is false, then only the lines between the ELSE and the ENDWHEN will be performed. Not only can you place as many lines between the WHEN and the ELSE (or ELSE and ENDWHEN) as you need, you can nest WHEN statements inside of each other. BASIC knows which ELSE and which ENDWHEN goes with each WHEN and will indent the listing so that it is easy for you to read. Every WHEN must have an ENDWHEN, but the ELSE is optional. The example programs below show two ways of making the same decision. See if you can find a simpler way.

<pre> 100 INPUT X,Y 110 WHEN X>100 AND Y>100 THEN 120 PRINT "BOTH ARE BIG" 130 ELSE 140 WHEN X>100 THEN 150 PRINT "ONLY X IS BIG" 160 ENDWHEN 170 WHEN Y>100 THEN 180 PRINT "ONLY Y IS BIG" 190 ENDWHEN 200 ENDWHEN </pre>	<pre> 100 INPUT X,Y 110 WHEN X>100 AND Y>100 THEN 120 PRINT "BOTH ARE BIG" 130 ELSE 140 WHEN X>100 OR Y>100 THEN 150 PRINT "ONLY X IS BIG" 160 PRINT "ONLY Y IS BIG" 170 ELSE 180 PRINT "ONLY Y IS BIG" 190 ENDWHEN 200 ENDWHEN 210 ENDWHEN </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

LOOPS

The only other reason for using a GOTO in a program is to create a LOOP. If you think about loops, you will see that BASIC has a loop structure for every application. The standard FOR loop is perfect if you know in advance how many times the loop is to be performed. If you want to loop WHILE something is true or UNTIL something happens then you have several options. The WHILE loop decides at the beginning of the loop and the REPEAT loop decides at the end. Pascal only has WHILE and REPEAT loops, so it maintains a GOTO in order to exit from the middle of a loop. BASIC can eliminate the GOTO entirely because it has the LOOP-EXITWHEN-ENDLOOP construct to let you exit anywhere inside the loop. You may even use several EXITWHEN statements if you need them. The examples below show how BASIC's loop structures can eliminate the GOTO from similar situations in Applesoft.

<pre> Applesoft version 10 REM beginning of program 20 body of 30 loop 40 INPUT "AGAIN?":A\$ 50 IF A\$="YES" THEN 10 </pre>	<pre> BASIC version 10 REPEAT 20 body of 30 loop 40 INPUT "AGAIN?":A\$ 50 UNTIL A\$="NO" </pre>
---------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------

<pre> 10 IF NOT(X>Y) THEN 50 20 body of 30 loop 40 GOTO 10 50 REM rest of program </pre>	<pre> 10 WHILE X<Y 20 body of 30 loop 40 ENDWHILE 50 REM rest of program </pre>
-------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------

PERFORM, DEFINE, FINISH, COMPILER
 Instead of GOSUBing to a line number, BASIC allows you to PERFORM a named procedure. Each procedure must begin with a DEFINE "NAME" statement and end with FINISH. All characters in the name are significant, but longer names do slow down the program slightly. If the name does not contain any reserved words it does not have to be in quotes, but I recommend them. Use a PERFORM "NAME" to execute the desired procedure (subroutine). The first statement in any BASIC program that uses PERFORM should be COMPILER. Applesoft penalizes you for using subroutines because Applesoft searches the entire program for the specified line number. COMPILER causes BASIC to create a table of subroutines and their addresses. This makes PERFORM faster than GOSUB because a small table can be searched instead of an entire program.

You generally should not use IF to decide whether to PERFORM a module or not. The following line, for example, will operate correctly, but it will not list properly.

```

Enter      2000 IF K=2 THEN PERFORM "SOME MODULE"
Lists as   2000 IF K=2 THEN & CALL "SOME MODULE"
                
```

A better way to handle this situation is with WHEN as shown below.

```

2000 WHEN K=2 THEN
2010 PERFORM "SOME MODULE"
2020 ENDWHEN
                
```

Another option is to use the CASE statement. The following line is functionally identical to the examples above.

```

2000 CASE K=2: "SOME MODULE"
                
```

CASE

CASE is actually much more powerful than the above example would indicate. It is similar to the Applesoft ON K GOSUB statement. In the following example, module A will be performed if K=1, module B will be performed if K=2, etc. If K is less than one or more than the number of modules, then no action is taken.

```

2000 CASE K: "A", "B", "C", "D"
                
```

MODULAR PROGRAMMING

Modern control structures are only part of a well designed program. Structures are used to control the flow through a program by deciding which parts are executed and how many times they are repeated. These "parts" of a program need to be organized into modules. Each module

should have one, and only one, well defined function. Poorly structured programs are hard to design and debug because their logic is distributed throughout the program. Modular design lets you concentrate your efforts on one problem at a time.

When you approach a programming problem, break it down into smaller and simpler problems (modules). If these new modules are still complicated, just break them down into even smaller, easier-to-solve modules. Let's look at an example. Suppose we wanted to create a program that would move a ROBOT across the room to a door on the other side. The main program could look like this.

```

1000 REPEAT
1010   PERFORM "LOCATE DOOR"
1020   PERFORM "PAGE DOOR"
1030   PERFORM "MOVE FORWARD"
1040   UNTIL FLAG$="AT DOOR"
1050 END
  
```

The PERFORM statement is very much like AppleSoft's GOSUB except that the subroutine can be called by its name instead of a line number. Notice how the use of modules makes the logic of this program easier to understand. Naturally, we must clearly define the function of each of these modules and write the code for them. If a module's function is complex, then we can break it down into smaller modules. Let's look at the "MOVE FORWARD" module for example.

```

1060 DEFINE "MOVE FORWARD"
1070   PERFORM "CHECK FOR OBSTACLE"
1080   WHEN OK=1 THEN
1090     PERFORM "MOVE FORWARD 6 INCHES"
1100   ELSE
1110     PERFORM "GO AROUND OBJECT"
1120   ENDWHEN
1130 FINISH
  
```

Naturally, we now have to create modules to solve these newly introduced tasks. The new modules may also be complicated so we just keep creating new solutions until the new tasks become simple enough to code without defining new modules.

Since you only have to think about one module at a time, you can concentrate your efforts without being distracted by related problems. After you become accustomed to thinking about programming in this manner, you will find that you will not only more productive but that programming will be more enjoyable.

BUILDING A LIBRARY

As you learn to structure your programs, you will find that many of your modules will be general purpose. Since it is undesirable to continually re-invent the wheel, BASIC provides an easy way for you to maintain a procedure library. For example, you could save the "MOVE FORWARD" module defined earlier with the command:

```

FILE "MOVE FORWARD"
FILE IS SIMILAR TO SAVE EXCEPT THAT IT ONLY SAVES THE MODULE SPECIFIED. YOU MAY ADD A FILTER MODULE TO A PROGRAM IN MEMORY WITH:
MERGE "MOVE FORWARD"
  
```

Merged modules will always be added to the end of the existing program no matter what line numbers they have. After merging, you should RENUM or you may not be able to EDIT some of the lines.

If you create a library of useful routines, you may wish to send them to me (on disk please). When I get enough routines to fill a diskette I will make them available at a modest price. If I use any of your routines you will get a free copy of the final disk. If you wish to contribute please follow these guidelines. Use REM statements in your code to define the function as clearly as possible. Specify what variables are used to pass data to and from your module. If a small demo would be useful then please add one. If your module is called "SCREEN BUILDER" then name the demo something like SCREEN BUILDER DEMO. To help get you started let me offer a module for entering data. This procedure could use many enhancements, but it should give you some ideas.

```

1000 COMPIL
1010 HOME
1020 SY = 3
1030 READ N
1040 PRINT "PLEASE ENTER THE FOLLOWING"
1050 FOR I = 1 TO N
1060   READ PROMPT$,SL$,SH$,SS
1070   SX = LEN (PROMPT$) + 2
1080   VTAB SY
1090   PRINT PROMPT$:
1100   PERFORM "INPUT"
1110   SY = SY + 1
1120   A$(I) = SY$
1130 NEXT
1140 PRINT : PRINT
1150 PRINT "THE DATA IS:"
1160 FOR I = 1 TO N
1170   PRINT A$(I)
1180 NEXT
1190 DATA 6
1200 DATA "YOUR NAME (ALL CAPS)---" "A", "Z", ".15
1210 DATA "YOUR ADDRESS---" "0", "Z", ".20
1220 DATA "CITY---" "A", "Z", ".10
1230 DATA "STATE---" "A", "Z", ".2
1240 DATA "ZIP CODE---" "0", "9", ".5
1250 END
  
```

```

1260 DEFINE "INPUT"
1270 REM This procedure will present an inverse field for input.
1280 REM The user may specify the size of the field (SS), where the field
1290 REM is to be located (SX,SY) and the upper and lower limits (SH$,SL$).
1300 REM The input string (including leading and trailing spaces) will be in
1310 REM the variable SS$
1320 REM Other variables used are SC$ (the char. being input), ST (a temp.
1330 REM variable), SN (the number of characters entered so far), and SF (a
1340 REM flag that indicates the RETURN key was pressed).
1350 SS$ = "":SF = 0:SN = 0
1360 INVERSE
1370 FOR ST = 1 TO SS
1380   SS$ = SS$ + " "
1390 NEXT
1400 REPEAT
1410   VTAB SY: HTAB SX
1420   PRINT SS$:
1430   SC$ = " "
1440   REPEAT
1450     INKEY SC$
1460     UNTIL SC$ < > " "
1470     WHEN ASC (SC$) = 13 THEN
1480       SF = 1
1490     ELSE
1500       WHEN ASC (SC$) = 127 OR ASC (SC$) = 8 THEN
1510         SN = SN - 1
1520         IF SN < 0 THEN SN = 0
1530         WHEN SN = 0 THEN
1540           SS$ = " "
1550           FOR ST = 1 TO SS
1560             SS$ = SS$ + " "
1570           NEXT
1580           ELSE
1590             WHEN SN + 1 = SS THEN
1600               SS$ = LEFT$ (SS$,SN) + " "
1610             ELSE
1620               SS$ = LEFT$ (SS$,SN) + " " + RIGHT$ (SS$,SS - SN - 1)
1630             ENDWHEN
1640             ELSE
1650               WHEN (SC$ > = SL$ AND SC$ < = SH$ AND SN < SS) OR SC$ = " " THEN
1660                 WHEN SN > 0 THEN
1670                   SS$ = LEFT$ (SS$,SN) + SC$
1680                 ELSE
1690                   SS$ = LEFT$ (SS$,SN) + SC$
1700                 ELSE
1710                   SS$ = LEFT$ (SS$,SN) + SC$ + RIGHT$ (SS$,SS - SN - 1)
1720                 ENDWHEN
1730             ELSE
  
```


You can use a dummy loop to control the delay (and thus the speed of the flashing) between prints. This HIRRS flashing will also occur if you issue REVERSE in the immediate mode because the BASIC editor prints the line over and over as described above. If this happens just type NORMAL.

BOX, BOXFILL
 BASIC has commands for drawing boxes on the screen. BOX draws an open box and BOXFILL creates a solid box. Both commands use the last color specified. The syntax looks like this.

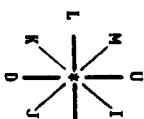
```
BOX X,Y,X1,Y1
BOXFILL X,Y,X1,Y1
```

X and Y are the coordinates of the top left hand corner of the box. X1 and Y1 are the coordinates of the lower right hand corner. Naturally, you can use any variable or formula for each of the arguments.

DRAW.USING

The DRAW.USING command provides a very easy way to draw simple shapes. (You may still use AppleSoft shape tables if you need to rotate your shapes, but I think you'll like the ease, convenience, and speed of BASIC's shapes.) As discussed in Chapter 2, DRAW.USING allows you to describe a shape with a string. The characters you may use in the string are listed below.

- Movement
- U - move up
 - D - move down
 - L - move left
 - R - move right
 - I - move diagonally up and right
 - J - move diagonally down and right
 - K - move diagonally down and left
 - M - move diagonally up and left



On/Off N - turns plotting on (leave trail during moves)
 DRAW.USING always starts in the N mode

P - turns plotting off (move without plotting)

Size l-9 indicates how many dots to move for each letter
 Note: l-9 must be characters (strings not numbers).

Spaces may be used as desired to improve readability.
 Use HCOLOR before drawing your shape to set the color to be used.

Let's try an example. Suppose you wanted to draw a box with a diamond inside it. Look at the example program below.

```
1000 HGR
1010 HQR
1020 A$ = "RRRRDDDDLLLLUUUU"
1030 B$ = "FJD N IJKN"
1040 C$ = "FJJR"
1050 SYMBOL$ = A$+B$+C$
1060 HPLOT 1,1
1070 FOR J = 1 TO 9
1080 DRAW.USING STR$(J) + SYMBOL$
1090 NEXT
1100 VTAB 20
```

*Printer's JUNK
 WHEN GIVEN:
 RUN*

The first two lines select the graphics mode and clear the screen. The string A\$ draws the box. B\$ turns the plotting off and moves to the point where the diamond is to be positioned. The plotting is turned back on and the diamond is drawn. C\$ turns the plotting off again and moves to the lower right hand corner of the box so that the next time a shape is drawn it will begin there. Line 1050 combines the pieces into one easy-to-use "shape" variable. The HPLOT determines where the first shape will begin. The FOR loop draws nine shapes. The first shape will be of size 1, the second of size 2, etc. Notice that the variable J must be converted to a string before it can be used to control the size of the shape. The final line in the program moves the cursor to line 20 so that it will not interfere with the drawing when the program ends.

BELL and SOUND
 You may use the BASIC command BELL to keep the speaker. BELL is not really any easier than printing a CHR\$(7), but it makes a more readable program.

If you want a more pleasant tone than BELL provides or if you need interesting sound effects, you can use the SOUND command. SOUND has the following syntax.

```
SOUND D,F,G
```

D is the duration and can have value of 0-127. The frequency of the tone is controlled by the second argument which may range from 1 to 191. Changing the frequency will have no effect on the duration. For normal tones, the last argument (effect) should be 0. You can use E (1-255) to create unusual sounds. The best sounds seem to have effects near the extremes (1 or 255). Experiment with SOUND and you will see that you can get lasers, machine guns, etc. The following program will demonstrate some of the possible sounds.

```
1000 FOR J = 1 TO 6
1010 READ E
1020 DATA 0,1,3,10,253,255
1030 FOR F = 1 TO 191 STEP 30
1040 PRINT: PRINT "F=";F;" E=";E
1050 SOUND 60,F,E
1060 FOR I = 1 TO 200
1070 NEXT I
1080 NEXT F
1090 NEXT J
```

*Random
 Frequency*

HSCRN
 The HSCRN command allows BASIC programs to determine if a given dot position on the HIRRS screen is off or on. The syntax is:

```
HSCRN X,Y,Z
```

The coordinates of the dot are X,Y. After the command is executed the HIRRS variable Z will be 1 if the dot is on and 0 if it is off.

HSCRN

CHAPTER 8 FASTER AND EASIER

In addition to modern control structures and mixed text and graphics, BASIC has many new commands that either make programming easier, or your programs run faster, or both.

SWAP

AppleSoft normally requires you to use three statements and a temporary variable if you want to swap the value of two variables. The SWAP command is not only easier for you, but it is considerably faster. If you need to exchange the value of X and Y use SWAP X,Y. Naturally, you can swap integers or strings just as easily.

DISK

One of the most unattractive things about handling DOS commands in AppleSoft is the required use of CHR\$(4). BASIC solves this problem by replacing PRINT CHR\$(4) with DISK. For example, you can get a CATALOG from inside your program with:

```
1000 DISK "CATALOG"
```

DISK also allows you to turn the printer on and off inside your program. This is necessary because (as stated in chapter 2) you should never use PR#1 as an immediate command. The example below shows how to let the user decide if the output should go to the screen or to the printer.

```
2000 INPUT "Screen or Printer (S/P) ";A$
:
:
3000 WHEN A$="P"
3010 DISK "PR#1"
3020 ENDWHEN
```

print statements in program

```
4000 DISK "PR#0"
```

INKEY, GET

AppleSoft programmers are familiar with the GET statement. BASIC's INKEY is very similar to GET except that INKEY does not stop and wait for an input like GET does. If no key has been pressed, then INKEY does nothing. If a key has been pressed, then INKEY acts exactly like GET. As with GET, INKEY should only be used with a string variable. The following example will print periods on your screen until the K key is pressed.

```
2000 A$=""
2010 R$PRAT
2020 PRINT ".":
2030 INKEY A$
2040 UNTIL A$="K"
```

The BASIC GET is slightly different from AppleSoft's GET. It looks the same to the user except that it no longer works with disk files. Many people have used GET to read a text file that contains quotes and commas. With BASIC you can accomplish the same thing much faster with INLINE. See the example below.

INLINE

The AppleSoft INPUT command does not let you enter such characters as quotes (") or commas (,). INLINE works like INPUT except that it

allows any character (except RETURN) to be entered. INLINE should be used only with a string variable. The example below shows how to print a sequential text file that may contain quotes or commas.

```
1000 INPUT "WHAT FILE NAME ";F$
1010 ONERR GOTO 60000
1020 DISK "OPEN ";F$
1030 DISK "READ ";F$
1040 LOOP
1050 INLINE A$
1060 PRINT A$
1070 ENDL00P
1080 END
```

FILES

```
60000 HANDLE ERR
60010 DISK "CLOSE"
60020 PRINT "ALL DATA READ"
60030 END
```

Note: Refer to Chapter 8 for more information about the use of ONERR.

PRINT.USING

BASIC's PRINT.USING command makes the formatting of numbers very easy. A string or string variable is used as a mask which specifies how the output should look, as shown below.

```
PRINT.USING "$$$$.00";2.5 or X = 2.5
MASK$ = "$$$$.##"
PRINT.USING MASK$;X
```

The example masks and outputs below show the various capabilities. Assume the number being printed is 23.058.

MASK and NUMBER	OUTPUT	COMMENT
"###.##";123.45	123.45	use # as place holders
"###.##";23.057	23.06	notice rounding
"###.##";.057	.6	leading 0's are suppressed
"###.00";.057	.06	keep 0's if you want
"###.00";.057	0.06	keep 0's if you want
"###.00";.12	\$ 12.00	\$ at beginning of field
"###\$.00";12	\$12.00	\$ floats to front of number
"ANSWER=#.00";6:2	ANSWER= 6:2	strings may be used
"###.##";12345.67	####	shows number won't fit

The # symbol is used to indicate where the number should go. When the program is run though, the # signs will be replaced with spaces.

DEL-ARRAY

If you need to erase (not just set the values to zero) an array from memory, you can use the command DEL.ARRAY. This is helpful if you no longer need the array or if you wish to redimension it.

RANDOMIZE

The AppleSoft RND function can be used to generate random numbers. It produces each number by applying a formula to the last number. The initial number in this sequence is called a seed. Unfortunately, when the Apple is turned on, it always starts with the same seed. The BASIC command RANDOMIZE solves this problem by using the time taken to press the last key to generate a random seed. Generally, you should only execute the RANDOMIZE command once in each program.

COLLECT

AppleSoft (and thus BASIC) dynamically allocates the memory used to store strings. This is good because it means that the minimum amount of memory is always used. This process does, however, have its drawbacks. In particular, dynamic allocation requires some memory to be set aside for temporary use. These temporary strings, which are often called garbage, continue to expand into any available memory. When the memory is full (or if your program needs the memory for its variables), then the string space must be reorganized and the temporary strings thrown away. This reorganization is affectionately called garbage collection. AppleSoft uses a very inefficient algorithm for

garbage collection, so the process can take many minutes under some circumstances. The BASIC command COLLECT will collect garbage much quicker than AppleSoft.

PRODOS offers its own fast collection process. (Refer to your DOS manual.) Rather than duplicate code, PRODOS BASIC uses the PRODOS collection process when you use COLLECT. Most of the new code required for PRODOS BASIC fits in the area originally set aside for COLLECT. This allows both DOS versions of BASIC to occupy the same amount of memory and to have identical starting points for the major subroutines.

SORT
The BASIC SORT command is both fast and convenient. You may sort any one or two dimensional array (integer, string, or real) by using SORT X where X is the name of the name of the array.

All sorts are in ascending order, but it is easy for the programmer to simulate descending order. If you have an array A\$ with elements 0-N, for example, you could sort it and print it out in descending order as follows.

```
2000 SORT A$
2010 FOR I = N TO 0 STEP -1
2020 PRINT A$(I)
2030 NEXT I
```

Notice that SORT uses the 0 element, not just 1-N. It is also important to know that BASIC sorts the entire array. If you have dimensioned an array to 100 and use only 25 of the elements then the remainder of the array (which will contain 0's or null strings) will be sorted right along with the actual data. Generally, this will mean that your data will end up at the end of the array. This may seem like a large problem, but it is actually very easy to solve. One of the easiest solutions is to fill the array with something that will appear very large to SORT. The largest number AppleSoft can handle is approximately 1.78E+38 so that works well for numeric arrays. With strings, initialize each element to CHR\$(255). If you are having trouble with the SORT command, study the example below.

```
1000 COMPIL
1010 SIZE=100: DIM A(SIZE)
1020 REM now fill the array with large number
1030 FOR I = 0 TO SIZE
1040 A(I) = 1.7 E+38
1050 NEXT
1060 PERFORM "INPUT ARRAY"
1070 HOME
1080 PRINT "BEFORE SORT":PRINT
1090 PERFORM "PRINT ARRAY"
1100 SORT A
1110 PRINT:PRINT "AFTER SORT": PRINT
1120 PERFORM "PRINT ARRAY"
1130 END

1140 DEFINE "INPUT ARRAY"
1150 TEXT
1160 PRINT "ENTER NUMBERS TO BE SORTED (RETURN WHEN DONE)"
1170 PRINT
1180 N = 0: REM number of names entered so far
1190 LOOP
1200 INPUT X$
1210 EXITWHEN X$=""
1220 A(N)=VAL(X$)
1230 N=N+1
1240 ENDOLOOP
1250 N=N-1
1260 REM the numbers are in A(0) through A(N)
1270 FINISH

1280 DEFINE "PRINT ARRAY"
1290 FOR I=0 TO N
1300 PRINT A(I)
1310 NEXT
1320 REM note: NEXT is faster without a variable
1330 FINISH
```

Generally, you will find the BASIC SORT 30 to 60 times faster than comparable AppleSoft sorts. Even so, you can obtain even greater speed by using BASIC's ability to sort two dimensional arrays. When a two dimensional array such as A(X,Y) is used, X is the column number and Y is the row number. The array is sorted by the first column. Each time elements are moved, everything in the entire row is moved. The elements in the row could be name, address, city, state, zip, etc.

The program below shows how to use the two dimensional SORT.

```
1000 COMPIL
1010 TEXT
1020 DIM A$(3,2)
1030 PERFORM "READ DATA"
1040 PERFORM "PRINT DATA"
1050 PERFORM "SORT DATA"
1060 PERFORM "PRINT DATA"
1070 END

1080 DEFINE "READ DATA"
1090 RESTORE,HERE
1100 FOR I = 0 TO 2
1110 READ A$(0,I),A$(1,I),A$(2,I),A$(3,I)
1120 NEXT
1130 DATA JOHN,37,160,BROWN
1140 DATA SUSAN,24,116,BLOND
1150 DATA TOM,28,175,BLACK
1160 FINISH

1170 DEFINE "PRINT DATA"
1180 PRINT "NAME AGE WEIGHT HAIR COLOR"
1190 PRINT "-----"
1200 FOR I = 0 TO 2
1210 PRINT A$(0,I):
1220 HTAB 8: PRINT A$(2,I):
1230 HTAB 14: PRINT A$(1,I):
1240 HTAB 23: PRINT A$(3,I)
1250 NEXT
1260 FINISH

1270 DEFINE "SORT DATA"
1280 PRINT
1290 PRINT "SORT USING WHICH COLUMN?"
1300 PRINT "1. NAME"
1310 PRINT "2. AGE"
1320 PRINT "3. WEIGHT"
1330 PRINT "4. HAIR COLOR"
1340 PRINT
1350 INPUT "ENTER NUMBER OF YOUR CHOICE ":C
1360 C = C-1
1370 PRINT
1380 FOR I = 0 TO 2
1390 SWAP A$(0,I),A$(C,I)
1400 NEXT
1410 SORT A$
1420 FOR I = 0 TO 2
1430 SWAP A$(0,I), A$(C,I)
1440 NEXT
1450 FINISH
```

Sometimes (especially with disk files) it is better never actually to move the data. Instead, a system of pointers can be used to keep track of the desired order. The simple program below shows how pointers can be used to manage six students and the grades they made on four tests.

```

1000 COMPILER "READ.DATA"
1010 PERFORM "READ.DATA"
1020 TEXT
1030 PRINT "BELOW ARE THE UNSORTED NAMES AND GRADES"
1040 PRINT
1050 PERFORM "PRINT.DATA"
1060 SORT NAMES$
1070 PRINT
1080 PRINT "USING POINTERS MEANS YOU NEVER MOVE THE DATA"
1090 PRINT
1100 PERFORM "PRINT.DATA"
1110 END

1120 DEFINE "READ.DATA"
1130 RESTORE HERE
1140 READ NS
1150 DIM NAMES$(1,NS-1)
1160 FOR SN = 0 TO NS-1
1170 READ NAMES$(0,SN)
1180 NAMES$(1,SN) = STR$(SN)
1190 NEXT
1200 READ NT
1210 DIM G(NS-1,NT-1)
1220 FOR TN = 0 TO NT-1
1230 FOR SN = 0 TO NS-1
1240 READ G(SN,TN)
1250 NEXT
1260 NEXT
1270 DATA 6
1280 DATA SMITH,JONES,BLANKENSHIP,MILLS,BLACK,WILLIAMS
1290 DATA 4
1300 DATA 66,75,85,82,91,81
1310 DATA 72,73,89,80,96,78
1320 DATA 53,79,81,90,100,92
1330 DATA 78,71,86,79,94,83
1340 FINISH

1350 DEFINE "PRINT.DATA"
1360 PRINT "NAME "
1370 FOR I = 1 TO NT
1380 PRINT " TT":I;
1390 NEXT
1400 PRINT: PRINT
1410 FOR SN = 0 TO NS-1
1420 WHEN LEN(NAMES$(0,SN)) >= 10 THEN
1430 PRINT LEFT$(NAMES$(0,SN),10);
1440 ELSE
1450 PRINT NAMES$(0,SN);
1460 PRINT SPC(10-LEN(NAMES$(0,SN)));
1470 RNDWHEN
1480 FOR TN = 0 TO NT-1
1490 PT = VAL (NAMES$(1,SN))
1500 PRINT USING "###";G(PT,TN)
1510 NEXT
1520 PRINT
1530 NEXT
1540 FINISH

1550 DEFINE "VARIABLE DOCUMENTATION"
1560 RRM NS - number of students
1570 RRM NT - number of tests
1580 RRM SM - student number
1590 RRM TN - test number
1600 RRM PT - pointer
1610 RRM G - array for grades
1620 RRM NAMES$ - array for names and pointers
1630 FINISH

```

SEARCH

When you need to search an array with AppleSoft you have to use a loop, which can be very time consuming. BASIC provides an extremely fast SEARCH command for searching arrays. SEARCH M,X,B,C means to search for the Mth occurrence of X in the array B (integer, string, or real). If no match is found, then the real variable C will be set to -1. Otherwise, C is equal to the position in the array that X was found. The module below shows how to list all occurrences of a user specified string in the array A\$.

```

2000 DEFINE "PRINT.ALL"
2010 INPUT "SEARCH FOR WHAT ";X$
2020 N = 1
2030 REPEAT
2040 SEARCH M,X$,A$,P
2050 IF P>-1 THEN PRINT "FOUND AT ";P
2060 N = N+1
2070 UNTIL P=-1
2080 FINISH

```

When searching a string array, the second argument in the SEARCH command must be a string variable (and not a formula or a literal string). This restriction does not apply to numeric arrays.

INSTR\$

The INSTR\$ command lets you search one string for the occurrence of another. INSTR M,X\$,B,C searches for the Mth occurrence of X\$ in B\$ and sets the real variable C equal to the character position found. C will equal 0 if not found. The program below lists the individual words in a user provided sentence by looking for the spaces.

```

1000 SPACES$ = " "
1010 INPUT "ENTER YOUR SENTENCE ";X$
1020 P1=1
1030 N=1
1040 REPEAT
1050 INSTR$ M,SPACES$,X$,P2
1060 WHEN P2>0 THEN
1070 PRINT MID$(X$,P1,P2-P1)
1080 P1 = P2+1
1090 N = N+1
1100 ENDWHEN
1110 UNTIL P2=0
1120 PRINT RIGHT$(X$,LEN(X$)-P1+1)

```

CHAPTER 7 POWER USERS

A number of BASIC owners have written to ask for a BASIC compiler. I thought about it for a while, but (at least for now) I have decided against it. It's not that a compiler wouldn't be nice, but the speed gained doesn't appear to be worth the effort required. This is especially true with BASIC (as indicated in chapter 6).

First of all, I prefer the interactive environment of an interpreter (as discussed in Chapter 1) over a compiler. Second, and perhaps even more important, you don't necessarily need a compiler to create programs with professional speed. This is true because BASIC can perform many of the most needed functions many times faster than compiled code. BASIC's SORT, SEARCH, INSTR, and DRAW.Using are prime examples. These commands perform their operations 30 to 100 times faster than equivalent AppleSoft code. Compiled AppleSoft only gives you speed increases of 2 to 10 times, with the average closer to 2 than 10. This often means that interpreted BASIC programs run much faster than programs written with a compiler. Further, the compiled programs generally will take longer to write and debug and often will require considerably more memory.

IMPROVING YOUR HARDWARE

If you really want maximum performance from your system, I have several suggestions for you. First, unless you have a GS, you should purchase an accelerator card. I have an accelerator for the Titan Technology and I am very satisfied with it, although there are several new ones on the market with additional features. I have never found any program that wouldn't run with it, and they run three times as fast. Many AppleSoft compilers don't offer that much speed increase. And a compiler will only speed up your BASIC programs. An accelerator card will speed up wordprocessing, graphics, spreadsheets, everything! Study the accelerators on the market and buy one of them. You'll get a 300% increase in power for only a 15% increase in your investment.

The accelerator won't increase your disk speed, though, because DOS commands are automatically slowed down to the normal clock speed (required by DOS). If you are using 3.3 DOS you can speed up disk access by as much as 500% by upgrading to ProDOS or by using one of the fast 3.3 DOS's such as DAVID DOS or Diversal-DOS. If you have a hard disk or a 3.5 inch disk, then ProDOS is a logical choice because of its subdirectory environment. Otherwise, I prefer DAVID or Diversal because they not only give you the same speed as ProDOS, but they are capable of moving themselves into the language card, freeing up an additional 10K of memory for BASIC. An accelerator and a fast DOS will make you think you have a new machine: but why stop there.

If you really want to see your Apple fly, add a RAM disk to the above configurations. ProDOS users have one built in. Otherwise, you will have to add special hardware, software, or both. You will find the speed to be astonishing. An accelerator and a fast DOS can BLOAD HIRRS screens from the RAM disk so fast you can almost perform animation.

WINDOWS

After you have built your SUPER Apple (accelerator, fast DOS, and RAM disk), you will be able to do things with BASIC that you might never have considered before. For example, you may have seen windows on the MAC and wanted to use them in your programs. You may have thought about spending a couple of weeks writing a machine language module to handle the task. You might have even seen such routines in your favorite magazine, but they probably only handled windows on the text screen. The reason for this is that HIRRS windows not only require high speed to move BK of memory, they also need a place to put it.

Since each window takes BK, you may run out of memory very quickly if you open several windows at once. When you realize that your program will also have to manage output from PRINT statements to the WINDOWS, etc., you may put the idea aside, assuming it would be too much trouble.

With a SUPER Apple, you don't have to resort to machine language. I was able to develop a simple BASIC windowing system in less than an hour. The number of windows open at any time is limited only by your (RAM) disk space. When each window is closed, the previous screen is restored exactly as it was. Even the cursor is returned to its original position. The following program demonstrates my windows. FILE these routines on your library disk and MERGE them into your programs whenever you need windows.

```

1000 COMPILR
1010 HGR
1020 HOME
1030 WNUM= 0
1040 PRINT "Windows Demo": PRINT
1050 INPUT "HOW MANY WINDOWS TO OPEN ":N
1060 FOR I = 1 TO N
1070 INPUT "L,W,R,WB ":L,W,R,WB
1080 PERFORM "OPEN WINDOW"
1090 PRINT "THIS IS WINDOW NUMBER ":I
1100 INPUT "PRESS RETURN TO CONTINUE":A$
1110 NEXT
1120 FOR I = 1 TO N
1130 INPUT "PRESS RETURN":A$
1140 PERFORM "CLOSE WINDOW"
1150 NEXT
1160 END

1170 DEFINE "WINDOW ROUTINES"
1180 DEFINE "OPEN WINDOW"
1190 POKR 17362,165: POKR 17363,34
1200 WNUM= WNUM + 1
1210 REM save vital information
1220 FOR W = 0 TO 5
1230 WSIZE(W,WNUM)= PEEK (32 + W)
1240 NEXT
1250 NEXT "BSAVE WINDOW#":WNUM: ",A$2000,1$2000"
1260 REM create new window
1270 HCOLOR= 0
1280 BOXFILL WL * 7,WT * 8,WR * 7,WB * 8
1290 HCOLOR= 3
1300 BOX WL * 7,WT * 8,WR * 7,WB * 8
1310 BOX WL * 7 + 3,WT * 8 + 3,WR * 7 - 3,WB * 8 - 3
1320 WL= WL + 1:WR = WR - 1
1330 WT= WT + 1:WB = WB - 1
1340 POKR 32,WL: POKR 33,WR
1350 POKR 34,WT: POKR 35,WB
1360 REM get into window
1370 VTAB WT + 1: HTAB WL + 1
1380 FINISH

1390 DEFINE "CLOSE WINDOW"
1400 REM restore everything as it was
1410 FOR W = 0 TO 5
1420 POKR 32 + W,WSIZE(W,WNUM)
1430 NEXT
1440 DISK "BLOAD WINDOW#":WNUM
1450 DISK "DELETE WINDOW#":WNUM
1460 WNUM= WNUM - 1
1470 FINISH
1480 FINISH

```

HOW MANY WINDOWS

THIS IS WINDOW NUMBER 1
PRESS RETURN TO CONTINUE

THIS IS WINDOW NUMBER 2
PRESS RETURN TO CONTINUE
ML,MR,MT,MB 23,36,11,23

THIS IS MAIN
WINDOW NUMBER 3
PRESS RETURN TO CONTINUE
ML,MR,MT,MB 3,25,6,14

AppleSoft uses locations 32-35 (decimal) to hold the size of the window (for scrolling text and wrapping PRINT statements). BASIC not only supports text windows just like AppleSoft, but it also handles HIRRS windows with two differences. TEXT windows use location 33 to hold the window width. BASIC HIRRS windows use location 33 to hold the TAB of the right side of the window. The second difference is that text in a HIRRS window cannot scroll.

I also discovered a small error in the internal BASIC window routines while working on this program. Line 1190 corrects the problem and should be included in any program that uses HIRRS windows.

If you run this program on a normal Apple it will be very slow. On a SUPER Apple, though, windows open and close almost instantaneously. Any one of the three recommended improvements will help a little. Together, the speed will make your heart flutter. WARNING: If you ever try a SUPER Apple, you will never be satisfied with anything less.

CHAIN

One of the biggest reasons for having a RAM disk is the BASIC CHAIN command. The syntax is CHAIN "PROGRAM.NAME" where PROGRAM.NAME is the name of a program that your program wants to run. (Note: CHAIN requires Apple's CHAIN program to be on your disk. It has been licensed from Apple and comes on your BASIC disk. You can also find it on your DOS master.) The new program is loaded and executed just as if you had run it with a DISK "RUN PROGRAM.NAME" with one exception: CHAIN maintains all the variables so they can be used by the second program. If you break down your program into appropriate segments, your program can be as large as your available disk space. And with the SUPER Apple's RAM disk, each segment will load with very little decrease in performance. The RAM disk can also be used to hold arrays larger than memory (using random access files). The file variables will be slower than a normal array, but a SUPER Apple makes their use very acceptable.

LOCAL VARIABLES

Occasionally someone writes to ask why BASIC does not support local variables. There are really two reasons. First, local variables make interpreters very slow and memory-hungry. Second, and perhaps more important, I wanted to maintain total compatibility with AppleSoft's format for storing variables. Doing so lets you use most third party AppleSoft enhancements and utilities with BASIC. Even with this explanation, some people still want to use local variables. Since I don't have any immediate plans for adding local variables to BASIC, I thought I would show you how they can be simulated. The basic premise is to create a stack (the array SS) for saving and passing variables. The following demo program shows a simple method for implementing this idea. If two stacks were used (one for passing and one for saving variables), the implementation might be a little easier, but this should get you started (assuming you're interested in exploring local variables).

This program uses the fact that $N! = N!(N-1)!$ to allow factorials to be calculated with a subroutine that calls itself (something that normally cannot be done unless the language supports local variables). This example is only meant to be an educational exercise, but with a little work, you might develop a useful utility.

```
1000 COMPILR "INITIALIZATION"
1010 PERFORM "INITIALIZATION"
1020 HOME
1030 PRINT "N","N!"
1040 PRINT "-----"
1050 FOR I = 1 TO 10
1060 SS = SS + 1:SS(SS) = 1: REM PUSH NUMBER ON STACK
1070 PERFORM "FACTORIAL"
1080 NF = SS(SS):SS = SS - 1: REM PULL ANSWER FROM STACK
1090 PRINT I,NF
1100 NEXT
1110 DEFINE "FACTORIAL"
1120 REM THIS PROCEDURE WILL FIND THE FACTORIAL OF THE ITEM ON THE STACK AND
1130 REM PLACE THE ANSWER BACK ON THE STACK IN ITS PLACE
1140 REM ALL VARIABLES USED WILL BE SAVED AND RESTORED MAKING IT RE-ENTRANT
1150 SS = SS + 1:SS(SS) = N: REM SAVE VARIABLE N
1160 SS = SS + 1:SS(SS) = NF: REM SAVE VARIABLE NF
1170 N = SS(SS - 2): REM GET NUMBER OFF STACK
1180 WHEN N = 1 THEN
1190 SS(SS - 2) = 1: REM PUSH ANSWER ON STACK
1200 ELSE
1210 N = N - 1
1220 SS = SS + 1
1230 SS(SS) = N
1240 PERFORM "FACTORIAL"
1250 NF = SS(SS):SS = SS - 1: REM GET ANSWER OFF STACK
1260 NF = NF * (N + 1): REM CALCULATE NEW ANSWER
1270 SS(SS - 2) = NF: REM AND PLACE IT ON THE STACK
1280 ENDWHEN
1290 REM NOW RESTORE VARIABLES USED TO THEIR ORIGINAL VALUE
1300 NF = SS(SS)
1310 SS = SS - 1
1320 N = SS(SS)
1330 SS = SS - 1
1340 FINISH
1350 DEFINE "INITIALIZATION"
1360 DIM SS(100)
1370 SS = 1
1380 REM SS() IS THE STACK
1390 REM SS IS THE STACK POINTER
1400 REM FOR PROGRAMS USING STRINGS, USE S$(S)
1410 FINISH
```

VECTOR

I have tried to include in BASIC most of the features you normally will need. However, I realize that sometimes you will have an application that requires something I have left out. Often that means that you will have to write your own ampersand extensions. Since BASIC already uses the ampersand vector, I wanted a simple way for you to interface new extensions. The BASIC command VECTOR helps solve this problem. For example, let's assume you write a routine that starts at \$300 (768 decimal). Let's also assume that you normally call your routine with the command "A.A.B.C" and that it determines which is bigger (A or B) and puts the answer in the variable C. In order to use your routine with BASIC, you must do three things. First, your program must load your routine. Second, you must use VECTOR to tell BASIC where to go when it finds a non-BASIC ampersand command. Finally, you need to use a double ampersand when calling your routine. The double ampersand ensures that AppleSoft's "get character" pointer will point to the second ampersand and your program can follow the

normal] rules for Applesoft ampersand extensions. The lines below show how these three requirements can be met.

```
2000 DISK "BLOAD NAME, A$300"  
2010 VECTOR 768  
2020 AA A,B,C
```

Some extensions will use BBASIC reserved words and may cause you trouble. If you cannot get by without using reserved words, try to solve the problem by PEEKing BBASIC's vector and saving it. PEEK in your vector, execute your command, and restore BBASIC's vector. In some cases, you might also need to turn off the BBASIC editor with a DISK "IN#0". You can turn it back on with an KI inside your program. This technique *should* allow any ampersand routine to work with BBASIC as long as there are no memory conflicts.

CHAPTER 8 HANDLING ERRORS

Applesoft has the ONERR GOTO statement to help your program deal with errors. Rather than duplicate the error-handling code, BBASIC makes as much use of ONERR as possible. The first problem I had to fix was that when an error occurs, ONERR transfers execution to a specified line number. Since BBASIC's RENUM command does not alter any line numbers in the body of the program, the use of RENUM would make ONERR very difficult. Consequently, RENUM will not renumber any line numbers above 59904 (I use 60000). You should place each of your error-handling routines at 60000 or greater.

APPLESOFT ONERR BUG

As indicated in Apple's documentation, there is a bug in the Applesoft ROM's that prevents proper operation of the ONERR statement. BBASIC fixes the problem with the command HANDLE.ERR. You should start each of your error handling routines with this command. HANDLE.ERR also turns off the ONERR flag. This means that any errors that occur inside your routine will generate normal error messages. If you suspect errors in your routine you could use another ONERR statement, though I would generally discourage such complexity especially since it is not necessary. Let's examine some proper methods for handling errors.

You should not expect to handle all possible errors with one routine. In fact, usually your program can test for most errors by using more conventional means. For example, suppose your program asks the user to enter two numbers that are going to be used in a division problem. Instead of using ONERR to catch a division by zero error, your program could simply prevent the user from entering a zero to begin with. The following example shows how.

```
200 REPEAT  
210 INPUT "ENTER TWO NUMBERS ":A,B  
220 IF B=0 THEN PRINT "The 2nd number cannot be 0"  
230 UNTIL B<>0  
240 C = A/B
```

Even though the above example shows the best way for preventing division by zero errors, let's see how ONERR can be used to accomplish the same thing. Let me emphasize that I would not use this method, but this simple example does provide an effective way to demonstrate ONERR.

```
200 INPUT "ENTER TWO NUMBERS ":A,B  
210 ONERR GOTO 60000  
220 C = A/B  
230 ERR.OFF
```

```
60000 HANDLE.ERR  
60010 PRINT "The 2nd number cannot be 0"  
66020 INPUT "ENTER YOUR NUMBER AGAIN ":B  
66030 RESUME
```

ERR.OFF

There are several items in this example that need discussion. First, notice the new command ERR.OFF. It cancels the last ONERR command and should always be used immediately after the line that might cause the error. Proper bracketing of your program with ONERR-ERR.OFF statements can simplify error-handling because you can design different handlers for each section of your program.

RESUME returns control to the line that caused the error (line 220 in this case). Sometimes it is desirable to transfer control to some line other than the offending one. Although Applesoft does not support such a transfer, you can do so with BASIC. Make sure the potential error will occur in a loop. Instead of using RESUME in your error handler, use the loop terminator to restart the loop. (The loop being restarted must be active; that is, it must be the inner loop if several loops are nested.) The example below demonstrates this principle.

```

100 PRINT "ENTER NUMBERS TO BE DIVIDED"
110 PRINT "ENTER TWO ZEROS WHEN DONE"
120 ONERR GOTO 60000
130 LOOP
140 INPUT "ENTER TWO NUMBERS :A,B
150 EXITWHEN A=0 AND B=0
160 PRINT "THE ANSWER IS :A/B
170 ENDOLOOP
180 END

60000 HANDLR.ERR
60010 PRINT "THE SECOND NUMBER CANNOT BE 0"
60020 ENDOLOOP

```

The ENDOLOOP in the error-handling routine will not indent properly because of the absence of a corresponding LOOP, but it will execute properly by returning control to line 130.

The proper use of error-handling routines can make your programs more user friendly. Use them carefully, though. Improperly thought out routines can lock up your program in an endless loop.

ERROR MESSAGES

In addition to the error messages of Applesoft, there are 3 new messages for BBASIC. They are as follows.

Unexpected Terminator (error # 17) means a command such as ENDWHILE or ENDWHEN was found without an appropriate beginning command (such as WHILE or WHEN).

Terminator Missing (error # 18) means a terminating command was expected but not found.

Undefined Procedure (error # 19) means the COMPILER command was missing or the name in the PBRFORM command does not match exactly a defined module.

(ProDOS BBASIC uses error codes 22,23, and 24.)

CHAPTER 9 TECHNICAL SPECIFICATIONS

Much of the material in this chapter will be of little interest to the average BBASIC programmer, so don't be alarmed if you find it less than useful or even confusing. You don't need any of this information to use BBASIC. For those of you who desire to go where no man has gone before, I hope you will find it helpful.

AMPERSAND USAGE

The BBASIC command is really an invisible ampersand (&) command. The following is a list of the commands actually used for each deferred BASIC statement.

REPEAT	&CONT	HOMR	LASC
UNTIL	&TO	INVERSE	&NORMAL
WRITE	&FOR	REVERSE	&RECALL
ENDWHILE	&RSUMS	NORMAL	&CLEAR
ENDWHEN	&STOP	BELL	&ABS
ELSE	&OR	SOUND	&PERK
EXITWHEN	&NOTRACE	COLLECT	&RETURN
PERFORM	&CALL	INLINE	&TRACE
DEFINR	&DEF	SORT	&AND
FINISH	&END	SWAP	&AND
COMPILER	&STORE	SWAP	&LATN
CASE	&ON	SEARCH	&ONERR
VECTOR	&USR	INSTR\$	&GOSUB
PRINT.USING	&PRINT	BOXFILL	&XDRAW
HSCRN	&SCR	BOX	&GR
WIDTH.40	&4	RANDOMIZE	&RESTORE
WIDTH.80	&8	CHAIN	&HGR2
GET	&G	ENDLOOP	&RIGHT\$
INKEY	&AT	LOOP	&COS
DISK	&GRT	WHEN	&DRL
DRAW.USING	&POP	DEL.ARRAY	&DL
TEXT	&DRAW	HANDLE.ERR	&WAIT
HGR	&SGN	ERR.OFF	&STEP
	&FN	RESTORE.HERR	&SPERD=

MEMORY USAGE

The memory map for BBASIC is nearly identical to that of normal Applesoft except that the starting point of the application program has been moved up in memory. BBASIC (2.7) resides from \$800 to \$4C45 (\$2520 for max.mem). I chose to put BBASIC below the program area for two reasons. First, the easiest (and most often used) place for you to put ampersand extensions is below DOS at HIMEM, so I did not want to use that space. Second, normal Applesoft programs using HIRRS graphics have only 6K of workspace unless the application program is moved above the HIRRS screen. When this is done, the 6K below the screen is wasted because it cannot be used for program or variable storage. Most of BBASIC fits into this 6K space. Only 2K of BBASIC takes up memory that could normally be used (the area immediately following the HIRRS screen). This means you effectively get 8K of new code but you only have to give up 2K to get it.

You may use page 3 just as you did with Applesoft. (I do use page 3 temporarily for some of the optimizing utilities.) BBASIC does use some additional zero page locations. They are 0.1,2,3,4,5,6,7,8,19,1A,1B,5B,5C,5D,FA,FB,FC,FD,FE, and FF. I should also point out that BBASIC uses all of the Applesoft ROM routines, so you really have an 18K BASIC. After summing up all of the zero page locations used by

AppleSoft, the monitor, DOS, and BBASIC, there are very few left over for your use. Generally, unless you specifically know you will not cause a conflict, you should only use locations 9, 1B, 1C, 8B, CF, D6, D7, 83 8F, EF, and F9.

The new locations chosen for BBASIC are not used by most programs. The only known exception at this time is an early printer interface card that used locations 0 and 1. The LLIST command fails with this card.

CUSTOM MODIFICATIONS

Occasionally, I hear from someone who dislikes my editor. Since editors are one of the most personal aspects of computing, I want to help you customize BBASIC to satisfy your requirements. At location \$046 (version 2.7), there is a JSR instruction to \$1910 which is the address of the subroutine that accepts a line for BBASIC. You might want to alter my editor or just change the JSR to some other routine entirely. For example, if you use \$FD1B, then BBASIC will use the old Apple RSC-JRM editor. If you know how to determine the entry point of your favorite editor (like GPE, for example), you should be able to use it with BBASIC. The only problems I expect you to have are with MERGE and FILE. The first eight instructions in my editor check to see if a FILE or MERGE is in effect and a jump is made to adjust the appropriate pointers. I suspect that you may want to place a JMP to your editor at \$1924 instead of modifying the JSR, but a lot may depend on the editor you are trying to install. If you get something working, send me your patches and I will pass them on in a newsletter.

If you customize BBASIC for your use, PLEASE DO NOT GIVE IT TO ANYONE. It will become impossible for me to help people if there are modified versions floating around. I'm more than willing to provide technical information to those who need it, but you must help me by not distributing your modifications.

For those of you who really want to get into BBASIC, let me provide you with a starting point. The BBASIC (2.7) dispatch table starts at \$FB8. Each entry in the table contains three bytes. The first byte is the token for the reserved word used by BBASIC as described earlier. The next two bytes contain the address (less one) of the routine that handles that command. The first three bytes in the table are 83 71 11. 83 is the token for STOP which is used by BBASIC for ENDWHEN. The address of the routine that handles ENDWHEN is \$1171+1 or \$1172. Remember, if you don't understand any of this, just ignore it and be happy you aren't a die-hard computer freak. Luckily, BBASIC has nearly every feature you will ever need, so customizing is usually unnecessary.

80 COLUMNS FOR THE II+

A few people with VIDEX or other 80 column boards have asked about special support. Since I do not have any II+ 80 boards, I have not tried any of the following suggestions. I did however design the video interface for BBASIC in such a way that it would be easy to handle both 40 and 80 columns on the I/O as well as text on the HRBS screen. The secret is that I do not edit the screen memory (which is different in all three cases). Instead, I edit in the page 2 buffer and continually reprint the buffer to the screen. This means (at least theoretically) that any 80 column card should be easily interfaced with BBASIC.

I'll try to explain the requirements and then any of you familiar with your 80 column card's requirements can work on a patch. If you are successful, please pass on the information and I'll put it in a newsletter.

BBASIC already assumes that you turn on your card with a PR#3, which should work for all cards. Turning the card off is another matter, though. Many cards will not fully disconnect with a PR#0. Instead, they require a special control code to be sent to them with a print statement. This is accomplished in BBASIC at address \$48C7 which holds a LDA immediate instruction to pick up the code to turn off the card (which is \$15 for the I/O).

The only other problem I anticipate is that different cards use

different locations for their horizontal and vertical tabs. Most cards (I have been told) use location \$26 for the vertical tab just like the 40 column screen. This number is acquired by the LDA instruction at \$1959.

Horizontal tabs are often different from \$24, which is used by the 40 column screen. The I/O uses \$57B to hold its horizontal tab in the 80 column mode. The code from \$1939 through \$1949 determines if BBASIC is using 40 or 80 columns and loads the X register with either location \$24 or location \$57B, whichever is appropriate. Later, at \$1873, there are four instructions that reset the locations to the proper horizontal position.

I believe that these are the only places that might cause conflict with other 80 column cards. All addresses above are for the GRAPHICS version of BBASIC (both 3.3 and ProDOS 2.7). If you get your card working, let me know and I will provide the addresses for the max memory version.

PATCHING PRODOS BBASIO

Always BLOAD a fresh copy of ProDOS BBASIC before you modify it and BSAVE it before you run it. This is necessary because ProDOS BBASIC has a few bytes of self-modifying code, and saving the code after execution will cause your computer to hang on the next run.

CHAPTER 10
OTHER BASIC PRODUCTS

BASIC comes in two formats, 3.3 DOS and ProDOS. Both versions look alike to the user, but there are significant internal differences. They sell for \$25 each or \$39.95 for both versions together (plus \$2 shipping). If you have purchased either version, you may get the other for \$15 plus \$2 shipping. Because I feel that you should be able to try software before you buy it, I allow *unmodified* BASIC diskettes to be distributed to your friends. YOU MAY NOT, HOWEVER, DISTRIBUTE ANY WRITTEN DOCUMENTATION ABOUT BASIC (including this manual) in any form -including magnetic media. If your friends like and plan to use BASIC, please encourage them to become a registered owner. They can do so by ordering this manual (and all the back newsletters) for \$19.50. Remember, registered owners may receive free newsletters and purchase special BASIC products such as those described below. Even though I allow BASIC diskettes to be distributed as *shareware*, you may not do so with any of the other BASIC programs offered through me. BASIC took thousands of hours to write. Please confirm my trust by honoring the above conditions.

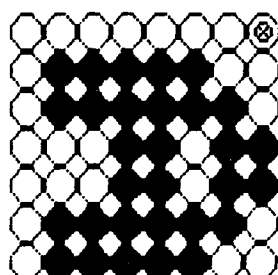
BASIC TEXTBOOK

One of the most popular BASIC products is a text book called *Structured Programming With BASIC*. It is 142 full-sized pages and spiral bound. Although it assumes you know very little about programming, it takes you through such subjects as sorting, searching, disk files, graphics, and even how to write your own "TINY" interpreter and compiler. It is written in an informal, easy-to-read style with the figures imbedded in the text for easy reference. It sells for \$30 plus \$3.50 shipping (\$10 foreign) and that includes a diskette with all the example programs typed in for you. The diskette presently only comes in 3.3 DOS format. If you don't have 3.3 BASIC add \$5 to your book order and I'll include a copy on the book diskette. I don't feel I have to apologize for the quality of the book, but you should know that all the text and figures were generated with my computer and the book was printed at my local print shop. As with all my products, it comes with a money-back guarantee.

UTILITY DISKETTES

A special utility diskette is available for \$20 plus \$2 shipping. You may specify ProDOS or 3.3 or get both for an additional \$5. One of the items it contains is a character editor so you can create your own characters for the MIRS BASIC character set. The figure below shows a screen dump from the editor.

B. BASIC CHARACTER GENERATOR/EDITOR

CHARACTER GRID	EDIT GRID		
<pre> @ 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 c i s </pre>	<pre> CURSOR MOVERS JK JK H-HELP </pre> 		
FILE: BOLD TYPE			
KEY CHR\$(65)			
C-CREATE	L-LOAD	F-FILL	R-RESTORE
C-DELETE	S-SAVE	F-ERASE	R-WRITE
	A-QUIT		

In addition to the editor itself, you also get several new character sets ready to be used in your programs. You get a **BOLD** set, an **ITALICS** set, and an **UNDERLINE** set. You even get several combination sets such as **NORMAL/BOLD** and **NORMAL/ITALIC**. These combination sets are upper case only because the lower case letters have been replaced with the alternate set. They make it very easy to mix typestyles in your BASIC programs. If you have a file, file, or GS, these sets are as easy as typing lower case letters. If users can access these sets with a POKR 243,32. After the POKR, all MIRS PRINT statements will be in lower case (or bold or italic). Use POKR 243,0 (or TEXIT) to return to the normal state. The figure below shows a screen dump of the pre-defined character sets.

THIS IS THE CGE EXAMPLE. A LIST OF THIS PROGRAM WILL SHOW YOU HOW WE CHANGED STYLES OF TYPE. IF YOU USE THE SUBROUTINE "STYLE" JUST AS WE HAVE IT YOU SHOULD HAVE NO PROBLEMS IN SWITCHING SETS. THIS IS NOT NECESSARILY THE EASIEST WAY OF INTERLACING SETS BUT IT SHOULD ALWAYS WORK.

The utility disk also contains a set of programs called SHRINK, EXPAND, and SHORTEN. You may be familiar with optimizing programs for Applesoft that shrink programs by combining many statements on one line. These programs won't work with BASIC because of the special rules about multi-statement lines. The utilities disk has two SHRINK programs that can optimize your BASIC programs. SHRINK combines lines when possible, but it does not remove remarks from your program. SHRINK, RM combines lines and removes all RM statements. It is important to have both choices available to you because the diskette also contains an EXPAND program that restores a shrunk program to its original size so it can be edited. All three of these programs are very easy to use. Just BRUN the one you need, and the program in memory will be altered in a couple of seconds. That's all there is to

It. You can shrink or expand at any time, without saving or loading or anything. Most programs shrink to about 80% of their original size and run a little faster. If you always shrink before saving, your disks will seem 20% larger.

The last program on the utilities disk is SHORTRN. It shortens procedure names. As you know, all the characters in a procedure name are significant which encourages you to use long meaningful names. Unfortunately, very long names take up memory and slow execution. SHORTRN is a BBASIC module that can be merged into your program. It converts all procedure names to AA, AB, AC, ..., BA, BB, etc. When a program is both shortened and shrunk, it can be reduced in size as much as 50% (I average about 30%). You probably won't want to shorten all your programs, but if you need more space for your variables then it can be invaluable.

As stated earlier, all four of these utilities can be purchased by registered owners for only \$20 plus shipping. As with all BBASIC products, the diskette is unprotected. I think you will find them a useful addition to your BBASIC library.

PROCEDURE DISKETTES

If you develop general purpose procedures (such as those discussed in Chapter 4) and would like to share them with others, please send them to me on diskette. Use REM statements to document your routine and add an example program if appropriate. When I get enough routines I will make a diskette available to registered owners for a small fee. If you have submitted a routine, you will get a free copy of the diskette. As of the writing of this manual I do not have enough routines. When I do I will announce the diskette in one of the newsletters.

SCHOOLS NEED BBASIC

Since so many schools have purchased at least one copy of BBASIC, I am setting up a special quantity price schedule and licensing arrangements. If your school would like to license BBASIC for their machines or would be interested in making special BBASIC packages available to the students, then write and request a school price schedule.

Products and Prices

BBASIC (3.3 or ProDOS).....	\$25.00
Upgrade to other version.....	\$15.00
Both versions.....	\$39.95
Registration and newsletters (no disk).....	\$19.50
Character/Editor & Shrink/Expand.....	\$20.00
Final Draft of text book (3.3 DOS only).....	\$30.00
Add \$2 (\$5 outside the U.S) shipping and handling for all non-book orders.	
Add \$3.50 (\$10 outside the U.S.) for each order then includes one or more books.	

Mail your check to:
John Blankenship, P.O. Box 47934 Atlanta GA 30362

QUANTITY DISCOUNTS AVAILABLE FOR SCHOOLS AND USER GROUPS

CHAPTER 11 COMMERCIAL PROGRAMS

If you write a commercial program using BBASIC, you may include a copy of the BBASIC system on your diskette as long as you follow these rules.

1. Your diskettes must include all of the programs from the BBASIC system master.
2. The BBASIC opening menu must be available so that the user can select and read the BBASIC documentation and advertisement asking them to become registered owners for \$19.50.
3. Your documentation must state why you chose to write your program in BBASIC as opposed to AppleSoft and encourage the users to become registered if they wish to use BBASIC for purposes other than your program.
4. If you wish to have a turn key system that does not provide the advertisement information, please check with me about a low cost licensing agreement.
5. You must send me a copy of your program and documentation.

CHAPTER 12 SUMMARY OF COMMANDS

This summary of BASIC commands can serve as a quick reference. For more details refer to the earlier chapters. In addition to these commands, you can use any Applesoft command except for HGR2.

Modular Construction

DEFINE Used to define the beginning of a module (subroutine).

Syntax: DEFINE "NAME"

Quotes are only required if the name contains a reserved word, but are always recommended. NAMES may not contain commas or colons.

FINISH Used to mark the end of a module. Only one FINISH is allowed per module.

PERFORM Causes the execution of a module.

Syntax: PERFORM "NAME"

COMPILE Must occur in the program before the first PERFORM. Causes a table of addresses of modules to be created. If left out of a program then PERFORM may cause strange errors.

CONTROL STRUCTURES

Because of the many loop structures of BASIC, any program can be written without GOTO's. GOTO will execute properly so that Applesoft programs will run under BASIC, but the use of GOTO has been discouraged by having RETURN not support it.

WHILE - ENDWHILE (check at beginning of loop)

Used to create a WHILE loop.
Syntax: WHILE A\$=B\$+C\$

body of loop
ENDWHILE

REPEAT - UNTIL (check at end of loop)

Used to create a UNTIL loop.
Syntax: REPEAT

body of loop
UNTIL A=B

LOOP - ENDLOOP - EXITWHEN

Used to create an infinite loop or one that can be exited at any point. EXITWHEN is optional and any number of EXITWHENs may be used if needed.
Syntax: LOOP

body of loop
EXITWHEN A\$="DONE"
ENDLOOP

WHEN - BLSB - ENDWHEN
Expanded version of the IF statement.
Syntax: WHEN A*B THEN
do this if
true
BLSB
and this if
false
ENDWHEN

note: Applesoft's IF is still valid. If should not be used however, with the PERFORM statement. Use WHEN without BLSB as shown below.

WHEN A=B THEN
PERFORM "NAME"
ENDWHEN

CASB Allows modules to be PERFORMed based on the value of a variable (similar to Applesoft's ON A GOSUB).

Syntax: CASB A: "NAME1", "NAME2",etc

Note: All control structures may be nested in any combinations. Actual depth allowed is determined by the stack.

HIRES GRAPHIC EXTENSIONS

HGR - TEXT

Similar but not identical to Applesoft HGR and TEXT. HGR allows graphics and text but no scrolling. HGR sets color to 3 and does not clear screen. TEXT automatically performs a HOME. Full use of all PRINT and TAB commands are supported in HGR. (HGR2 is not supported.)

NORMAL - INVERSE - HOME - REVERSE

Provide the NORMAL, INVERSE, and HOME functions in both TEXT and HGR modes. REVERSE prints in the reverse color of the background (HGR mode), but should not be used in the immediate mode. FLASH will produce strange characters if used in the HGR mode.

DRAW.USING

Draws based on a string variable that contains any of the following characters.

- U - move up
- D - move down
- L - move left
- R - move right
- I - move up and right
- J - move down and right
- K - move down and left
- M - move up and left
- N - ON, plot with moves (default)
- F - OFF, don't plot, just move
- 1-9 - cause all future plots to be done from one to nine times.

System remains in this mode till a new number is found.
SPACE - may be used to improve readability

note: Draw starts from the last HPLOT or the last DRAW.USING and uses the last color plotted.

After a HOME, DRAW.USING starts in the center of the screen. Every DRAW.USING starts in the ON mode automatically.

Syntax: A\$="GURDL F9R3D M1JHM"
DRAW.USING A\$
REM draws a square and a diamond 3 dots apart. The square will have 6 dots per side and the diamond will have 3.

BOX - BOXFILL

Creates the outline of, or a solid box of the last color used. Specify the coordinates of the upper left hand and lower right hand corners of box.

Syntax: BOX 20,25,100,150
BOXFILL X,Y,W,H,2#Z

OTHER GENERAL COMMANDS

GRT

Appears exactly like AppleSoft's GRT, but works with my line editor. Also works with 80 columns, but cursor will always appear on an even column regardless of real position.

INKEY

If a key is pressed it performs a GRT, otherwise execution continues without action.

Syntax: INKEY A\$

INLINE

Just like INPUT except that commas and quotes are allowed in the input data. (You may *not* specify a prompt as with INPUT.)

Syntax: INLINE A\$

HSCRN

Reads on/off status of a HIRBS coordinate.

Syntax: HSCRN X,Y,Z
(Z=1 if point X,Y is ON, Z must be REAL)

MERGE

Adds a program on disk to the program in memory. Line numbers are not altered, use RENUM before editing. (a space *must* occur between MERGE and "NAME")

Syntax: MERGE "NAME"

FILE

Saves a named subroutine from the program in memory to the disk. As with MERGE, a space must separate FILE and "NAME"

Syntax: FILE "NAME"

RENUM

Renumbers program starting at 1000 by 10's

Syntax: RENUM

LIST

Lists the program starting from a line number, a module NAME, or the beginning of program. (If you need to list one line use EDIT.) ESC or RETURN stops listing, any other key will pause. Listings are automatically indented based on first command on a line (if you use a FOR-NEXT on the same line, start it with a colon). All BASIC commands must be first on the line to list properly (a special packing program is under development). LIST will turn off any slot before printing is actually started. If you reset before a list you will see the & commands.

Syntax: LIST or LIST 100 or LIST "NAME"
Any key will pause, RETURN or ESC to ABORT.

LIST

Same as LIST but sends output to slot 1

INSTR\$

Finds the Nth occurrence of X\$ in Y\$ and places position found in the REAL variable C. C will = 0 if no match is found.

Syntax: INSTR\$ N,X\$,Y\$,C

COLLECT Forces a fast garbage collection if available memory is less than 1K.

EDIT

Presents lines for alteration starting at the point specified by the command. Use ESC or CTL-X to abort

Syntax: EDIT or EDIT 100 or EDIT "NAME"

RANDOMIZE

Reseeds the random number generator. Generally use RANDOMIZE only once in a program (valid after any input).

DISK

Allows DOS commands without CHR\$(4). (You'll be surprised at how much you will like this command after you used it.)

Syntax: DISK "OPEN NAME"

CHAIN

Syntax: CHAIN "NAME"

CHAIN

Runs a program from the disk without destroying the values of present variables. Requires Apple's (TM) CHAIN program to be on your diskette. CHAIN can be found on a DOS 3.3 master diskette.

Syntax: CHAIN "NAME"

SWAP

Swaps the values in any two variables of same type.

Syntax: SWAP A,B

VECTOR

Sets up address to jump to other programs. Jump will occur if command name is different than those of BASIC.

Syntax: VECTOR 8192

BELL

Produces the apple bell tone.

SOUND

Produces sounds of Duration D (0-127), Freq F (0-191) and Effect E (0-255). Effect of 0 gives normal tone. Duration gives same length for all frequencies. Effects close to 1 and 255 are the most interesting.

Syntax: SOUND D,F,E

PRINT.USING

Rounds off numbers and prints to desired decimal places. Use \$ to indicate field length. The mask may be a string variable.

Syntax: PRINT.USING "\$\$.##.##":X

DEL.ARRAY

Deletes an array to free memory or allow it to be re-dimensioned.

Syntax: DEL.ARRAY B\$

SORT

Sorts a 1 or 2 dimensional array into ascending order. Two dimensional arrays are sorted on the 1st column, and the other columns are ordered with the 1st. First column is the 0 column in the array. (And don't forget the 0 elements).

Syntax: SORT A\$

RESTORE

SETS starts of DATA read pointer to the present line. Allows each procedure to have its own DATA statements.

SEARCH

Searches the array B for the Nth occurrence of the item X. Position found is placed in the real variable C. For multi-dimensional arrays the search is in the order of the dimensions, so use a formula to convert "position" to "row-column". String arrays will match if the item being searched for, matches the left hand side of an array element.

Syntax: SEARCH N,X,B,C

WIDTH.40 and WIDTH.80 These commands select 40 and 80 column modes on IIC and IIC only. RTAB will not work past 40 columns (use F0RK 1403.POS as Apple recommends). Once a mode is selected, it will be the default for the TEXT command. HGR text will always be 40 columns.

ERR.OFF Turns off ONERR GOTO
 HANDLE.ERR Turns off ONERR and fixes ONERR bug.
 Should be used to start the module that handles errors.

INDEX

41, 7,8,10,29
 Accelerator cards, 25
 Ampersand (&), 2,7,10,32
 Applesoft, 1,5,7,10,11
 Autonom, 9
 BELL, 18,42
 BOX, BOXFILL, 17,41
 CASB, 12,40
 CHAIN, 27,42
 Character Editor, 35,36
 COLLECT, 20,42
 COMPILER, 7,39
 Compiler, 1,25
 Control structures, 2,6,39,40
 DEFINE-FINISH, 5,39
 DRL.ARRAY, 20,42
 DISK, 19,42
 DRAW.USING, 5,17,40
 EDIT, 9,10,42
 Editor, 2,4,5,7-10
 Errors, Error handling, 20,30,31
 FILE, 13,41,30
 FLASH, 16
 GRT, 19,41
 GOTO, GOSUB, 7,9,11
 Graphics, 2,4,16,17
 Hardcopy, see PR, see LIST, 5
 HGR, 16,40
 HOME, 6,16,40
 HSCRN, 18,41
 IN#, 29
 INKEY, 19,41
 INLINE, 19,41
 INSTR\$, 24,41
 Interpreter, 1,25
 INVERSE, 16,40

Library routines, see Modules, 13
 LIST, 5,9,41
 LIST 5,10,41
 Local variables, 27,28
 LOOP-EXIT/WHEN-ENDLOOP, 11,39
 Memory usage, 2,32,33
 MERGE, 13,41
 Modules, 12,13,39
 Newsletter, 3
 NORMAL, 16,40
 PERFORM, 5,66,13,39
 PR#, 5,10,19
 PRINT.USING, 20,42
 RAM drives, 25
 RANDOMIZE, 20,42
 RENUM, 9,41
 REPEAT-UNTIL, 12,39
 Reset, 7,10
 RESTORE.HERE, 15,42
 REVERSE, 16,17,40
 SEARCH, 24,42
 SELF-addressed-stamped-envelope, 3
 SHRINK/EXPAND/SHORTEN, 36,37
 SORT, 21-23,42
 SOUND, 18,42
 Structured Programming, 2,3,11-15
 Structured Programming With BASIC, 3,35
 SWAP, 19,42
 TAB#, 16,40
 TEXT, 6,7,16,40
 VECTOR, 28,42
 WHEN-ELSE-ENDWHEN 5,6,11,12,40
 WHILE-ENDWHILE, 12,39
 WIDTH, 40, WIDTH.80, 16,33,43
 Windows, 25-27

Blankenship BASIC

FOR THE APPLE II*, IIC, IIC, and GE

Copyright January 1984
 by John Blankenship
 (diskette and manual modified 1985, 1986, 1987)

P.O. BOX 47934 Atlanta GA 30362

No part of this document may be reproduced in any form.
 Apple and Applesoft are trademarks of Apple Computer Inc.
 You may freely give unaltered copies of BASIC to others

but ... this documentation.

I believe LOW prices are the way to beat piracy.
 HELP PROVE ME RIGHT

CHAPTER	TITLE	PAGE
1	Introduction to BASIC	1
2	Tutorial	4
3	The Editor	8
4	Modular and Structured Programming	11
5	Text, Graphics, and Sound	16
6	Faster and Easier	19
7	Power Users	25
8	Handling Errors	30
9	Technical Specifications	32
10	Other BASIC Products	35
11	Commercial Programs	38
12	Summary of Commands	39

DISCLAIMER
 BASIC has been thoroughly tested, but programs of this size and complexity can always have errors. If you find an error you can duplicate, please send me a diskette copy of your program (include your version of BASIC) and a description of the problem and I will attempt to fix it.

I assume no responsibility for any damages, including lost profits or other incidental or consequential damages, arising from the use of, or inability to use, BASIC. My sole responsibility will be to replace the product or refund the purchase price, whichever is appropriate.

It is possible that there will be updates, additions, and utilities available for BASIC. Information about these topics, as well as helpful hints, can be found in the BASIC newsletters. This update of the manual includes the information found in the first six newsletters. Registered owners can receive free newsletters by keeping a self-addressed stamped envelope on file with me as described in Chapter 1.