$14.95

INTERMEDIATE APPLE

By Bill Parker

Take that intermediate step from elementary BASIC to machine language programming!

Intermediate Apple

by
Bill Parker

Illustrated by
Robert J. Peters

20660 Nordhoff Street
Chatsworth, CA 91311-6152
(818) 709-1202

ISBN 0-88190-241-1

Printed in U.S.A.

ACKNOWLEDGMENTS

To Bill Sanders for the methodology and motivation to finish this book.

Dedicated to Nancy and Z. Parker, two of my closest friends.

Table of Contents

EDITOR'S INTRODUCTION

William B. Sanders, Ph.D.

This book is for the computer user who understands the fundamentals of BASIC programming and is wondering what to do next. There is a point when the novice computer user reaches a plateau, where he or she decides whether or not to learn more about computer programming. At this point, one has little alternative than to make a giant leap into the world of machine level language or into various other higher level languages. Mastering elementary BASIC is often traumatic enough to dissuade the budding programmer from risking his life on the cliffs of machine language. So the decision is often limited to taking a leap into the morass of a new language or doing nothing at all.

This book offers another choice. It is an INTERMEDIATE step that will immensely improve Applesoft BASIC programming skills and provide a whole array of proven programming techniques. Yet at the same time, it deals with the familiar constructs of Applesoft BASIC. In fact, the purpose of this book is to make programming easier, not more difficult. Rather than looking at BASIC in terms of single statements, functions or commands, it shows the user how to deal with program blocks and modules. Small, simple programs are fine for learning how to program, but there will come a point at which you will want to write a useful, fairly large program. If you've spent any time at all programming, you must have LISTed others' programs and asked yourself, "How could they keep a big program like that straight?" This book shows you how.

You may well wonder how writing larger, more sophisticated programs can be easier than writing a small simple program. For the most part, a large program is nothing more than a well organized set of small programs, and the key to that is organization and structure. First, rather than rewriting an entire program every time you sit down at the computer this book shows you how to save and then re-use program modules that can be employed in several programs. With only a few program lines, it is possible to connect several previously written modules into a larger program. Thus, the larger program is actually simpler than blindly piecing together a small one. Also by using structured programming techniques, you will be able to see more clearly what you have done. For anyone who has written a huge

program and then gone back to it, one is often at a loss to remember what everything does and why it does it. By clearly documenting and arranging it, any Applesoft progam can be made clearer. Mr. Parker even provides some utilities to assist in making Applesoft more lucid.

In addition to showing the user how to attack a programming problem, this book shows how to master some of the more advanced features of Applesoft. In my introductory book, *The Elementary Apple*, I only wanted to get a new user started in programming. The book was for someone who brought home their computer and wanted to get started without too many tears. However, DATAMOST wanted a book that would take the next step in programming the Apple Computer. Therefore, many of the features that were just touched upon in *The Elementary Apple*, such as graphic animation, shape tables, POKEs, PEEKs and disk file handling, are explored in depth here.

Next, the book takes the first step toward advanced programming. It is not the giant leap described above, but rather it is an introduction to assembly language programming. The best way to introduce an assembly level program is to explain how to use an assembler, and that is exactly what it does. This gives the user a chance to take a look before the leap. You will be shown how to get an assembler up and working, along with some sample assembly level programs. Everything will be kept simple, and while you cannot expect to become an expert at assembly level programming, you will learn enough to get started.

Finally, the book concludes with a number of utility programs provided by the author, along with some suggested commercial programs. All of these programs are utilities to make programming easier, clearer and more efficient.

MEET THE AUTHOR

Bill Parker has the ideal background for creating a book such as this. With a solid background in both journalism and computers, he has both the ability to communicate clearly in English and to write computer programs. Bill is a staff writer for *Call-A.P.P.L.E.*, and former editor of the *Sandy Apple Press*, the club magazine of Apple Corps of San Diego. He has taught computer courses in both the University of California, San Diego and at San Diego State University Extension programs. He is currently a full-time computer consultant and writer.

CHAPTER 1

INTRODUCTION TO STRUCTURED PROGRAMMING ON THE APPLE

This book will take you from the point of being a fledgling Apple programmer and show you some important principles that can help you handle more complicated programming problems.

The method emphasized here is a technique known as *structured programming*: a "one-step-at-a-time" method of reducing big problems into smaller, more manageable ones.

The Hazards of Unstructured Programming on the Apple

Consider the following two programs:

Program 1:

```
100 I = I + 1
110 GOSUB 200
120 GOTO 100
130 :
200 PRINT I
210 IF I < 100 THEN 120
220 RETURN

RUN
1
2
3
4
5
6
7
```

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

?OUT OF MEMORY ERROR IN 210
```

Program 2:

```
100 FOR I = 1 TO 200
110 GOSUB 200
120 NEXT I
130 :
200 PRINT I
210 IF I < 100 THEN 120
220 RETURN

RUN
1

?NEXT WITHOUT FOR ERROR IN 120
```

Why are these programs generating nonsense error messages? After all, little Program 1 couldn't possibly consume all of the Apple's memory and line 100 of Program 2 contains a FOR statement as plain as the nose on your face. So why the errors?

These programs are typical examples of *unstructured programming*, a kind of a programming "by the seat of your pants" approach to problem solving. The problem with it is that it just doesn't follow the way your computer "thinks," and the error messages show it.

Another problem with this type of programming is that it is unclear. The bigger the program becomes, the harder it is to read and understand. This becomes an even bigger problem when someone unfamiliar with your program needs to enhance, customize or just learn from it. In extreme cases, *you may not even be able to understand your own program*, which could occur if you come back to it after a few months. Your memory of the intricate portions of your program will be gone, and you will then be unable to make sense of the more tangled portions of your own code.

Some computer scientists in the 1960's noticed these things and decided to take a long, hard look at the state-of-the-art in programming at the time. This is what they found:

- There was no organized, systematic way of even approaching a programming problem. Good programmers simply "dove in" (sound familiar?) and came up with something that worked — most of the time. Bad programmers floundered about even longer.

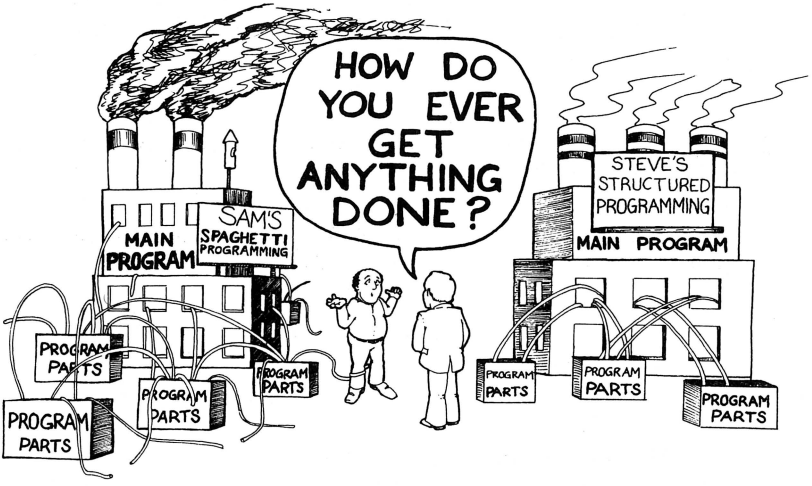
- There was no real assurance that a program was written "correctly" and would be reliable.

- There was no standardized vocabulary for describing the way a problem was turned into a program (this is known as *algorithm* development). You had to be intimately familiar with the computer language chosen by the program author to understand how he solved the problem.

- Programming projects became stalled for expensive periods of time when old programmers left the project and new ones came in to replace them. It took a certain amount of time just to be able to understand what the previous programmer had done and then to be able to continue from that point.

- Flowcharts were worthless. Although they were supposed to be done at the beginning of a problem to provide the programmer with an easy-to-follow graphical representation of the steps necessary to write the program, they were frequently left (if done at all) until after the program was up and running. This happened because the more complicated a program became, the more complicated (and unclear) the flowchart became.

- Trying to read a long program in an effort to understand how it worked was an exercise in futility, because control branched all over (through the use of the ubiquitous GOTO statement). This method of programming, which also prevented large programs from being broken into smaller, more manageable "modules," came to be known as *spaghetti programming*.



This sorry state of affairs led Edsger Dijkstra, one of the "Fathers of Structured Programming," to remark in the preface to his book, *A Discipline of Programming*:

> . . . on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even of any beauty at all, totally unfit for mechanical appreciation.

Fortunately, there is help! The same group of scientists who analyzed the problems caused by "normal" programming practices also came up with a way to avoid them. This method allowed programmers to write programs with "compelling and deep logical beauty" and has come to be known as structured programming.

Benefits of Structured Programming

We will now investigate the basic elements of structured programming in Applesoft. Among its benefits are:

1. Programs are more easily understood. Programs are easier to read and the logic flow is easier to follow.

2. The possibility of making errors is reduced. This is known as "anti-bugging." You may also have heard of this concept in the well-known commercial where it is stated, "problems are built out — not in."

3. Programs are easier to maintain. This makes it easier for anyone to understand, improve or enhance your program.

4. It's faster to code with structured programming. The logic is simple and straightforward.

5. It provides an easier transition to other higher level languages. Currently, most programs written with the aid of higher level languages use structured programming techniques.

6. It's easier to code large programs. Large programs can be broken into "modules" and given to different programmers for development. This speeds up the development of the program and makes it easier to coordinate the finished modules into one system.

The Three Control Structures

The benefits of structured programming are the direct result of a key discovery by researchers:

No matter how complicated a program is and no matter in which language a program is written, any program can be written with just three basic *control structures*: sequence, decision and loop.

As the name implies, control structures tell the computer which instruction to execute next. It's how you control the logic flow in your program that determines how clear and manageable the program is. Let's take a look at some examples of each kind of control structure:

SEQUENCE

True Control Structures

```
100 X = 3
110 Y = 6
120 PRINT "THIS IS A TEST"
```

Applesoft

```
100 X = 3
110 Y = 6
120 PRINT "THIS IS A TEST"
```

Here, you can see that there is no difference between true control structures and the structures that Applesoft provides. This is because sequential statements are the most basic commands that can be used on a computer; all languages must use them.

DECISION

True Control Structures

```
100 IF X > Y
    THEN PRINT "X > Y"
```

Applesoft

```
100 IF X > Y
    THEN PRINT "X > Y"
```

This kind of IF THEN decision is the most basic decision structure available; all languages have this structure in one form or another.

```
100 IF X > Y THEN

110 X = 3
120 Y = 6
130 PRINT "X > Y"
140 IF END
```

```
100 ON X > Y GOTO 110
    : GOTO 140
110 X = 3
120 Y = 6
130 PRINT "X > Y"
140 REM IF END
```

This is an example of a *multi-line* IF THEN structure. Only the most advanced BASICs have this feature. Note the use of the IF END command to tell the computer where the end of the IF block is. (Statements within the block are executed only if the IF condition evaluates to true.) Applesoft, which is a minimal BASIC, must be "forced" to perform the same type of control through the use of the ON GOTO statement. ON GOTO is used here instead of IF THEN because, unlike IF THEN, if the ON condition is false, control goes to the next statement within the current line.

```
100 IF X > Y THEN

110 X = 3
120 Y = 6
130 PRINT "X > Y"

140 ELSE
150 Z = 5
160 IF END
```

```
100 ON X > Y GOTO 110
    : GOTO 140
110 X = 3
120 Y = 6
130 PRINT "X > Y"
135 GOTO 160
140 REM ELSE
150 Z = 5
160 REM IF END
```

With an IF THEN ELSE structure we can specify which of the two blocks of code will be executed. If the condition is true, the first block is executed and the second block is skipped. If the condition is false, the first block is skipped and the second block is executed. Notice here how closely Applesoft's ON GOTO statement simulates the true IF THEN ELSE structure. In fact, it can be read as:

```
ON X > Y GOTO 110 : GOTO 140 ⟶
IF X > Y THEN 110 ELSE 140
```

True Control Structures

```
100 FOR I = 1 TO 10
110 X = X + 1
120 NEXT I
```

Applesoft

```
100 FOR I = 1 TO 10
110 X = X + 1
120 NEXT I
```

As you can see, there is no difference between the true FOR NEXT loop and the FOR NEXT loop provided by Applesoft.

```
100 REPEAT
110 INPUT ANS$
120 UNTIL ANS$ = "STOP"
```

```
100 REM REPEAT
110 INPUT ANS$
120 ON ANS$ = "STOP"
      GOTO 130 : GOTO 100
130 REM UNTIL
      ANS$ = "STOP"
```

The REPEAT UNTIL loop is known as a "trailing loop" because the decision to leave the loop is made *after* the body of the loop has been executed. This means the body of a REPEAT UNTIL loop is executed at least once.

```
100 WHILE ANS$
      < > "STOP"
110 INPUT ANS$

120 WEND
```

```
100 REM WHILE ANS$ < >
      "STOP"
110 ON ANS$< > "STOP"
      GOTO 120: GOTO 140
120 INPUT AN$
130 GOTO 100
140 REM WEND
```

A WHILE WEND loop is known as a "leading loop" because the decision to execute the loop is made *before* the body of the loop is executed. This means it is possible to completely avoid executing the WHILE WEND loop. In case you are not familiar with the syntax, "WEND" simply means W(hile) END and marks the end of the loop body.

The Problem With GOTOs

You may have heard that GOTOs are the archenemy of structured programming. In fact, if you look at the sample true control structures, you'll notice that there is not a single GOTO in the entire lot. This notion may strike you as a bit odd, especially if you are used to working with Applesoft which requires GOTOs to perform loops and decisions.

When Applesoft was created, it was designed with a minimum of features. Some of the first features to go were structured concepts, which were thought at the time to be unnecessarily space and time-consuming. Three control structures were built into Applesoft: IF THEN, GOTO and FOR NEXT, because any of the "true" control structures could be simulated by these. (GOSUB is merely an extension of regular sequential programming.)

A problem quickly arose with the GOTO statement. It allowed a "quick and dirty" means of transferring control throughout the program and encouraged a "program first, think later" mentality. If you got in a jam while writing a program, you simply put in a GOTO to get out! As a result, programs quickly degenerated into a mass of twisted and even unused code.

A second problem with GOTOs in Applesoft is that they slow down your program. To execute a GOTO statement, Applesoft finds the line number by usually starting at the beginning of the program and searching downward. GOTOs at the end of a large program, then, are notoriously slow. The problem extends to other versions of the GOTO statement, such as:

```
IF TRUE THEN 100
GOSUB 100
ON TRUE GOTO 100
```

Finally, GOTOs complicate program editing. If you happen to delete a line that is the target of a GOTO statement, your program will crash when it tries to execute the GOTO! True control structures utilize the concepts of blocks of code that are delineated by beginning and ending commands. For example, REPEAT marks the beginning of a block of code to be executed and UNTIL marks the end, thus making GOTOs and destination line numbers unnecessary.

ASA: An Alternative to GOTOs

In Chapter 10 of this book, you'll see a program that allows you to use true structured programming with Applesoft. This program is called, "ASA," which means "Ampersand Structured Applesoft." It works by using the "&" character to extend Applesoft's command set. Here are some examples:

DECISION

ASA

```
100 & IF X > Y THEN

110 X = 3
120 Y = 6
130 PRINT "X > Y"
140 & END
```

```
100 & IF X > Y THEN

110 X = 3
120 Y = 6
130 PRINT "X > Y"

140 & ELSE
150 Z = 5
160 & END
```

Applesoft

```
100 ON X > Y GOTO 110
    : GOTO 140
110 X = 3
120 Y = 6
130 PRINT "X > Y"
140 REM IF END
```

```
100 ON X > Y GOTO 110
    : GOTO 140
110 X = 3
120 Y = 6
130 PRINT "X > Y"
135 GOTO 160
140 REM ELSE
150 Z = 5
160 REM IF END
```

LOOP

ASA

```
100 & RUN
110 INPUT ANS$
120 & STOP IF
    ANS$ = "STOP"
```

Applesoft

```
100 REM REPEAT
110 INPUT ANS$
120 ON ANS$ = "STOP"
GOTO 130 : GOTO 100
130 REM UNTIL
    ANS$ = "STOP"
```

```
100 & ON ANS$ < >
    "STOP"
110 INPUT ANS$

120 & CONT
```

```
100 REM WHILE
    ANS$ < > "STOP"
110 ON ANS$ < > "STOP"
    GOTO 120 : GOTO 140
120 INPUT ANS$
130 GOTO 100
140 REM WEND
```

Notice that in the loops, RUN STOP is used instead of REPEAT UNTIL and ON CONT is used instead of WHILE WEND. This is done to reduce the amount of coding necessary to interpret the commands following the "&" character. To keep the explanation brief, Applesoft reduces a set of "reserved words" like RUN and STOP to one byte "tokens." A one byte character is infinitely easier to handle than a multi-byte word like, "REPEAT", which is not normally recognized by Applesoft as a valid command.

With ASA, you'll find that your Applesoft programs are easier to create and change. In addition, they execute faster than the GOTO equivalent. Examples in this book will usually use normal Applesoft. You are free to choose the method you find more comfortable.

Nesting

Despite the many anti-bugging features of control structures, there is still one way programs are commonly fouled up: the lack of proper "nesting." There may be occasions in which you will have to place one control structure inside another. This is called "nesting":

```
100 FOR LINE = 1 TO 5
110   FOR CLMN = 1 TO 20
120     PRINT "*";
130   NEXT CLMN
140   PRINT
150 NEXT LINE
```

Output:

```
******************
******************
******************
******************
******************
```

In this program, which prints out a box of "stars," there is a FOR NEXT loop (lines 110-130) nested inside another FOR NEXT loop (lines 100-150). As explained so eloquently by William B. Sanders in his book, *The Elementary Apple*: (See pg. 3-16 for an interesting picture of this.)

> Think of nested loops as a series of fish eating one another, the largest fish's mouth encompassing the next largest and so forth down to the smallest fish.

Any control structure can be nested inside any other. With proper nesting, however, there is only one entry point into a control structure and one exit point. Troubles begin when you violate this principle. (The first two programs at the beginning of this chapter are classic examples of violations of this rule.) The concept of proper nesting will be brought out more fully in a later chapter on flow diagramming.

A Word On Variable Names

The previous "draw stars" program was easy to understand because:

1. Each statement was on a separate line.
2. Long, self-descriptive variable names were used.
3. Loop index variables were used.
4. Numbers were used instead of variable names:

```
100 FOR LINE = 1 TO 5
110   FOR CLMN = 1 TO 20
120     PRINT "*";
130   NEXT CLMN
140   PRINT
150 NEXT LINE
```

However, it can be made to run significantly faster by:

1. Putting as many statements on one line as possible.
2. Using at most two letters for a variable name.
3. Not using loop index variables in NEXT statements.
4. Using variables instead of numbers:

```
100 N1 = 1
    : N5 = 5
    : NT = 20
    : FOR LN = N1 TO N5
    :   FOR CL = N1 TO NT
    :     PRINT "*";
    :   NEXT
    :   PRINT
    : NEXT
```

There are commercial products on the market now that can "optimize" Applesoft by changing the first version above into the second version. As you can see, however, clarity is beginning to suffer at the expense of execution speed. As will be explained in the next section, it is a good idea to have two versions of the program you are working on, one "for show" and the other "for execution."

Program List Formatting

You may have noticed that the programs listed here look a lot "cleaner" than the way you can get programs to appear on your

screen with Applesoft's LIST command. This is due to the fact that the LIST routine in Applesoft is very short and can only provide a minimum of formatting. The problem becomes very apparent with more complicated programs:

```
210   LET D$ = CHR$ (13) + CHR$
      (4):OP$ = D$ + "OPEN":DE$ =
      D$ + "DELETE":WR$ = D$ + "WR
      ITE":CL$ = D$ + "CLOSE":Q$ =
      CHR$ (34):N$ = "CONVERT": PRINT
      OP$N$;DE$N$;OP$N$;WR$N$: POKE
      33,33: LIST 50000 -: POKE 3
      3,40: PRINT "RUN 50000": PRINT
      D$"CLOSE CONVERT": END

50000 D$ = CHR$ (13) + CHR$ (4)
      : GET N$: PRINT N$: HOME : INVERSE
      : PRINT "        C O N
      V E R T        ": NORMAL
      : PRINT

50001 PRINT "WHAT NAME WOULD YOU
      LIKE TO SAVE THE CONVERTE
      D (TEXT FILE) VERSION OF THE
         PROGRAM AS?": PRINT

50002 INPUT "REMEMBER TO PUT A '
      .T' ON THE END OF THENAME TO
      MAKE IT DIFFERENT FROM THE
      APPLESOFT NAME: ";N$

50003 PRINT D$"OPEN ";N$;D$"DELE
      TE ";N$;D$"OPEN ";N$;D$"WRIT
      E ";N$: POKE 33,33: LIST 0,4
      9999: LIST 50005 - : TEXT : PRINT
      D$"MONO"

50004 HOME : INVERSE : PRINT "DO
      NE!": NORMAL : PRINT : PRINT
      "  A TEXT FILE VERSION OF T
      HIS PROGRAM": PRINT "NOW EXI
      STS UNDER THE NAME OF:": PRINT
      : PRINT N$: PRINT D$"CLOSE "
      ;N$: DEL 50000,50004
```

Now, compare the previous program with this version:

```
100 REM  ************************************
110 REM  *       == MAKE CONVERT ==         *
120 REM  *      RUN THIS PROGRAM TO         *
130 REM  *        CREATE "CONVERT".         *
140 REM  *       'EXEC CONVERT' WILL        *
150 REM  *        THEN CONVERT ANY          *
160 REM  *       PROGRAM IN MEMORY          *
170 REM  *          INTO A TEXT FILE        *
180 REM  *          UNDER ANY NAME          *
190 REM  *              YOU LIKE.           *
200 REM  ************************************

210    LET D$ = CHR$(13) + CHR$(4)
       : OP$ = D$ + "OPEN"
       : DE$ = D$ + "DELETE"
       : WR$ = D$ + "WRITE"
       : CL$ = D$ + "CLOSE"
       : Q$ = CHR$(34)
       : N$ = "CONVERT"
       : PRINT OP$N$; DE$N$; OP$N$; WR$N$
       : POKE 33,33
       : LIST 50000-
       : POKE 33,40
       : PRINT "RUN 50000"
       : PRINT D$"CLOSE CONVERT"
       : END
50000 D$ = CHR$(13) + CHR$(4)
       : GET N$
       : PRINT N$
       : HOME
       : INVERSE
       : PRINT "        C O N V E R T        "
       : NORMAL
       : PRINT
50001 PRINT "WHAT NAME WOULD YOU LIKE
      TO SAVE THE      CONVERTED (TEXT FILE)
      VERSION OF THE       PROGRAM AS?"
       : PRINT
50002 INPUT "REMEMBER TO PUT A '.T' ON THE
      END OF THE NAME TO MAKE IT DIFFERENT
      FROM THE        APPLESOFT NAME: ";N$
```

```
50003 PRINT D$"OPEN ";N$; D$"DELETE ";N$;
      D$"OPEN ";N$; D$"WRITE ";N$
      : POKE 33,33
      : LIST 0,49999
      : LIST 50005-
      : TEXT
      : PRINT D$"MONO"
50004 HOME
      : INVERSE
      : PRINT "DONE!"
      : NORMAL
      : PRINT
      : PRINT "A TEXT FILE VERSION
        OF THIS PROGRAM"
      : PRINT "NOW EXISTS UNDER THE
        NAME OF:"
      : PRINT
      : PRINT N$
      : PRINT D$"CLOSE ";N$
      : DEL 50000,50004
```

You'll notice that the clarity of this program has been greatly enhanced simply by controlling the spacing between statements. Both programs are identical and run the same. The first was listed with Applesoft's built-in LIST command, the second was converted into a text file and edited with a word processor. (Any word processor that can handle text files will do; most Apple owners already own at least one word processor.)

This brings us to an important point: Applesoft will fight you tooth and nail over structured programming. This is because it was developed before the concept of structuring was thought desirable for personal computers and also because it had to be a "stripped-down," small-as-possible language to fit within a small portion (12K) of memory in the Apple's ROM. (Other more powerful BASICs typically consume at least twice as much memory and run slower than Applesoft.)

In order to follow the idea of illustrating how structured programming increases clarity, programs listed in this book will be formatted as shown above. You can perform the same process by using the above program (that's why it's listed here) to convert

programs into text files and then using a word processor to properly format your listing. (The text file version can be RUN just like the Applesoft version simply by first EXECuting it back into memory.)

This means that to do things right, you will wind up with *two* files for every properly structured Applesoft program you write: an "execution copy" (in Applesoft) that will be directly RUNable in a "stripped-down" version and a "presentation copy" (a text file) that is properly REMarked and formatted for maximum clarity. This is especially important when you need to explain how your program works to someone else, such as in commercial documentation, magazine articles and the like.

Summary

Unstructured programming is often called programming "by the seat of your pants." It requires no forethought and generally leads to **spaghetti code**. Among its drawbacks:

* Hard to understand large programs.
* No assurance of reliability.
* No standardized vocabulary.
* Unable to break program into modules.
* Flowcharts are worthless.

Structured programming is an example of "think first, program later." Among its benefits:

* It is easy to understand.
* It is reliable.
* It uses a standardized vocabulary.
* It breaks a program into modules.

* Flow diagrams are helpful.

* It is easier to code.

There are three basic control structures in structured programming: **Sequence, Decision** and **Loop**. Decision can be expressed by IF THEN and IF THEN ELSE. Loops can be expressed by FOR NEXT, REPEAT UNTIL, and WHILE WEND. ON GOTO must be used in Applesoft to simulate the IF THEN ELSE, REPEAT UNTIL and WHILE WEND structures.

GOTOs are to be avoided whenever possible. They have the following drawbacks:

* They encourage sloppy programming.

* They slow down program execution.

* They complicate editing.

Unless & (ampersand) commands are used to extend Applesoft's command set, GOTOs must be used to simulate certain control structures. Their use must be limited to that function.

Nesting refers to the placing of one control structure inside another. There must be one entry point and one exit point for every control structure or an improper nesting condition will result and strange syntax errors may occur during program execution.

Two programs may be needed when using structured programming in Applesoft: a **Presentation Copy** (for clarity) and an **Execution Copy** (for speed). Here are two lists of elements that are characteristic of each type:

Presentation Copy (text file)

1. Each statement is listed on a separate line.

2. Self-descriptive variable names are used.

3. Long loop index variable names are used.

4. Numbers are used instead of variable names.

5. Plenty of REMarks are used.

Execution Copy (Applesoft program)

1. Two letter variable names are used.

2. Short loop index variable names are used.

3. Variables are used instead of numbers.

4. No REMarks.

For Further Reading

Algorithms + Data Structures = Programs, Niklaus Wirth (Prentice-Hall, Englewood Cliffs, NJ, 1976). The classic, definitve statement on structured programming. Not for the beginning student, however. Just something to keep in mind when you are ready for it. May well "expand your programming horizons."

How To Write An Apple Program, Ed Faulk (Datamost, Inc., Chatsworth, CA, 1982). Good discussion on structured programming in Applesoft.

Introduction To Computers and Data Processing, Gary Shelly and Thomas Cashman (Ahaheim Publishing, Fullerton, CA, 1980). Textbook with in-depth coverage of general structured programming techniques. Numerous full color illustrations and flowcharts.

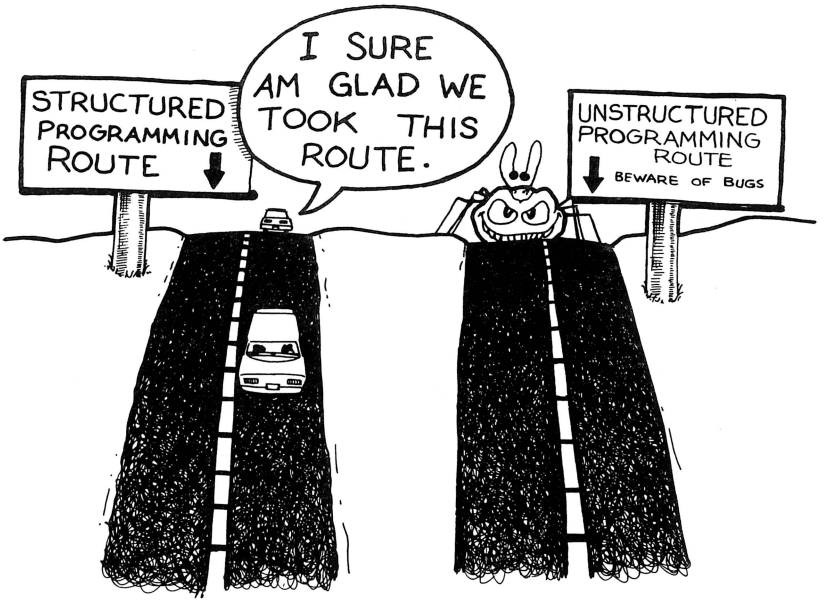
Problem Solving and Structured Programming In BASIC, Elliot Koffman and Frank Friedman (Addison-Wesley, Reading, MA, 1979). Excellent, down-to-earth book on structured programming in a wide variety of BASIC capabilities. Techniques presented here are easy to apply to Applesoft.

CHAPTER 2

PROBLEM SOLVING USING STRUCTURED PROGRAMMING

The Five Steps of Algorithm Development

Within the discipline of structured programming, there is a certain *way* to go about solving a problem. This way provides a clear and logical path to program problem solving. Let's take a look at how this is done.



A properly structured program is a *restated expression* of a problem in machine executable form. This means that the program is the problem, rewritten in a way that allows the computer to analyze it and print out an answer.

The method of deciding how to restate (solve) a problem is called *algorithm development.* It can be summarized in five steps:

Structured Programming: Problem Solving Steps

1. General Statement of the Problem.
2. Data Table.
3. Stepwise Refinement (Top Down Design).
4. Language Implementation.
5. Debugging/Modification.

Here's what each step means:

1. **General Statement of the Problem:** You must crystalize what it is you're trying to accomplish into one (or just a few) statement(s). At this point, you are just trying to focus on the essence of the problem. Do not worry about how to actually solve the problem or how to state it in a way "close" to how it would be written in Applesoft.

2. **Data Table:** On a separate sheet of paper, keep track of the variables used during program development. This is especially important with Applesoft, where only the first two letters in Applesoft variable names are significant. A helpful suggestion is to capitalize just the first two letters of variable names in the data table to make the significant portion of the variable name stand out better.

   For example, a program like the one presented below, which does not keep track of the fact that FILENO and FINISHED look the same to Applesoft, results in the destruction of the variable used to keep track of the file number:

```
10 FILENO = FILENO + 1
20 INPUT "FINISHED? (1=Y, 0=N): "; FINISHED
30 IF NOT FINISHED THEN 10
40 PRINT FILENO
```

   (You'll find that determining the output variables first makes the creation of the data table easier.)

   A data table also provides documentation for what the variables do. This can help someone else who is not intimately

familiar with your program to understand how it works. Finally, as we will see later, a data table can help you debug your program.

3. **Stepwise Refinement**: Continually restate the general statement of the problem in simpler steps, until it becomes very easy to translate the steps into actual Applesoft statements. This "top down" design "begins at the beginning," and works its way down into finer and finer detail until you finally wind up with executable code.

   The opposite of this process is known as "bottoms up" programming, which refers to beginning the programming process with simple Applesoft routines that you "know" work and gradually building them into a complicated program. This is the antithesis of structured programming: while it is useful for experienced programmers working with short, uncomplicated programs, it can result in all of the bad qualities previously mentioned that are associated with spaghetti programming.

   You'll notice that as you refine the problem, the language you use will gradually change from plain English to a sort of "computerese," e.g., "Draw a box of stars" becomes:

```
Step 1.  For 5 lines
Step 2.     For 20 columns
Step 3.        Print an "*" at the column and line
               indicated
```

   This language is known as *pseudo code* and will be discussed later.

4. **Language Implementation**: Depending on your familiarity with Applesoft, you will reach a point in the stepwise refinement process of a problem when you intuitively recognize what Applesoft statements are needed to solve the problem. You can then begin the actual writing of the program.

   In some cases, you will be unable to figure out how to translate a step into Applesoft. In this case, leave the step and assume that you will figure it out later. This is called a *stub procedure* and allows you to proceed with the rest of the program. An example of this would be:

```
10 INPUT "ENTER ANGLE: "; ANGLE
20 REM COMPUTE & PRINT TANGENT OF ANGLE
   HERE
30 INPUT "AGAIN? "; ANSWER$
40 IF ANSWER$ = "YES" THEN 10
```

In the above example, statement 20 is a stub procedure (using a REM statement), waiting for the programmer to figure out how to find (and print) the tangent of an angle. Stub procedures give you the advantage of being able to finish the rest of the program.

5. **Debugging/Modification:** It's rare when a program runs right the first time. Mistakes in a program are called "bugs" and the process of finding and removing them is called, "debugging." A useful way of catching bugs is to hand trace a program manually (i.e., without RUNning it on the Apple) and use the data table to record how the variables change. If the variables change the way you want them to, chances are pretty good that your program will run.

   Also, as you run your program, you'll probably notice ways it can be improved. As you make changes, the possibility of introducing a bug increases and the debugging/modification cycle begins all over again.

An Actual Example: The Shadow and the Building

Let's take a look now at an actual problem and see how the principles of structured programming can help solve it.

> You notice that a certain tall building casts a shadow on the ground and wonder if it's possible to find its height from the length of the shadow it casts.

Here's the procedure, broken down by steps:

STEP 1. General statement of the problem: Find the height of a building from the length of the shadow it casts.

STEP 2. Data Table: The data table will have to be filled out in parts as the program develops. For right now, this is what we know: (*Hint:* It's easiest to begin by defining the output variables first.)

Input Variables	Constants, Functions & Temporary Variables	Output Variables
SHADOW		HEIGHT

STEP 3. Stepwise Refinement:

1. Find the length of the shadow.
2. Use trigonometry to find the height of the building.
3. Print out the height of the building.

Step 2 stands out like a sore thumb. For most of us, it is just too vague to be translated directly into Applesoft. Let's recall our high school trigonometry and refine that step a bit.

The triangle formed by the building and the shadow it casts is called a *right triangle* because it contains a *right angle* (or an angle of 90 degrees). A principle of trigonometry states that the length of any side of a right triangle can be found from the length of any other side and one of the non-right angles of the triangle.

It's becoming rather obvious that we are going to need another input variable, the angle between the end of the shadow and the top of the building. This is called *the angle of elevation.* Let's update our data table to reflect this new addition:

Input Variables	Constants, Functions & Temporary Variables	Output Variables
SHADOW ANGLE		HEIGHT

Any trigonometry textbook will tell us that the tangent of the angle of elevation is equal to the height divided by the length. Restating this, we find that the height is equal to the tangent of the angle of elevation times the length. We are now ready to update our pseudo code with a substep (indicated by a ".1"):

1. Find the length of the shadow.
2. Use trigonometry to find the height of the building.
   2.1 Compute the height as the tangent of the angle of elevation times the shadow length.
3. Print out the height of the building.

Are we done yet? No, because of an idiosyncrasy of Applesoft: it expects all angles to be expressed in radians (there are "pi" (3.14) radians in 180 degrees) and not degrees. This means that we must add another substep:

1. Find the length of the shadow.
2. Use trigonometry to find the height of the building.
   2.1 Convert the angle of elevation into radians.
   2.2 Compute the height as the tangent of the angle of elevation (in radians) times the shadow length.
3. Print out the height of the building.

Problem Solving Using Structured Programming

We must also update our data table:

Input Variables	Constants, Functions & Temporary Variables	Output Variables
SHADOW ANGLE	PI = 3.14	HEIGHT

It should be obvious by now that we are ready to translate each step and substep directly into Applesoft. The process of stepwise refinement has made this task almost trivial. Notice also that you may have to rethink your algorithm several times during this process, inserting substeps, and clarifying what you mean.

STEP 4. Language Implementation: We now have everything we need to write our Applesoft program directly from the data table and the pseudo code:

```
100 PI = 3.14 : REM NOTE HOW CONSTANTS
    COME FIRST
110 INPUT "SHADOW LENGTH: "; SHADOW
120 INPUT "ANGLE: "; ANGLE
130 RADIANS = (ANGLE * PI) / 180
140 HEIGHT = TAN(RADIANS) * SHADOW
150 PRINT "BUILDING HEIGHT IS: "; HEIGHT

] RUN

SHADOW LENGTH: 100
ANGLE: 45
BUILDING HEIGHT IS: 99.9203991
```

STEP 5. Debugging/Modifications: This program ran right the first time! You'll find this experience to be more and more common as you adopt structured programming techniques.

The Four Standard Program Modules

The Building Height Estimator program can be "spiffed up" a bit to make each of its main parts easier to see and understand. There are four main parts or modules of a program: Identification, Data Table, Initialization and Main Program:

```
100 REM ****************************************
110 REM *          BUILDING HEIGHT             *
120 REM *            ESTIMATOR                 *
130 REM *          BY BILL PARKER              *
140 REM ****************************************
```

(Identification Module:
Contains the program's title, author and description of how it works.)

```
150 REM THIS PROGRAM COMPUTES
160 REM THE HEIGHT OF A BLDG
170 REM FROM THE LENGTH AND
180 REM ANGLE OF ITS SHADOW

190 REM +----------------+
200 REM :  DATA TABLE  :
210 REM +----------------+
```

(Data Table Module:
Contains a description of all variables used in the program.)

```
220 REM          INPUT VARIABLES
230 REM SHADOW=LENGTH OF SHADOW
240 REM ANGLE=ANGLE BETWEEN
250 REM BUILDING AND SHADOW
260 :
```

(Initialization Module:
Although in this program this is part of the Data Table, this is where variables are defined.)

```
270 REM          CONSTANT
280 PI = 3.14
290 :
300 REM          OUTPUT VARIABLE
310 REM HEIGHT=HEIGHT OF BLDG
320 :
```

(Main Program Module:
Main program begins here.)

```
330 REM +--------------------+
340 REM :  MAIN PROGRAM  :
350 REM +--------------------+
```

```
360 REM        GREETING SCREEN
370 TEXT
    :HOME
380 INVERSE
    :PRINT"       BUILDING HEIGHT ESTIMATOR"
    : NORMAL
    : PRINT
    : PRINT
390 :
400 REM        GET INPUTS
410 INPUT "SHADOW LENGTH: "; SHADOW
420 INPUT "ANGLE: "; ANGLE
430 :
440 REM        COMPUTE HEIGHT
450 RADIANS = (ANGLE * PI) / 180
460 HEIGHT = TAN(RADIANS) * SHADOW
470 :
480 REM        DISPLAY RESULTS
490 PRINT
    : PRINT "BUILDING HEIGHT IS: "; HEIGHT
```

Note the format of the REM statements. There are many ways to REMark a program, but the method shown here is one of the "safest" for two reasons:

-They are short (less than 33 characters wide) to insure a neat printout for users with a 40 column screen.

-There are no embedded control characters in the REM statements. There are some commercial Applesoft editors available that allow you to embed a CTRL-M or a CTRL-J (useful for going to the next line) or a CTRL-H (forces a backspace to cover up a line number). While this provides for a very impressive program listing on the screen or on a printer, it can really foul things up in a hurry if you try to capture the program in a text file or try to transmit the program over a telephone line to another computer with a modem.

Another item: The Renumber program on the DOS 3.3 System Master diskette was used here to keep the line numbers nice and neat after statements were added to the program. Instructions

on how to use the Renumber utility (which is designed so it can be in the computer at the same time as a program under development!) can be found by RUNning the program called, RENUMBER INSTRUCTIONS on the DOS 3.3 System Master diskette.

While RENUMBER INSTRUCTIONS contains a more detailed explanation than what will be presented here, nevertheless, here is a brief description of how to use it:

STEP 1: Make sure that you have saved the program you are working on and insert your DOS 3.3 System Master diskette.

STEP 2: RUN RENUMBER (yes, it's an Applesoft program) and press the RETURN key twice. Renumber will load in a machine code program that actually does all the work and "hide" it in the upper part of RAM, just under DOS.

STEP 3: LOAD the program you want to renumber and edit it as you normally would. Renumber will stay out of the way, allowing you to use your program nearly any way you like. When you want to renumber your program, you can just type in something as simple as: &F1000. The & (ampersand) is used to "activate" the machine code RENUMBER program, which then reads the F and the 1000 and interprets these characters to mean, "renumber the Applesoft program in memory, making the first line number 1000 and incrementing by 10 until the end of the program is reached." The renumbering takes place very quickly and smoothly; normal control of your Apple will then be returned to you.

Summary

The method of deciding how to restate (solve) a problem is called **algorithm development.** It can be summarized as follows:

The Five Steps of Algorithm Development

1. General Statement of the Problem
2. Data Table
3. Stepwise Refinement (Top Down Design)
4. Language Implementation
5. Debugging/Modification

A **General Statement** is the essence of what you are trying to accomplish in plain English.

A **Data Table** is a list of variables used in your program, organized according to type: input, output, constants, functions and temporaries. A Data Table serves as documentation for how your variables function and can also help you debug your program.

Stepwise Refinement is the continual breakdown of the General Statement into sentences that become very close to (approximate) Applesoft statements. The sentences are halfway between English and Applesoft and are known as **pseudo code.**

Language Implementation is the almost trivial task of translating pseudo code into actual Applesoft statements.

Debugging is the process of finding and removing errors in your program. **Antibugging** is writing your program in such a way that the errors never occur in the first place.

Modification refers to finding ways your original program can be improved.

Top Down Design means creating a program by beginning with a general statement of the problem and refining it into smaller, more manageable steps. It requires planning ahead and results in better programs. Bottoms up programming refers to creating a program by starting with a small procedure that works and expanding it into a more complicated program. It requires no forethought and generally results in **spaghetti programming.**

A **stub procedure** is a REMark statement that stands in place of a procedure that will be coded later. It allows the programmer to proceed with the rest of the program and can also be used in debugging to isolate faulty sections of code.

There are four main parts or modules of a program: **Identification, Data Table, Initialization** and **Main Program:**

The **Identification Module** contains the program's name, author and purpose.

The **Data Table Module** contains a list of variables used in the program.

Initialization Module refers to variables which must be defined or procedures which must be performed before the main program can work correctly.

The **Main Program** is the body of the program; it contains the essence of what you are trying to accomplish.

For Further Reading

Algorithms + Data Structures = Programs, Niklaus Wirth (Prentice-Hall, Englewood Cliffs, NJ, 1976). The classic, definitive statement on structured programming. Not for the beginning student, however. Just something to keep in mind when you are ready for it. May well "expand your programming horizons."

How To Write An Apple Program, Ed Faulk (DATAMOST, Inc. Chatsworth, CA, 1982). Good discussion on structured programming in Applesoft.

Introduction To Computers and Data Processing, Gary Shelly and Thomas Cashman (Anaheim Publishing, Fullerton, CA, 1980). Textbook with in-depth coverage of general structured programming techniques. Numerous full color illustrations and flowcharts.

Problem Solving and Structured Programming In BASIC, Elliot Koffman and Frank Friedman (Addison-Wesley, Reading, MA, 1979). Excellent, down-to-earth book on structured programming in a wide variety of BASIC capabilities. Techniques presented here are easy to apply to Applesoft.

CHAPTER 3

INTRODUCTION TO FLOW DIAGRAMS

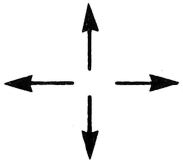
Sometimes, when working on large, complicated programs, you may find that even the best use of pseudo code and stepwise refinement still does not yield results that are understandable to you.

In such cases, it may be helpful to create a chart of what it is you're trying to accomplish. Such a chart is called a *flow diagram.*

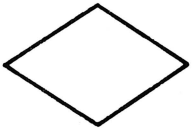
The Basics

The basic components of flow diagram figures are:

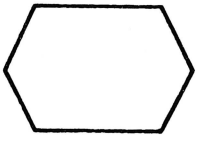
Node



Flow Arrows



Decision



Loop

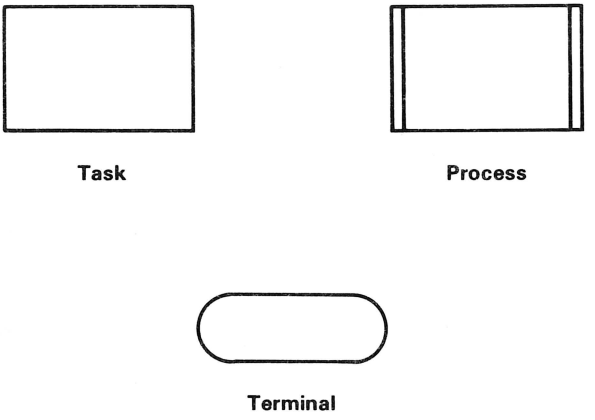


FIGURE 3.1

Here is an example with a couple of one-line sequential instructions:

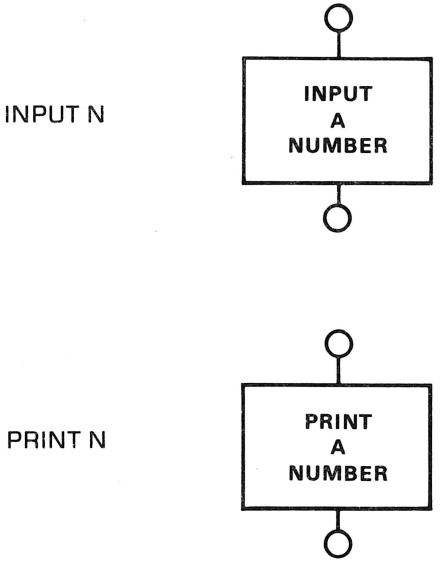


FIGURE 3.2

We can combine these structures at their common node points to form a program:

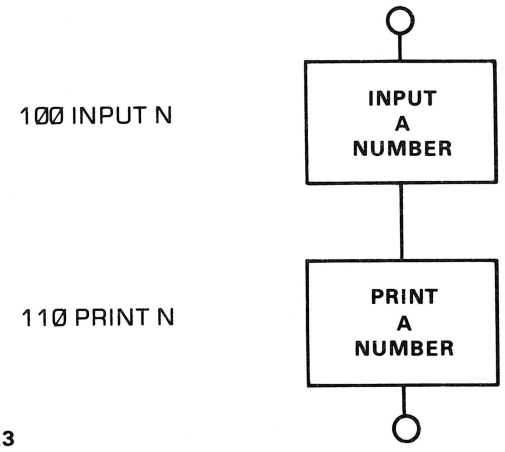


FIGURE 3.3

We can indicate the beginning and end of an actual program with "terminals":

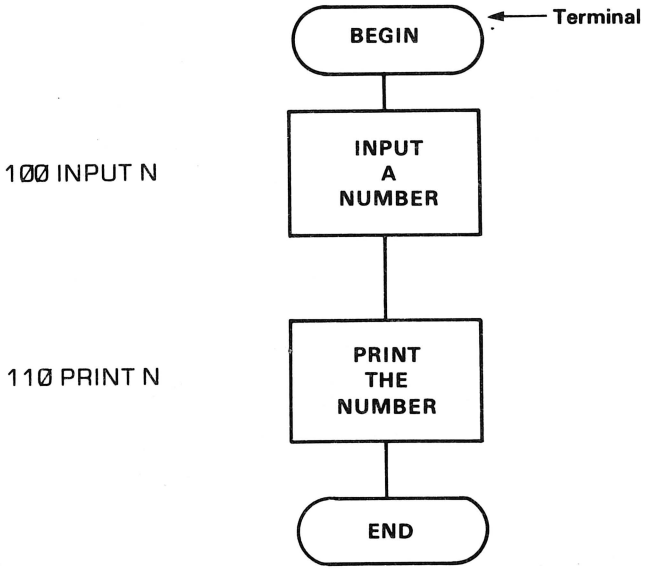


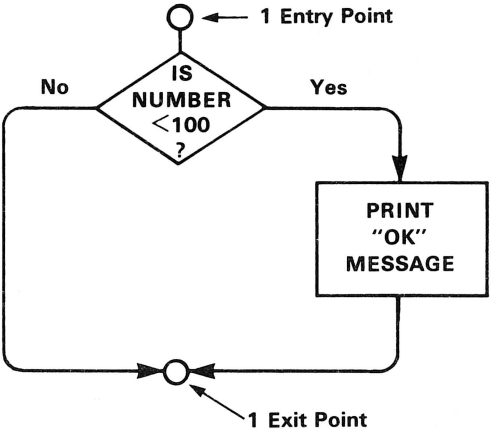
FIGURE 3.4

The Control Structures

Decisions are indicated by structures branching off to the side and meeting again at a common node:

IF THEN

```
100 IF N<100 THEN
      PRINT "OK"
```



IF THEN ELSE

```
100 ON N<100 GOTO 110
    : GOTO 130
110 PRINT "OK"
120 GOTO 150
130 REM ELSE
140 PRINT
      "NOT OK"
150 REM IF END
```

```
100 & IF N<100 THEN
110 PRINT "OK"
120 & ELSE
130 PRINT "NOT OK"
140 & IF END
```

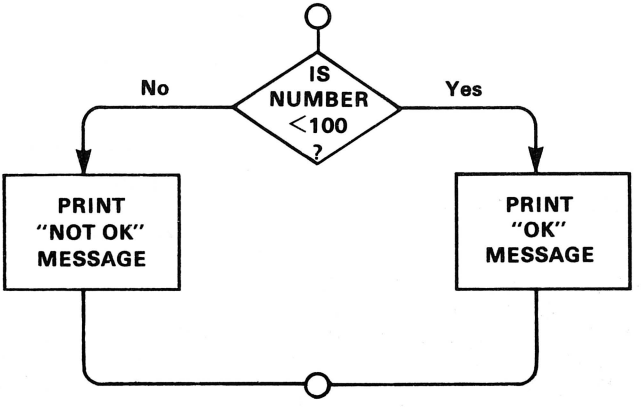
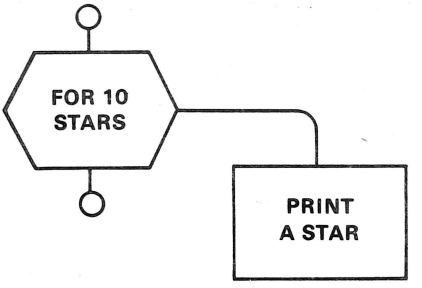
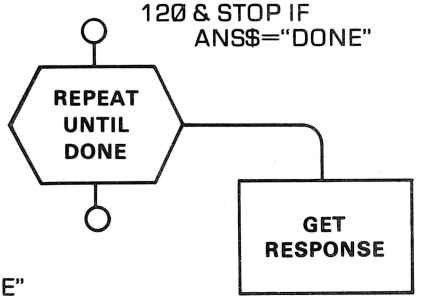


FIGURE 3.5



FOR NEXT

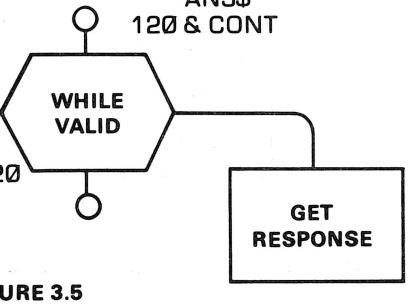
```
100 FOR I=1TO 10
110 PRINT "*";
120 NEXT
```



REPEAT UNTIL

```
100 REM REPEAT
110 INPUT ANS$
120 ON ANS$="DONE"
      GOTO 130: GOTO 100
130 REM UNTIL ANS$="DONE"
```

```
100 & RUN
110 INPUT ANS$
120 & STOP IF
      ANS$="DONE"
```



WHILE WEND

```
100 REM WHILE VALID
110 ON ANS$="YES" GOTO 120
    : GOTO 140
120 INPUT "VALID?";ANS$
130 GOTO 100
140 REM WEND
```

```
100 & ON ANS$="YES"
110 INPUT "VALID?";
      ANS$
120 & CONT
```

FIGURE 3.5

Nesting

The advantage of using flow diagrams to illustrate how a program works really stands out when nested control structures are used:

```
100 ON A>B GOTO 110: GOTO 130
110 IF C>D THEN
      PRINT "TEST 1"
120 GOTO 140
130 PRINT "TEST 2"
140 REM IF END
```

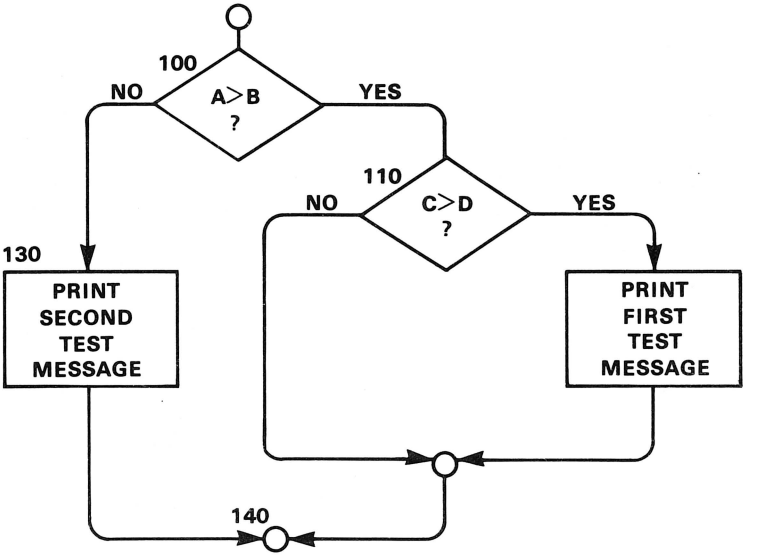


FIGURE 3.6

Flow Diagrams vs. Flowcharts

The previous figure shows two key features of flow diagrams: the control lines do not cross and the diagram "unfolds" into a spacious and almost symmetrical chart. With this system, it's easy to see why program reliability is assured: *there is simply no other path for the program to take.*

This is in contrast to flowcharts, which are holdovers from unstructured days:

Example of a Bad Flowchart

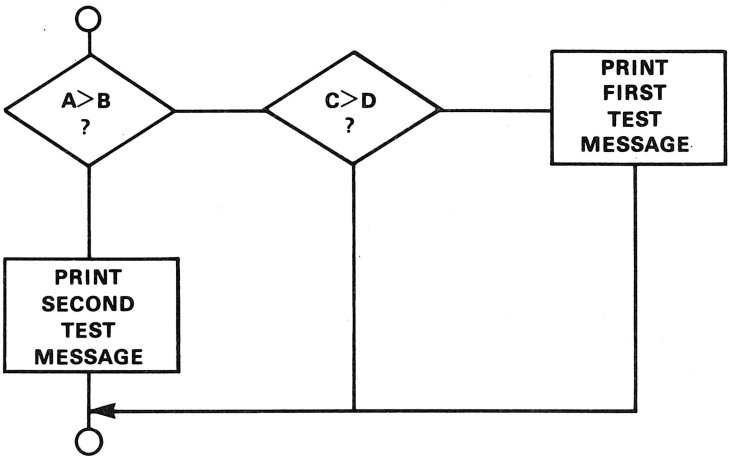


FIGURE 3.7

Flowcharts mindlessly follow just one rule: if true, branch to the right; otherwise branch down. While figure 3.7 does not appear to be totally unclear, it is not as clear as the structured version in figure 3.6. The truth be told, it is typical to find flowcharts with control lines crossing each other, helter-skelter and with control figures sprawled all over the page in one, big confusing mess.

Another Example of a Bad Flowchart

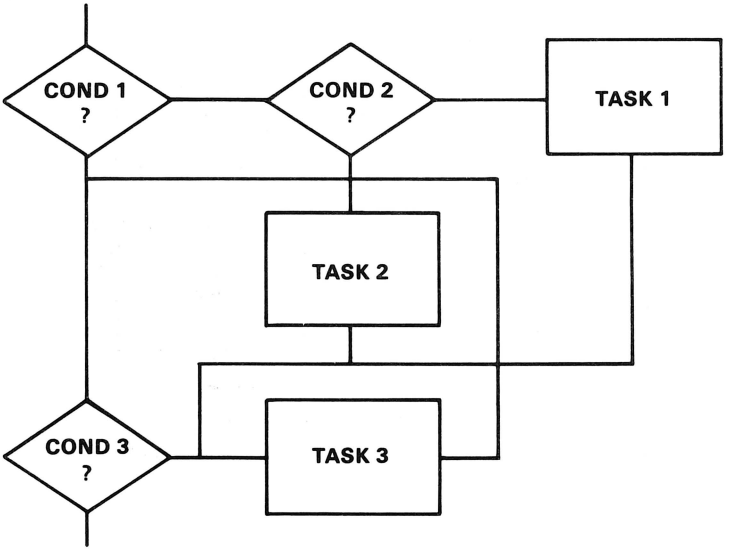


FIGURE 3.8

Refining Flow Diagrams

Flow diagrams can be refined just as pseudo code can. Here's a simple program that begins with the first level of refinement:

1. Enter a password
2. Leave a message
3. Say goodbye

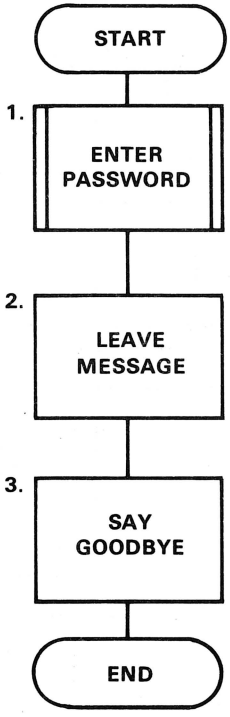


FIGURE 3.9

Note here that the "process" box at top which indicates a further level of refinement. We now can finish up with the subsequent level(s) of refinement by placing additional "expansion" diagrams off to the side of the main diagram:

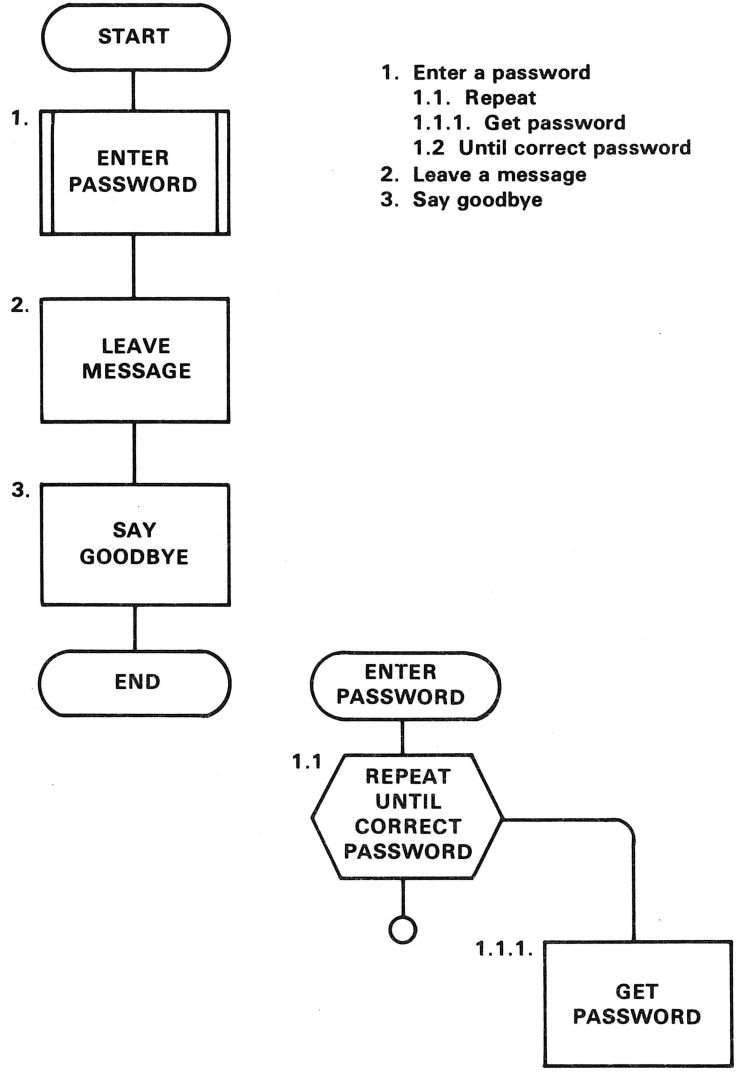


FIGURE 3.10

Actual Example

Here is a sample pseudo code listing and flow diagram for a program that reads all the file names off a disk and stores them in an array. (The actual program is listed in the next chapter on sample algorithms.)

LOAD ALL FILE NAMES

Pseudo Code

1. Display greeting screen.
2. Load all file names from directory into array.
   2.1. Repeat (all steps beneath 2.1).
   2.1.1. Read a directory sector into the buffer.
   2.1.1.1. Call the directory read routine indexed by whether or not this is the first time the routine has been called.
   2.1.2. Initialize Beginning and End of Name Pointers.
   2.1.3. While not end of directory and Beginning of Name Pointer <= end of buffer.
   2.1.3.1. Increment file name counter.
   2.1.3.2. While End-of Name Pointer points to a blank
   2.1.3.2.1 Decrement End-of-Name pointer.
   2.1.3.3. For each character in the current file name.
   2.1.3.3.1. Make sure the high bit is off.
   2.1.3.3.2. Add the character to the name being constructed in the array.
   2.1.3.4. Move Beginning and End-of-Name Pointers to next name
   2.2 Until end of directory.
3. Clear error condition caused by RWTS.
4. Print the file names loaded.

LOAD ALL FILE NAMES

Flow Diagram

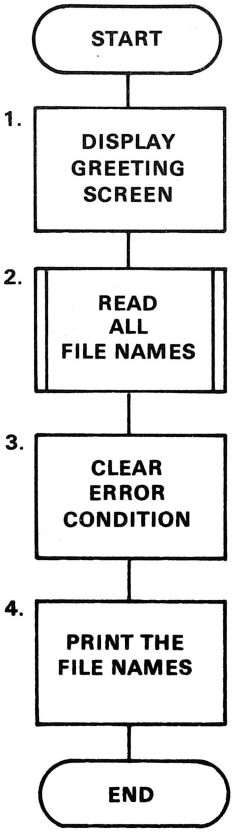


FIGURE 3-11 **(continued)**

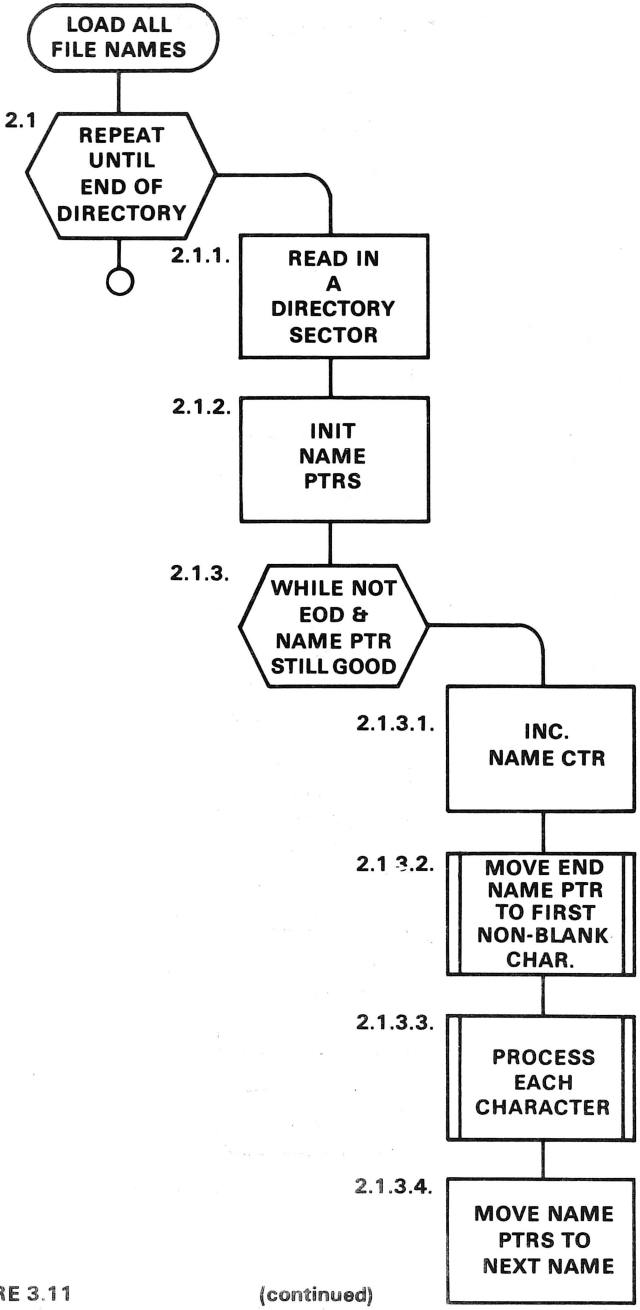


FIGURE 3.11 (continued)

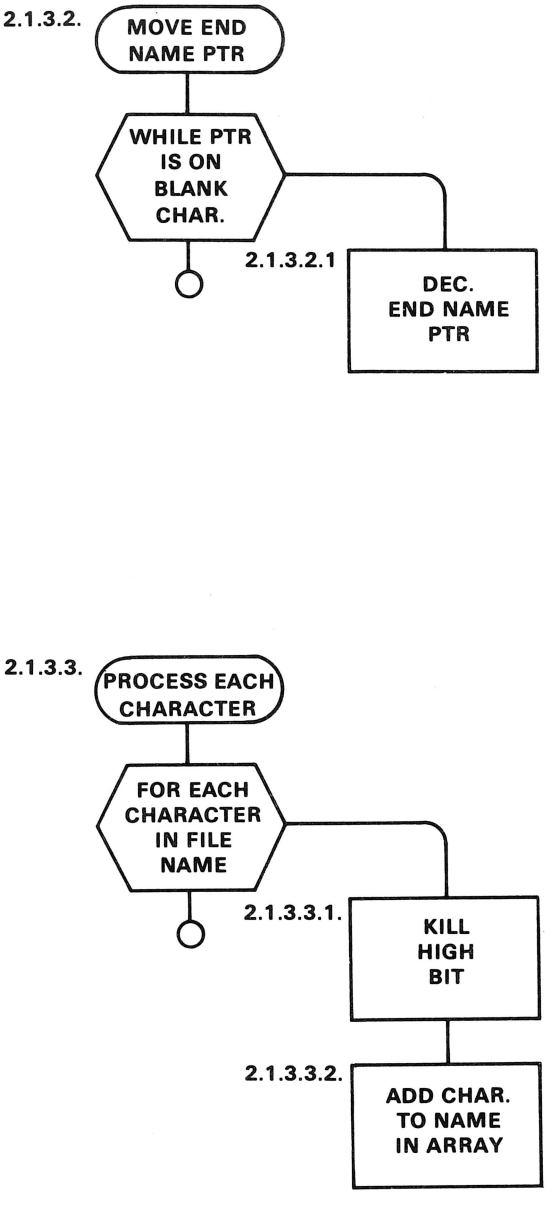

2.1.3.2.
MOVE END NAME PTR
WHILE PTR IS ON BLANK CHAR.
2.1.3.2.1
DEC. END NAME PTR
2.1.3.3.
PROCESS EACH CHARACTER
FOR EACH CHARACTER IN FILE NAME
2.1.3.3.1.
KILL HIGH BIT
2.1.3.3.2.
ADD CHAR. TO NAME IN ARRAY

FIGURE 3-11

Summary

A **flow diagram** is a graphical representation of the logic flow of a program. It is a simplified version of **pseudo code** and can be used when the pseudo code of a large, complicated program is not clear.

The figures used in flow diagrams represent nodes, flow lines, tasks, processes, decisions, loops and terminals.

Flow diagrams can be **nested** and **refined** just like pseudo code. Unlike **flowcharts**, flow diagrams unfold into spacious, almost symmetrical diagrams with control lines that do not cross. A properly drawn flow diagram is proof of the program's reliability.

For Further Reading

Introduction To Computers and Data Processing, Gary Shelly and Thomas Cashman (Anaheim Publishing, Fullerton, CA, 1980). Textbook with in-depth coverage of general structured programming techniques. Numerous full color illustrations and flowcharts.

Problem Solving and Structured Programming In BASIC, Elliot Koffman and Frank Friedman (Addison-Wesley, Reading, MA, 1979). Excellent, down-to-earth book on structured programming in a wide variety of BASIC capabilities. Techniques presented here are easy to apply to Applesoft.

CHAPTER 4

USEFUL ALGORITHMS

This chapter contains some practical routines and subroutines for use in your own programs. These examples are used to illustrate the use of structured programming and are solutions to commonly occurring problems. Such solutions are called *algorithms.*

Sort Algorithms

Three commonly used sort algorithms are presented here: bubble, select and Shell. It must be remembered that these are "bare bones" subroutines. The name of the actual array (if it is not A$()) and the size of the array (ARRAYSIZE in these examples) must be specified.

In addition, since speed is a critical factor in sorting, certain liberties were taken with Applesoft to avoid GOTOs wherever possible: a special form of the FOR NEXT loop was used--

```
FOR I = Ø TO 1

(stmts)

I = (exit condition): ← Note use of Boolean expression
NEXT I
```

This type of looping structure adjusts the loop index variable I to keep the loop repeating until the exit condition is true. This then, is a REPEAT UNTIL loop in disguise. The use of the FOR NEXT loop in this manner is not generally allowed from a structured standpoint because it is too easy to fall into "spaghetti" programming.

Bubble Sort

The bubble sort is the easiest sort for beginning programmers to design. It is also the slowest of all the sort algorithms. It works by starting at the top of the array and comparing subsequent pairs of elements, switching out of order pairs as it works it way down to the bottom of the array. It repeats this process until it can go all the way to the bottom of the array without switching any pairs (i.e., the array is properly sorted). The slowness of this method is due to the fact that many comparisons and switches are needed for every pass through the array. Note the use of the SRTED (sorted) flag to keep the logic structured:

```
1Ø REM BUBBLE SORT
2Ø S1$="CAT" : S2$="BAT" : S3$="DOG"
3Ø GOSUB 1ØØ
4Ø END
```

```
100 REM REPEAT

110 FOR SRTED = 0 TO 1
  : FOR I = 1 TO ARRAYSIZE - 1
  :  SRTED = 1
  :   IF A$(I+1) < A$(I) THEN
        SRTED = 0
  :     TEMP$ = A$(I)
  :     A$(I) = A$(I+1)
  :     A$(I+1) = TEMP$
120   NEXT
  : NEXT
  : RETURN

  : REM UNTIL SORTED
```

Results:

```
BAT
CAT
DOG
```

Select Sort

The select sort works by scanning the entire array and selecting the least element. The least element is then swapped with the top element and the size of the array is reduced from the top element. The process is repeated, *with the array growing shorter until the array is in order.*

A common use of this method is to sort a hand of freshly dealt cards into order. Assuming that a 9, 7, 2, 4 and an 8 have been dealt, the hand is scanned for the least card, which in this case is the 2. The 2 is moved to the left side of the hand and the four remaining cards, 9, 7, 4 and 8, are scanned again. The 4 card is moved to the left side and the remaining three cards are scanned, etc:

```
          9  7  2  4  8
             ▲ The hand is scanned for the least card

       2     9  7  4  8
       ▲-----------------The least card is placed at the beginning

2              9  7  4  8
                  ▲----------------------Scanning begins again
2  4           9  7  8                 and the process is repeated
2  4  7        9  8                    until the cards are in order
2  4  7  8     9
2  4  7  8  9
```

The select sort is faster than the bubble sort but is slower than others. Its beauty lies in its simplicity: it consists primarily of just two FOR NEXT loops:

```
10 REM SELECT SORT
20 S1$="CAT" : S2$="BAT" : S3$="DOG"
30 GOSUB 100
40 END


100 FOR I = 1 TO ARRAYSIZE -1
   : FOR J = I + 1 TO ARRAYSIZE
   :  IF A$(J) < A$(I) THEN
        TEMP$ = A$(I)
   :    A$(I) = A$(J)
   :    A$(J) = TEMP$
120  NEXT
   : NEXT
   : RETURN
```

Results:

```
BAT
CAT
DOG
```

Shell Sort

The Shell sort of algorithm was invented by a mathematician named Donald Shell. The Shell algorithm works to avoid "public enemy #1" of all sort algorithms: wasting time on comparisons and swapping. Unlike the bubble and the select sorts, where a dramatic increase in time is needed to sort large arrays, the Shell sort is one of fastest, all-around sorts available. (Because of the complexity of code involved, however, its advantages are not seen until an array of intermediate size (approximately 100 elements) is sorted.)

Unfortunately, the explanation of how it works is not simple and only a brief description can be given here. The Shell sort works by determining a "stride" or a distance between elements. The elements at the beginning and end of the stride are compared and swapped if out of order. The stride is moved down to the next pair and the process is repeated until the program "walks through" the array. The stride length is reduced and the program walks through the array again. This process is repeated until the stride length is less than one element. Its speed results from elements being selectively swapped over large distances. Interestingly enough, the speed of this sort can be "fine tuned" by the stride determining statements at the beginning of the program. It is curious to note that no one yet has discovered the formula for determining stride lengths that deliver the fastest results!

```
10 REM SHELL SORT
20 S1$="CAT" : S2$="BAT" : S3$="DOG"
30 GOSUB 100
40 END


100 STRIDEBND = 1
110 STRIDEBND = 3 * STRIDEBND + 1
    : IF STRIDEBND < ARRAYSIZE THEN 110


120 REM WHILE STRIDEBND >= 1
```

```
130 FOR I = 0 TO 1
    : STRIDEBND = (STRIDEBND - 1) / 3
    : IF STRIDEBND < 1 THEN 170
140 FOR BTM = STRIDEBND + 1 TO ARRAYSIZE
     LNTH = BTM - STRIDEBND
    : TEMPBTM$ = A$(BTM)
150 IF A$(LNTH) > TEMPBTM$ THEN
     A$(LNTH + STRIDEBND) = A$(LNTH)
    :  LNTH = LNTH - STRIDEBND
    :  IF LNTH > 0 THEN 150
160  A$(LNTH + STRIDEBND) = TEMPBTM$
    : NEXT
    : I = 0
170 NEXT
    : RETURN

    : REM WEND (WHILE END)
```

Results:

```
BAT
CAT
DOG
```

DOS Algorithms

Two useful DOS algorithms are presented here: how to load the directory into an array and how to read or write any track or sector.

Load Directory Into An Array

The directory of a normal DOS 3.3 diskette can contain up to 105 file names. A CATALOG command will display the names, but you will be unable to "capture" the information and manipulate it into other useful formats. The algorithm presented here reads all of the file names from the directory (including deleted files!) and loads them into the array of your own choice. You can write your own program to take advantage of this and do some customization such as fancy double column catalogs, printing the names in condensed type on a mailing label for your disk, etc., etc.

```
10 REM LOAD DIRECTORY INTO AN ARRAY
   15 DIM FILNAME$(105)

20 REM CONSTANTS

30 RDDIRSEC = 45073 : REM READ DIRECTORY
   SECTOR ROUTINE
40 B1STNAME = 46281 : REM BEG OF 1ST FILE
   NAME IN BUFFER
50 EBUFFER = 46522 : REM END OF FILE NAME
   BUFFER
60 REM PROGRAM VARIABLES

70 REM NAMENO : REM FILE NAME NUMBER
80 REM BNAMEPTR, ENAMEPTR : NAME
   POINTER BEG & END

100 REM REPEAT

110 CALL RDDIRSEC + 15 * (NAMENO > 0)
    : BNAMEPTR = B1STNAME

120 REM WHILE NOT END OF DIR &
130 REM BEG OF NAME PTR <= END OF BUFR

140 ON (PEEK(BNAMEPTR-3) < > 0)
      AND (BNAMEPTR <= EBUFFER)
      GOTO 150
    : GOTO 210 : REM <-- ELSE
150 NAMENO = NAMENO + 1
    : ENAMEPTR = BNAMEPTR + 29

160 REM WHILE END OF NAME PTR = BLANK
170  IF PEEK(ENAMEPTR) = 160 THEN
       ENAMEPTR = ENAMEPTR - 1
    :    GOTO 170

180 FOR CHAR = BNAMEPTR TO ENAMEPTR
    : BYTE = PEEK(CHAR)
    : IF BYTE > 127 THEN BYTE = BYTE - 128
190  FILNAME$(NAMENO) =
       FILNAME$(NAMENO) + CHR$(BYTE)
    : NEXT
```

```
200 BNAMEPTR = BNAMEPTR + 35
    : GOTO 120

210 ON (PEEK(BNAMEPTR-3) = 0) GOTO 220
    : GOTO 100

220 REM UNTIL END OF DIRECTORY
```

Results:

```
FOR I=1 TO NAMENO: PRINT FILNAME$(I): NEXT

HELLO
FID
PROG 1
PROG 2
```

RWTS

RWTS are the initials chosen by Apple Computer Inc. to stand for "Read or Write a Track and Sector." As the name implies, RWTS routines allow you to directly read a 256-byte sector from the disk or write a sector to the disk. Uses for this include changing the name of your disk, changing the name of your HELLO program, making permanent patches to DOS, hiding secret codes, examining sectors, and similar DOS operations.

While the basic principles of RWTS are outlined on pp. 94 - 98 of *The DOS Manual*, the reader of that publication is warned in the very first paragraph, "You may skip this section if you're not familiar with machine language."

Actually, things don't have to be that difficult. You can do the same thing from Applesoft with a minimum of muss and fuss. (See *A Simplified Approach to RWTS* by the author in *CALL - A.P.P.L.E. In Depth: All About DOS*, for further details and a useful utility that uses RWTS.)

The following program performs RWTS for you — all you have to do is tell it which track and sector and whether you want to read from the disk or write to the disk and it will do all the work. Note that the routine uses a fixed buffer provided by DOS at 46267 to 46522. To read the buffer after you load a disk sector into it, just PEEK at that range. To change the buffer before writing it out to the disk, just POKE the new values into that range also:

```
10 REM RWTS

100 INPUT "ENTER TRACK, SECTOR:"; TK, SC
    : INPUT "READ (1) OR WRITE (2):"; RWCODE
    : POKE 45121, RWCODE
    : POKE 45975, TK
    : POKE 45976, SC
    : CALL 45111 <-- RWTS routine
    : POKE 45121, 2 <-- Restore write code
    : POKE 72,0  <-- Clear error condition
```

EXEC Files

EXEC files are text files of valid Applesoft or DOS commands that can be executed with the EXEC command. Their primary advantage is that they are executed as they are read from the disk; they do not alter any program or pointer in memory.

One area where an EXEC file comes in handy is with binary files. Often the name of a binary file does not indicate where the file loads into memory or its length. The beginning address is vital to ensure that files don't load on top of one another; the length is needed during a BSAVE.

The EXEC file presented below tells you the starting address and length of the last BLOADed or BRUNned binary file. The program MAKE FIND LAST BFILE creates EXEC file FIND LAST BFILE, which in turn displays in decimal and hexadecimal the starting address and length of the last binary file BLOADED or BRUN. (A simpler version of this program appears in Bill Sander's book, *The Elementary Apple*. The version below is more powerful and shows off some tricks you can do with monitor routines.)

```
10 REM MAKE FIND LAST BFILE
100 LET D$ = CHR$ (13) + CHR$ (4)
  :OP$ = D$ + "OPEN"
  :DE$ = D$ + "DELETE"
  :WR$ = D$ + "WRITE"
  :CL$ = D$ + "CLOSE"
  :N$ = "FIND LAST BFILE"
  :Q$ = CHR$ (34): REM QUOTE
110 PRINT OP$N$;DE$N$;OP$N$;WR$N$
120 PRINT
    "?:?"
    Q$"LAST BINARY FILE AT:"Q$
  : PRINT
    "CALL -998
    :CALL -998
    :?"
    Q$"   A"Q$
"PEEK(43634)+256*PEEK(43635)"
    Q$", L"Q$
    "PEEK(43616)+256*PEEK(43617)"
130 PRINT
    "POKE 69,PEEK(43635)
    :POKE 70,PEEK(43634)
    :POKE 58,65
    :POKE 59,249
    :POKE 49,0
    :?"
    Q$"   A$"Q$
    ";:CALL -327
    :POKE 69,PEEK(43617)
    :POKE 70,PEEK(43616)
    :POKE 58,65
    :POKE 59,249
    :POKE 49,0
    :?"
    Q$", L$"Q$
    ";:CALL -327
    :CALL -998
    :CALL -998"
140 PRINT CL$N$
```

Summary

A solution to a problem is called an **algorithm**. Three types of sample algorithms for commonly occurring programming problems on the Apple were presented in this chapter: Sort, DOS and EXEC algorithms.

Three types of sort algorithms were presented: **bubble, select** and **Shell**. The bubble is the slowest sort, but the one that is usually thought of by beginning programmers. The select sort is simpler and faster than the bubble sort and works well with short to intermediate length arrays. The Shell sort is complicated, but is one of the fastest sorts for intermediate to lengthy arrays.

Two types of DOS algorithms were presented: how to load the directory into an array and how to read or write any track or sector (RWTS). Loading the directory into an array allows the programmer to do fancy catalog displays, disk labels, etc. The RWTS routine presented here provides for a simple way of reading or writing a disk sector without the bother of machine code interfacing.

The example used to illustrate how to use EXEC files from a structured standpoint is a utility that finds the starting address and length of the last BLOADed or BRUNned binary file. This information is displayed in both decimal and hexadecimal form and requires no knowledge of DOS file storage areas.

For Further Reading

Algorithms + Data Structures = Programs, Niklaus Wirth (Prentice-Hall, Englewood Cliffs, NJ, 1976). The classic, definitve statement on structured programming. Not for the beginning student, however. Just something to keep in mind when you are ready for it. May well "expand your programming horizons."

CHAPTER 5

TEXT FILES

Purpose of Text Files

Text files provide a convenient means of storing large amounts of data on a disk for later retrieval and manipulation. If you centralize data in a text file, you can write any number of other programs to access the file. This concept is known as *database design* and provides for an efficient means of giving different programs access to a large body of data.

Another use of text files unique to the Apple is that if valid Applesoft, DOS, or monitor commands are saved in a text file, the EXEC <name of text file> command will execute the commands, just as if you had typed them in on the keyboard yourself. Further information on this can be found in *The Apple II DOS Manual* and in *The Elementary Apple.*

Structure of Text Files and the Disk

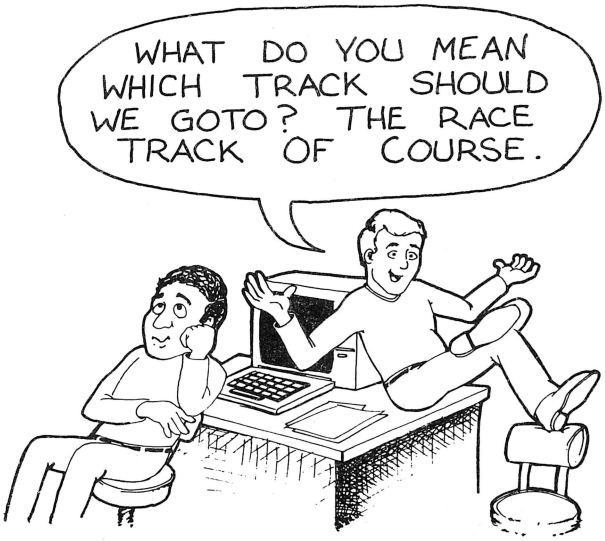
Despite the advent of newer and more powerful operating systems, Apple's DOS 3.3 still has some powerful features: it can typically hold more files on a disk and it uses less disk space for smaller files.

A standard Apple disk is divided into 35 concentric "rings" called tracks. Each track is divided into 16 wedge-shaped "sectors," giving a total of 16 * 35 = 560 sectors on a disk. Each sector holds 256 bytes or characters, giving the maximum number of bytes on a disk 256 * 560 = 143,360. It is more convenient to divide this number by 1,024 (a "K" or Kilobyte) and express it as 143,360 / 1,024 = 140K.

DOS resides on the outer three tracks of the disk and the directory resides on the middle track (track 17 / $11 hex). This results in the subtraction of four tracks for a total usable file space of

560 – 4*16 = 496 sectors or (496 * 256) / 1,024 = 124K bytes. (In case you're curious, the directory is put in the middle of the disk to reduce access time for the drive's head to reach it.)

The first sector of track 17 is reserved for something called "VTOC" (Volume Table of Contents). VTOC is used to keep track of which sectors are free, or available for file storage. This scheme serves to protect files that you've saved to disk: their storage sectors are marked in VTOC as "in use." Utility programs that compute the amount of free space on the disk usually examine VTOC for the basis of their calculations.



The remaining 15 sectors in track 17 are reserved for the directory. Each directory sector can contain up to seven file names, making a total of 7 * 15 = 105 file names possible in your directory. Thus, it is possible to have several screenfuls of text file names displayed with the CATALOG command!

If you were to use one of the commercially available "Disk Zap" programs, you would see that a text file is composed of hexadecimal numbers. The hexadecimal system is "base 16" which means that it has 16 digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F to use in creating numbers as opposed to our "base 10" number system, which only has 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. If you converted

these hexadecimal numbers into decimal, you would see that they correspond to something called an "ASCII Chart." You can see an example of an ASCII Chart on pg. 138 of the *Applesoft BASIC Programming Reference Manual.*

However, there is a fly in the ointment; *all* of the characters in an Apple text file have 128 added to their ASCII value! This is peculiar to the Apple and is done to help it distinguish ASCII characters from non-ASCII characters. An ASCII character with 128 added to it is said to be in "Apple ASCII," "high ASCII" or "negative ASCII," while regular or true ASCII is known as "low ASCII" or "positive ASCII." Here are some examples:

ASCII Character	Decimal Lo ASCII/ Hi ASCII		Hex Lo ASCII/ Hi ASCII	
	Upper case letters			
A	65	193	$41	$C1
B	66	194	$42	$C2
C	67	195	$43	$C3
	Lower case letters			
a	97	225	$61	$E1
b	98	226	$62	$E2
c	99	227	$63	$E3
	Digits			
0	48	176	$30	$B0
1	49	177	$31	$B1
2	50	178	$32	$B2
	Punctuation			
,	44	172	$2C	$AC
–	45	173	$2D	$AD
.	46	174	$2E	$AE
	Control characters			
<NULL>	0	128	$00	$80
<CTRL C>	3	131	$03	$83
<CTRL D>	4	132	$04	$84
<CR>	13	141	$0D	$8D

Notice from the above that the Apple ASCII code for the *character* "0" is "B0" and not "0". If your disk zap program shows you a "00" in the file, you are looking at the ASCII <NULL> character and not zero. NULLs are used in text files to tell the Apple when it has reached the end of the file or a record.

You should also note that while it is possible to POKE a number such as 255 (which consists of three digits) into a single byte in RAM, it will be written out to the disk file as three bytes: "2", "5" and "5". Applesoft is capable of reducing a three-digit number such as "255" into what is known as a "binary image" and putting it into a single byte. DOS, however, does not have this capability. Thus all Apple text files are "pure" ASCII; there are no binary representations of numbers in them.

Basic File Structure: Records and Fields

Files are usually composed of two structures: *records* and *fields*. A file is composed of many records:

Text File Structure

Record 1:	
Record 2:	
Record 3:	

Records, in turn, are composed of fields:

Text File Structure

Record 1:	Field 1	Field 2	Field 3
Record 2:	Field 1	Field 2	Field 3
Record 3:	Field 1	Field 2	Field 3

Though not shown here, for simplicity's sake a <CR> (141 or $8D) is at the end of every record and a < *NULL* > (0) is at the end of every text file to help DOS find its way around the file.

For example, if we wanted to construct a file of names, addresses and phone numbers for a telephone directory, a typical file structure might look something like this:

SAMPLE TELEPHONE DIRECTORY FILE STRUCTURE

Record 1	JOHN	SMITH	802 BRICK ST.	GLENDALE	CA	90023	714	202-9123
Record 2	BILL	REYNOLDS	90E. 43RD ST.	NEW YORK	NY	10123	217	952-0721
Record 3	NICOLE	TURNER	601 WING ST.	SAN DIEGO	CA	92115	619	562-4213
column no. for beginning of field	1	11	26	46	66	68	73	76
	FIRST NAME FIELD (10)	LAST NAME FIELD (15)	STREET ADDRESS FIELD (20)	CITY FIELD (20)	STATE FIELD (2)	ZIP CODE FIELD (5)	AREA CODE FIELD (3)	PHONE NUMBER FIELD (8)

FIGURE 5.1

Sequential and Random Access Files

A problem pops up immediately here, though: what do you do with the extra space left over when information does not fill up the field completely?

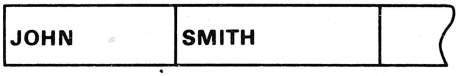


FIGURE 5.2

There are two techniques for dealing with this situation. The first avoids the wasted space by separating the fields with a single character called a "delimiter." DOS uses a comma as a field delimiter, but with the use of the LEFT$, RIGHT$ and MID$ string manipulation functions, a delimiter can be any character you want. It is usually best to pick a character which would not be confused with a valid data character in the rest of the record. Under this technique, records have different lengths. This is called *sequential file organization.*

Here is an example that uses the "@" character as a delimiter:

SEQUENTIAL FILE ORGANIZATION

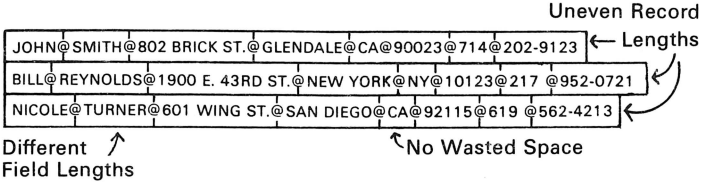


FIGURE 5-3

Note that it is usually simpler, faster and easier to write each field out as a record by itself (i.e., one field per record). This makes the <CR> serve as both an "end of field" delimiter and an "end of record" delimiter.

The second technique uses the blank space to keep the fields lined up evenly from record to record. Under this technique, all records have the same length. This is called *random access file organization.*

RANDOM ACCESS FILE ORGANIZATION

Same Record Lengths

JOHN	SMITH	802 BRICK ST.	GLENDALE	CA	90023	714	202-9123
BILL	REYNOLDS	1900 E. 43RD ST.	NEW YORK	NY	10123	217	952-0721
NICOLE	TURNER	601 WING ST.	SAN DIEGO	CA	92115	619	562-4213

Same Field Lengths

Padded With Blanks

FIGURE 5-4

Apple DOS supports both types of file organization and has specific commands for each one. However, this is not to say, that there are only these two types of file structures. There are many other structures, such as ISAM (Indexed Sequential Access Method), binary trees and B-trees, but they are more complicated and are not directly supported by existing DOS commands.

Sequential files save space and can be read or written relatively quickly, but they are hard to modify. Random access files waste space and take longer to be read or written but they are easier to modify. You must therefore plan ahead when considering which file structure to use for your specific application.

Some sample uses for sequential files would include files that are designed to be used as is and not modified that often:

- EXEC files
- Final drafts of documents
- Business forms, receipts, etc.

Some sample uses for random access files would include files that need to be periodically updated or modified:

- Telephone directory
- Inventory
- Billing records

Text File Memory Requirements

It is helpful to know during the planning stages of your program design how much memory and disk space a text file will occupy. The following chart provides a basis for this planning. The chart shows how much space a file containing 100-byte records will occupy. Memory usage is directly proportional to the figures presented here, so if your file uses 50-byte records instead, then it will use half as much memory.

TEXT FILE STORAGE REQUIREMENTS

Recs	Bytes	% RAM	Sectors	% Disk
350	35,000	96%	138	28%
300	30,000	83%	119	24%
250	25,000	69%	99	20%
200	20,000	55%	80	16%
150	15,000	41%	60	12%
100	10,000	28%	41	8%
50	5,000	14%	21	4%
25	2,500	7%	11	2%
10	1,000	3%	5	1%

Key to Column Headings:

Recs: Number of 100-byte records in the file. Records actually contain 99 bytes, but the <CR> at the end of the record counts as byte 100.

Bytes: Total number of bytes in the file.

% RAM: Amount of RAM used by file. Based on a 48K system with DOS booted. Does not include space required for the program itself.

Sectors: No. of disk sectors occupied by file. Includes the track/sector list sector.

% Disk: Amount of disk used by the file. Based on 496 available sectors.

Disk sectors required for random access files:

Sectors = INT(1.999 + ((Recs * Rec length) /256))

Disk sectors required for sequential files:

Sectors = INT(1.999 + (Bytes / 256))

File Design Considerations

It is very easy to change records in a random access file. Because they are all the same length, you can find and replace records by their numbers (position in the file). Sequential files require a re-write of the entire file into a temporary file. The original file is deleted and the temporary is renamed as the original file.

Conversely, it is easier to append records to a sequential file than it is to a random access file. Sequential files can use the APPEND command, random access files cannot.

Beware of commas. Applesoft's INPUT command is used to read information from a disk file. If there is a comma in the incoming data, Applesoft interprets the comma as a delimiter and will throw away the rest of your record. This is why you sometimes get the EXTRA IGNORED error message when reading files. If you must use commas, use the program listed in Chapter 10 to do your INPUTting: it will read text characters as is . . . commas and all.

Use an array to hold the file. There are two troublesome DOS commands, POSITION and B)yte that are used to access a file at specific locations. It is far easier and faster to simply load the entire file into an array and access the records directly through the use of array element numbers. If a file is too large to fit into an array, simply break it into smaller files.

Beware of trying to read or write a random access file as if it were a sequential file. A random access file sometimes contains zeros (nulls) which are interpreted as end of file characters and which prematurely stop file access.

Beware of ONERR. The ONERR command is useful with "error trapping," where mistakes are dealt with under program control instead of a system crash. The problem with ONERR, however, is that it is a "loop killer." This means that if you use ONERR within a GOSUB *or a* FOR NEXT loop, the RETURN and the NEXT statements at the end of these routines will not work. ONERR was not integrated very well with DOS and destroys a region of memory in the Apple known as the stack where the

return addresses for active GOSUBs and FOR NEXT loops are kept. To return from an ONERR return you must use a GOTO statement. Note also that anything after an ONERR statement is ignored:

```
100 ONERR GOTO 200 : GOTO 120
110 GOTO 200
120 PRINT "THIS STATEMENT IS NEVER
    EXECUTED"
130 :
200 END
```

Beware of defining D$=CHR$(13)+CHR$(4). To execute DOS commands within an Applesoft program, they must be PRINTed with a preceding <CTRL D>; e.g.: 100 PRINT CHR$(4) "CATALOG". It has become common to define the CHR$(4) as D$ and to use the following instead:

```
100 D$ = CHR$(4) : REM CTRL D
110 PRINT D$"CATALOG"
```

Some programmers have discovered that DOS commands were apparently intended to be one-statement-per-line commands. Thus:

```
110 PRINT D$"CATALOG"
120 PRINT D$"CATALOG"
```

will give you two CATALOGs in succession just fine, but trying to do the same thing with an Applesoft shortcut does not work:

```
110 PRINT D$"CATALOG"; D$"CATALOG"
```

Here, you will get the first catalog, but not the second. What some programmers have done to get around this limitation is to define D$ as a <CR> and a <CTRL D> with D$ = CHR$(13) + CHR$(4). When D$ is printed, the Apple sees the <CR> and is fooled into thinking that the following command is on a separate line. The problem with this trick is that any PRINT D$ command writes the <CR> into any file you have open and write enabled. For example, if you have two files OPEN so that you can read a record from one and write it to the other, when you try to read a record, the D$ will send a carriage return to the write enabled file.

There are some solutions: stick with one DOS command per line, use complete PRINT statements for each DOS command when combining them on one line, or use two forms of the D$ as needed: D$=CHR$(13)+CHR$(4) and D4$=CHR$(4).

Useful File Handling Techniques

Sample routines are included here, demonstrating how to make, read, append and "merge sort" both sequential and random access files. Each routine is designed to run by itself "as-is" as a demonstration program, but with appropriate attention to the variables used, each routine can be used as a subroutine within your main program.

Special attention should be paid to the concept of "merge sorting." There is precious little information avaliable on this topic relating to personal computers, which deals with the crucial question, "How do you sort a disk file that is too large to fit into memory?"

"That's easy," you say. "All you have to do is break up the large file and sort the subfiles." Then what? You'd wind up with several files that range from A to Z; append them together and you get a big file that is still out of order, because they are not sorted across the subfiles.

Actually, you almost solved the problem. You read in part of the original file into memory, sort the records there and write them out to disk as a subfile. You read in the next part of the original file and repeat the process until you have copied the original file in the form of several sorted subfiles. This process is known as *EXTERNAL SORTING*, because you are sorting records external to your computer's memory by bringing them into the high speed memory (RAM) and sorting them there.

The final step is to merge each of the subfiles together. This is done by taking two subfiles and reading a record from each. The records are compared and the smaller (lesser) is written to a merge file. Records are read from the subfiles, compared and written to the merge file until the subfiles are used up. Since

records in the subfiles are in ascending order, records in the merge file are also in ascending order. Files can be merged together until it reaches the size of the original file. A copy of the original file has now been made, except that it is in order.

The combination of external sorting and file merging is known as *MERGE SORTING*, and is more time and space efficient than alternate methods of using numerous smaller files to hold records. Routines on external sorting and file merging have been included here to help you with your file sorting problems.

Each of the following routines is preceded with a description of how the program works as well as tips and pitfalls. Different routines for sequential and random access files are presented, allowing you to use the file structure best suited for your purposes.

Make a Sequential File

A file can be created without the danger of partially overwriting a previously existing file by the same name by OPENing it (to create it if it doesn't exist), DELETEing it and reOPENing it again (for output). See lines 180 - 200. Once WRITE mode has been turned on, it remains on and records can be written to the file with an input loop (see lines 220 - 320). A WRITE mode will have to be turned off if any screen prompts are to be displayed (see lines 310 - 220).

```
100 REM ****************************
110 REM *      MAKE SEQ FILE       *
120 REM ****************************

130 LET D$ = CHR$ (4)
140 :
150 INPUT "FILENAME:";FILNAME$
170 :
180 PRINT D$"OPEN"FILNAME$
190 PRINT D$"DELETE"FILNAME$
200 PRINT D$"OPEN"FILNAME$
210 :
220 PRINT "REC";RC + 1;
```

```
230 INPUT ":";REC$
240 IF REC$ = "" THEN 340
250 :
260 REM RC = REC CTR

270 LET RC = RC + 1
280 LET RC$ = STR$ (RC)
290 PRINT D$"WRITE"FILNAME$
300 PRINT REC$
310 PRINT D$: REM TURN OFF WRITE MODE
320 GOTO 220
330 :
340 PRINT D$"CLOSE"
```

Read a Sequential File

This routine reads a sequential text file. It is extremely simple in operation: an endless loop in lines 240 - 250 reads records from the file until the end of the file is reached. This causes an error condition in DOS, which goes to the error handler indicated by line 190. Two important things to notice when using the ONERR statement: ONERR kills any active FOR NEXT loop or subroutine (which means the next NEXT statement or RETURN statement will cause a syntax error) and a POKE 216,0 *MUST* be used immediately after the ONERR statement is executed to keep your Apple functioning properly.

```
100 REM ****************************
110 REM *      READ SEQ FILE       *
120 REM ****************************

130 LET D$ = CHR$ (4)
140 :
150 INPUT "FILENAME:";FILNAME$
170 :
180 PRINT D$"OPEN"FILNAME$
185 PRINT D$"READ"FILNAME$
190 ONERR GOTO 270
200 :
240 INPUT REC$: PRINT REC$
250 GOTO 240
260 :
270 POKE 216,0
280 PRINT D$"CLOSE"
```

Make a Random Access File

Writing records to a random access file is a little trickier than writing to a sequential file. The file *MUST* be opened with an L (record length) parameter (see line 200) and the WRITE mode must be turned on EVERY time you want to write a record to disk. This is to allow you to specify where in the file (with the R, or record number parameter) you wish to place the record. See lines 290 - 300. Also note that variable names used in WRITE statements must be string variables, even if they specify numbers (see the RC$ (record counter) in line 290).

```
100 REM **************************
110 REM *     MAKE RND FILE      *
120 REM **************************

130 LET D$ = CHR$ (4)
140 :
150 INPUT "FILENAME:";FILNAME$
160 INPUT "RECORD LENGTH:";RL$
170 :
180 PRINT D$"OPEN"FILNAME$
190 PRINT D$"DELETE"FILNAME$
200 PRINT D$"OPEN"FILNAME$;",L";RL$
210 :
220 PRINT "REC";RC + 1;
230 INPUT ":";REC$
240 IF REC$ = "" THEN 340
250 :
260 REM RC = REC CTR

270 LET RC = RC + 1
280 LET RC$ = STR$ (RC)
290 PRINT D$"WRITE"FILNAME$",R";RC$
300 PRINT REC$
310 PRINT D$: REM TURN OFF WRITE MODE
320 GOTO 220
330 :
340 PRINT D$"CLOSE"
```

Read a Random Access File

As is similar to WRITE mode, the READ mode must be enabled every time a record is inputted. This is done to allow specific records (the "R" parameter) to be accessed. See lines 230 - 240.

```
100 REM ***************************
110 REM *      READ RND FILE      *
120 REM ***************************

130 LET D$ = CHR$ (4)
140 :
150 INPUT "FILENAME:";FILNAME$
160 INPUT "RECORD LENGTH:";RL$
170 :
180 PRINT D$"OPEN"FILNAME$",L";RL$
190 ONERR GOTO 270
200 :
210 LET RC = RC + 1
220 LET RC$ = STR$ (RC)
230 PRINT D$"READ"FILNAME$",R";RC$
240 INPUT REC$: PRINT REC$
250 GOTO 210
260 :
270 POKE 216,0
280 PRINT D$"CLOSE"
```

Appending Sequential Files

It is fairly easy to append sequential files together, due to the DOS APPEND command. It should be remembered, however, that APPEND is used in place of the OPEN command and that it is intended only for sequential files. See lines 190 - 200 for an example of opening one file to receive an APPENDed file, and the normal OPENing of another file to write it to the end of the first file.

Also note here that the FRE function is used in line 300 (X is a dummy variable) to force "garbage collection." Applesoft does not reuse memory left by reused variable names, but instead grabs another chunk of memory to store data. INPUT statements

(especially in loops) really use up memory and when memory runs out, Applesoft performs "garbage collection" to reuse no-longer-used memory. This generally takes a long time (as much as several minutes) because there is so much memory to sift through. Forcing garbage collection early, when there is less memory to scan, takes much less time.

```
100 REM **************************
110 REM *   APPEND SEQ FILES    *
120 REM **************************

130 LET D$ = CHR$ (4)
140 :
150 PRINT "NAME OF FILE TO APPEND TO:"
    : INPUT F1$
160 PRINT "NAME OF APPEND FILE:": INPUT F2$
170 :
180 ONERR GOTO 230
190 PRINT D$"APPEND"F1$
200 PRINT D$"OPEN"F2$
210 GOTO 270
220 :
230 POKE 216,0
240 PRINT CHR$ (7)"EH?"
250 GOTO 150
260 :
270 ONERR GOTO 350
280 PRINT D$"READ"F2$
290 INPUT REC$
300 LET X = FRE (0)
310 PRINT D$"WRITE"F1$
320 PRINT REC$
330 GOTO 280
340 :
350 PRINT D$"CLOSE"
```

Appending Random Access Files

Appending random access files together requires using Length and Record number parameters (see lines 220 - 240 and 360 - 380). Also remember that the APPEND command finds its way to the

end of the file by reading it sequentially and therefore cannot be used with random access files. A random access file sometimes contains NULL (zero) characters which are incorrectly interpreted by DOS as end of file characters.

The append method here is performed by writing the first random access file to a temporary file and writing the second file to the end of the temporary file. The original first file is deleted and the temporary file is renamed as the first file.

```
100 REM  ******************************
110 REM  *    APPEND RND FILES      *
120 REM  ******************************

130 LET D$ = CHR$ (4)
140 :
150 PRINT "NAME OF FILE TO APPEND TO:"
    : INPUT F1$
160 INPUT "RECORD LENGTH:";L1$
170 PRINT "NAME OF APPEND FILE:": INPUT F2$
180 INPUT "RECORD LENGTH:";L2$
190 IF L2$ < > L1$ THEN PRINT CHR$ (7)"NO!
    FILES MUST HAVE RECS OF EQUAL LEN."
    : GOTO 150
200 :
210 ONERR GOTO 280
220 PRINT D$"OPEN"F1$;",L";L1$
230 PRINT D$"OPEN"F2$;",L";L2$
240 PRINT D$"OPEN TEMP$,L";L1$
250 GOTO 330
260 :
270 REM I/O ERROR:
280 POKE 216,0
290 PRINT CHR$ (7)"EH?"
300 GOTO 150
310 :
320 REM COPY FILE1:
330 ONERR GOTO 440
340 LET R1 = R1 + 1
350 LET R1$ = STR$ (R1)
360 PRINT D$"READ"F1$;",R";R1$
370 INPUT REC$
```

```
380 PRINT D$"WRITE TEMP$,R";R1$
390 PRINT REC$
400 LET X = FRE (0)
410 GOTO 340
420 :
430 REM APPEND FILE2 TO FILE1:
440 POKE 216,0
450 ONERR GOTO 580
460 LET R2 = R2 + 1
470 LET R2$ = STR$ (R2)
480 PRINT D$"READ"F2$;",R"R2$
490 INPUT REC$
500 LET X = FRE (0)
510 PRINT D$"WRITE TEMP$,R"R1$
520 PRINT REC$
530 LET R1 = R1 + 1
540 LET R1$ = STR$ (R1)
550 GOTO 460
560 :
570 REM RESTORE ORIG FILE:
580 PRINT D$"DELETE"F1$
590 PRINT D$"RENAME TEMP$,"F1$
600 :
610 POKE 216,0
620 PRINT D$"CLOSE"
```

External Sort of a Sequential File

This routine sorts a sequential file that is too large to fit in available RAM. It follows the principles described in the sequential file routines listed previously, and works as explained at the beginning of this section. It can also be used to speed up the sorting of intermediate size files as well, because breaking a file into subfiles and sorting them is faster than sorting one large file (sort times are generally exponential according to the number of records involved).

```
100 REM ****************************
110 REM *      EXTERNAL SEQ        *
120 REM *       FILE SORTER        *
130 REM ****************************
```

```
140 LET D$ = CHR$ (4)
150 :
160 INPUT "FILENAME:";FILNAME$
170 INPUT "MAX NO. OF RECS TO READ:";
    MAXRECS
180 :
190 ONERR GOTO 320
200 PRINT D$"OPEN"FILNAME$
210 :
220 REM OPEN SUBFILE:
230 LET PART = PART + 1
240 LET PNAME$ = LEFT$ (FILNAME$,28) +
    "." + STR$ (PART)
250 PRINT D$"OPEN"PNAME$
260 PRINT D$"DELETE"PNAME$
270 PRINT D$"OPEN"PNAME$
280 :
290 PRINT D$"READ"FILNAME$
300 GOTO 370
310 :
320 POKE 216,0
330 PRINT CHR$ (7)"EH?"
340 GOTO 160
350 :
360 REM READ IN A SUBFILE:
370 ONERR GOTO 430
380 FOR I = 1 TO MAXRECS
390 INPUT REC$(I)
400 NEXT
410 GOTO 480
420 :
430 POKE 216,0
440 LET EOF = 1
450 :
460 REM SORT THE SUBFILE:
470 IF EOF AND I = 1 THEN PRINT D$
    "CLOSE"PNAME$: PRINT D$"DELETE"
    PNAME$: GOTO 630
480 LET I = I - 1
490 FOR J = 1 TO I - 1
500 FOR K = J + 1 TO I
510 IF REC$(J) > REC$(K) THEN TEMP$ =
```

```
    REC$(J):REC$(J) = REC$(K):REC$(K) =
    TEMP$
520 NEXT : NEXT
530 :
540 REM WRITE THE SUBFILE:
550 PRINT D$"WRITE"PNAME$
560 FOR J = 1 TO I
570 PRINT REC$(J)
580 NEXT
590 PRINT D$"CLOSE"PNAME$
600 :
610 IF NOT EOF THEN 230
620 :
630 PRINT D$"CLOSE"
```

External Sort of a Random Access File

This is the random access version of the previous external sort routine. It follows principles used by the random file routines listed previously. Note that, like the sequential version, it automatically saves subfiles to disk using the same name as the original, but with ".1", ".2", ".3", etc. appended to the filename.

```
100 REM  ****************************
110 REM  *      EXTERNAL RND        *
120 REM  *        FILE SORTER       *
130 REM  ****************************

140 LET D$ = CHR$ (4)
150 DIM REC$(100)
160 :
170 INPUT "FILENAME:";FILNAME$
180 INPUT "RECORD LENGTH:";RL$
190 INPUT "MAX NO. OF RECS TO READ:";
    MAXRECS
200 :
210 ONERR GOTO 250
220 PRINT D$"OPEN"FILNAME$;",L";RL$
230 GOTO 300
240 :
250 POKE 216,0
```

```
260 PRINT CHR$ (7)"EH?"
270 GOTO 170
280 :
290 REM OPEN SUBFILE:
300 LET PART = PART + 1
310 LET PNAME$ = LEFT$ (FILNAME$,28) +
    "." + STR$ (PART)
320 PRINT D$"OPEN"PNAME$
330 PRINT D$"DELETE"PNAME$
340 PRINT D$"OPEN"PNAME$;",L";RL$
350 :
360 REM READ IN A SUBFILE:
370 ONERR GOTO 450
380 LET I2 = 0
390 LET I = I + 1:I2 = I2 + 1
400 PRINT D$"READ"FILNAME$;",R" STR$ (I)
410 INPUT REC$(I2)
420 IF I2 < MAXRECS THEN 390
430 GOTO 500
440 :
450 POKE 216,0
460 LET EOF = 1
470 :
480 REM SORT THE SUBFILE:
490 IF EOF AND I2 = 1 THEN PRINT D$"CLOSE"
    PNAME$: PRINT D$"DELETE"PNAME$
    : GOTO 650
500 FOR J = 1 TO I2 - 1
510 FOR K = J + 1 TO I2
520 IF REC$(J) > REC$(K) THEN TEMP$ =
    REC$(J):REC$(J) = REC$(K):REC$(K) =
    TEMP$
530 NEXT : NEXT
540 :
550 REM WRITE THE SUBFILE:
560 IF EOF THEN I2 = I2 - 1
570 FOR J = 1 TO I2
580 PRINT D$"WRITE"PNAME$;",R" STR$ (J)
590 PRINT REC$(J)
600 NEXT
610 PRINT D$"CLOSE"PNAME$
620 :
```

```
630 IF NOT EOF THEN 300
640 :
650 PRINT D$"CLOSE"
```

Merge Sequential Files

This routine merges together two sorted sequential text files and saves them under any name you choose. It is normally used to merge together ".1", etc. subfiles generated by the external sort routines. Note that merging is intended for files which are sorted in order. Use the appending routines if you merely want to join two files together.

```
100 REM ***************************
110 REM *    MERGE SEQ FILES     *
120 REM ***************************

130 LET D$ = CHR$ (4)
140 :
150 PRINT "NAME OF 1ST FILE TO MERGE:"
160 INPUT F1$
170 PRINT "NAME OF 2ND FILE TO MERGE:"
180 INPUT F2$
190 PRINT "NAME OF MERGE FILE:"
200 INPUT F3$
210 :
220 ONERR GOTO 240
230 GOTO 280
240 POKE 216,0
250 PRINT CHR$ (7)"EH?"
260 PRINT D$"CLOSE": GOTO 150
270 :
280 PRINT D$"OPEN"F1$
290 PRINT D$"OPEN"F2$
300 PRINT D$"OPEN"F3$
310 PRINT D$"DELETE"F3$
320 PRINT D$"OPEN"F3$
330 :
340 REM GET A REC FROM EACH:
350 PRINT D$"READ"F1$
360 INPUT R1$
```

```
370 PRINT D$"READ"F2$
380 INPUT R2$
390 :
400 ONERR GOTO 480
410 :
420 REM PUT LOWER REC IN MERGE:
430 IF R1$ < = R2$ THEN F1 = 1: PRINT
    D$"WRITE"F3$: PRINT R1$: PRINT
    D$"READ"F1$: INPUT R1$: GOTO 430
440 :
450 LET F1 = 0: PRINT D$"WRITE"F3$: PRINT
    R2$: PRINT D$"READ"F2$: INPUT R2$
    : GOTO 430
460 :
470 REM ONE FILE EMPTY:
480 POKE 216,0
490 PRINT D$
500 ONERR GOTO 570
510 :
520 REM COPY OTHER FILE:
530 IF F1 THEN PRINT D$"WRITE"F3$: PRINT
    R2$: PRINT D$"READ"F2$: INPUT R2$
    : GOTO 530
540 :
550 PRINT D$"WRITE"F3$: PRINT R1$: PRINT
    D$"READ"F1$: INPUT R1$: GOTO 550
560 :
570 POKE 216,0
580 PRINT D$"CLOSE"
```

Merge Random Access Files

This routine is the random access version of the sequential merge routine listed previously.

```
100 REM ***************************
110 REM *   MERGE RND FILES     *
120 REM ***************************

130 LET D$ = CHR$ (4)
140 :
```

```
150 PRINT "NAME OF 1ST FILE TO MERGE:"
160 INPUT F1$
162 INPUT "RECORD LENGTH:";L1$
170 PRINT "NAME OF 2ND FILE TO MERGE:"
180 INPUT F2$
182 INPUT "RECORD LENGTH:";L2$
184 IF VAL (L1$) < > VAL (L2$) THEN PRINT
    CHR$ (7)"RECS MUST BE SAME LENGTH!"
    : GOTO 150
190 PRINT "NAME OF MERGE FILE:"
200 INPUT F3$
210 :
220 ONERR GOTO 240
230 GOTO 280
240 POKE 216,0
250 PRINT CHR$ (7)"EH?"
260 PRINT D$"CLOSE": GOTO 150
270 :
280 PRINT D$"OPEN"F1$;",L";L1$
290 PRINT D$"OPEN"F2$;",L";L2$
300 PRINT D$"OPEN"F3$
310 PRINT D$"DELETE"F3$
320 PRINT D$"OPEN";F3$;",L";L1$
330 :
340 REM GET A REC FROM EACH:
342 LET R1 = R1 + 1:R2 = R2 + 1
350 PRINT D$"READ"F1$;",R"; STR$ (R1)
360 INPUT R1$
370 PRINT D$"READ"F2$;",R"; STR$ (R2)
380 INPUT R2$
390 :
400 ONERR GOTO 480
410 :
420 REM PUT LOWER REC IN MERGE:
430 IF R1$ < = R2$ THEN F1 = 1:R3 = R3 + 1
    : PRINT D$"WRITE"F3$;",R"; STR$
    (R3): PRINT R1$:R1 = R1 + 1: PRINT D$
    "READ"F1$;",R"; STR$ (R1): INPUT R1$:
    GOTO 430
440 :
450 LET F1 = 0:R3 = R3 + 1: PRINT D$
    "WRITE"F3$;",R"; STR$ (R3): PRINT
```

```
    R2$:R2 = R2 + 1: PRINT D$"READ"F2$;
    ",R"; STR$ (R2): INPUT R2$: GOTO 430
460 :
470 REM ONE FILE EMPTY:
480 POKE 216,0
490 PRINT D$
500 ONERR GOTO 570
510 :
520 REM COPY OTHER FILE:
530 IF F1 THEN R3 = R3 + 1: PRINT D$"WRITE"
    F3$;",R"; STR$ (R3): PRINT
    R2$:R2 = R2 + 1: PRINT D$"READ"F2$;",R";
    STR$ (R2): INPUT R2$: GOTO 530
540 :
550 LET R3 = R3 + 1: PRINT D$"WRITE"F3$;",R";
    STR$ (R3): PRINT R1$:R1 = R1 + 1: PRINT
    D$"READ"F1$;",R"; STR$ (R1): INPUT R1$
    : GOTO 550
560 :
570 POKE 216,0
580 PRINT D$"CLOSE"
```

Summary

Text files provide a convenient means of storing large amounts of data on a disk for later retrieval and manipulation.

A centralized collection of data in a file that is available for use by other programs is called a **data base.**

An initialized disk is composed of **35 tracks** and **16 sectors** for a total of 560 sectors or **140K bytes.** DOS and the directory use four of these tracks, bringing the total amount of space available for files to 124K.

A maximum of 105 files can normally be stored on a disk.

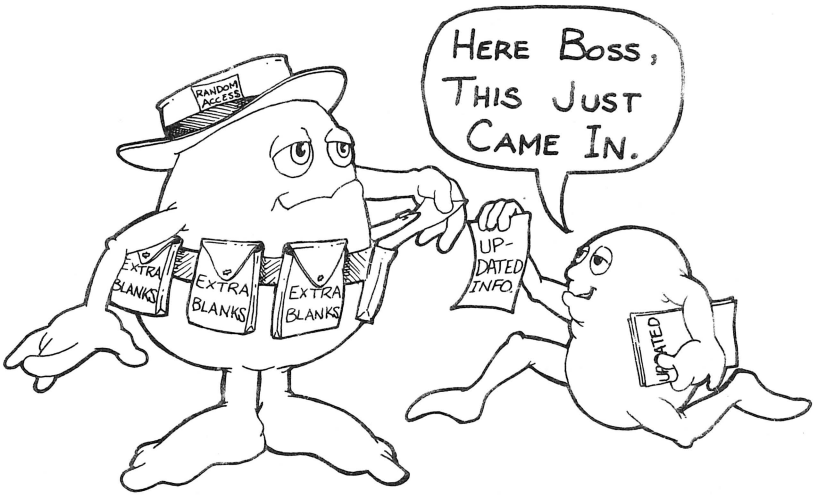
Text files are composed 100% of *Apple ASCII* or *high ASCII* characters. Files are composed of **records,** records are composed of **fields** and fields are composed of ASCII characters. Embedded

within the text file is the end of file marker (ØØ), the end of record marker, <CR> and the end of field marker, (usually a comma, although it can be a *delimiter* of your own choice).

Files can be organized in many ways. **Sequential, random access, ISAM, binary tree** and **B-tree** are the names of just a few of these organization methods. Apple DOS directly supports only the first two methods, sequential and random access.

Sequential files have records with different lengths and sometimes use delimiters to mark the beginning and ending of fields. They are easy to program and provide a fast and compact means of storing information; however, they are difficult to manipulate, and should be used to store information that does not change very often, such as EXEC files, final drafts of documents and receipts.

Random access files do not need delimiters; their fields are easy to identify because they have a uniform record length. The extra length wastes time and space, but allows for easier file manipulation. Random access files are useful for information that needs to be updated frequently, such as inventory records, billing records or possibly a telephone directory.



The presence of commas in a text file can cause loss of data when records are read with the INPUT command, and should be used with a great deal of planning.

An array is one of the best methods for manipulating a file. No time consuming disk drive accesses or confusing DOS commands are needed. In addition, many powerful Applesoft commands are available for manipulating records.

ONERR is a useful error trapping command, but if used within a GOSUB structure or a FOR NEXT loop, it becomes a "loop-killer." In such cases, a GOTO statement must be used to return to the calling routine.

Merge sorting is a technique that allows the sorting of disk files which are too large to fit in memory. It involves repeatedly reading a portion of a large file into memory where it can be sorted quickly, and writing the sorted records to disk in the form of subfiles. The subfiles are then merged together to form a sorted version of the original.

For Further Reading

The DOS Manual (Apple Computer, Cupertino, CA, 1980). The official Apple DOS user's manual. Contains useful and sometimes in-depth suggestions for using DOS.

All About DOS, (Call-A.P.P.L.E., Kent, WA, 1983). A book filled with useful DOS utilities written by users. Explains system bugs and how to overcome them. Detailed examination of various facets of DOS.

Apple Files, David Miller (Reward Books, Reston, VA, 1982). In depth book on text file handling. Covers sequential, random access and even VisiCalc type DIF files.

Beneath Apple DOS, Don Worth and Pieter Lechner (Quality Software, Reseda, CA, 1981). Popular book that goes into great depth in explaining the intricacies of DOS. For the intermediate to advanced user.

CHAPTER 6

ENHANCED GRAPHICS

The Apple II and Apple //e computers have the capability of generating high resolution (also called hi-res) graphics. Incredible things have been done with hi-res graphics on the Apple in the last few years, including creating a pseudo70-column board, projecting 3-dimensional figures, animation, and turning the Apple into a "strip-chart" recorder.

This chapter will focus on one feature, animation, and will show you some simple Applesoft routines to put a little "action" on your screen. We will gradually build a simple "Lunar Lander" game, where you try to land a gently descending spaceship onto a narrow mountain on the surface of the moon. If you miss, the spaceship will explode, complete with visual and sound effects.

Limitations of Applesoft

One of the first facts you'll have to face is that you can only go so far with Applesoft when working with animation. Every time you add another Applesoft statement to make a figure do something, you slow the action down. Add enough statements and the animation will no longer be worthwhile. Practically speaking, you can move one figure at a time and you can only make it do a limited number of things.

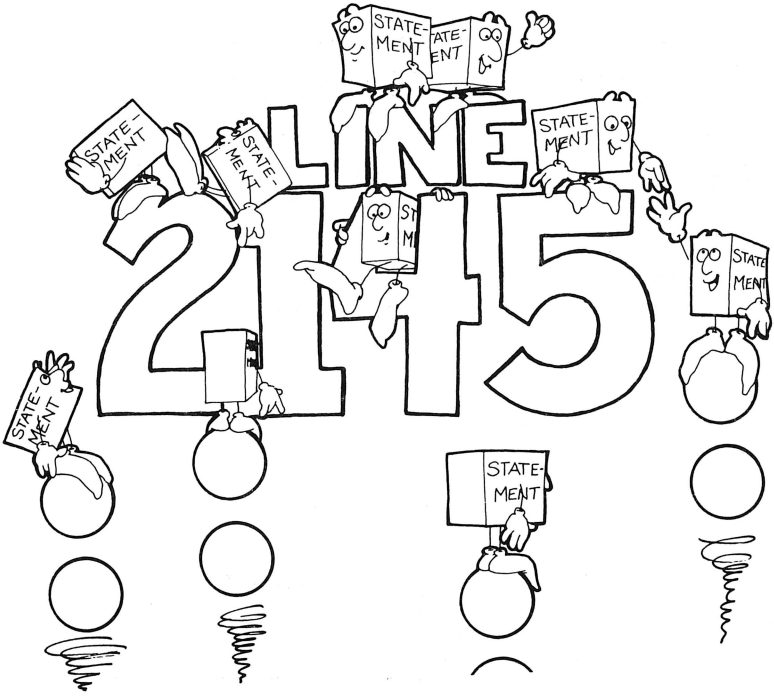
This is not to say that doing animation with Applesoft is not worth the effort. Some very impressive effects can be done if you work within the limitations. An example that comes to mind is a program called *Fly Menu* created by Beagle Bros. Software. When the program is run, a picture of two men (the Beagle Bros. logo) is displayed and a fly buzzes around their heads. When the fly lands on the man on the left, his eyes shift and look at the fly.

The fly then enters his ear, and flies through his head before exiting on the other side. The fly then buzzes around a bit more, stops in mid-air and is suddenly devoured by a anteater-like tongue darting from the other man's mouth. The entire effect is very entertaining and you never are aware that only one figure is being moved at a time.

The limitations of Applesoft can best be overcome by using the following techniques:

- Use shape tables. They are the fastest thing you have going for you.

- To erase a shape, don't use XDRAW. Instead, set HCOLOR to black and draw the same shape over the coordinates you want erased. XDRAW can sometimes leave unerased dots behind.



- Move only one shape at a time.

- Use "page flipping" to move objects around. This reduces visual flicker to a minimum.

- For speed purposes, put as many statements on one line as possible, using a colon to combine them.

- Use variable names instead of numbers. Applesoft takes extra time to convert a number into a format it can use.

- Use only one or two letter variable names. It takes Applesoft longer to go through a program with more characters in it.

Finally, if you really want to get the most out of animation, you will have to use an ultra-fast language like assembly language or FORTH. Be prepared, though; while they are extremely fast in terms of execution speed, these languages are not easy to learn.

Introduction to the Hi-res Screens

There are two "pages" of memory set aside for hi-res graphics, HGR and HGR2. These are also sometimes called "page one" and "page two," respectively. Each page consumes 8,192 bytes (8K) of memory and consists of 192 lines by 280 columns. Each column can hold one dot, which works out to (192 X 280) 53,760 dots on one page! Numbering begins with line 0, column 0 at the upper left corner of the screen:

A HI-RES PAGE

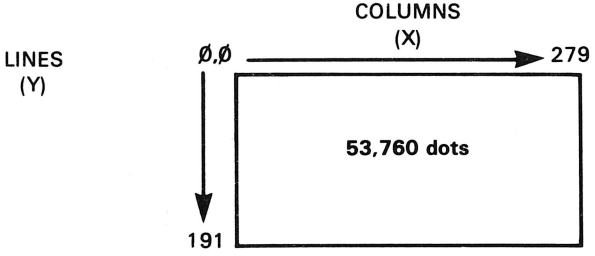


FIGURE 6-1

Some little known facts:

CALL -3100 displays hi-res page 1 without clearing it first, unlike the HGR command. CALL -3086 clears the hi-res screen selected by location 230 (see next paragraph), whether or not that screen is being displayed.

It is possible to draw on or erase either hi-res page without displaying it (i.e., without using a HGR or HGR2 command) by executing the following command first: POKE 230,32 (page one) or POKE 230,64 (page two).

It is possible to display a complete hi-res page one with no text lines at the bottom of the screen and it's also possible to display a "mixed" hi-res page two with 4 lines at the bottom of the screen for text. In fact, it's possible to display any screen without destroying any data just by using the following POKEs:

TEXT-- POKE 49236,0 : POKE 49234,0 : POKE 49233,0
HGR1-- POKE 49239,0 : POKE 49236,0 : POKE 49235,0 : POKE 49232,0
HGR1-- POKE 49239,0 : POKE 49236,0 : POKE 49234,0 : POKE 49232,0 (full screen HGR1)
HGR2-- POKE 49239,0 : POKE 49237,0 : POKE 49234,0 : POKE 49232,0
HGR2-- POKE 49239,0 : POKE 49237,0 : POKE 49235,0 : POKE 49232,0 (mixed screen HGR2)

By way of explanation, the above numbers are called *soft switches* because they are under software control. By POKEing them with any number (0 was used here), they perform certain special functions:

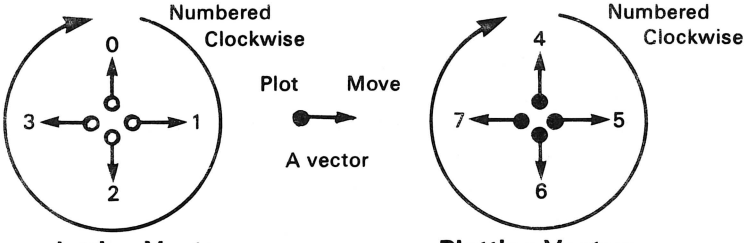
49232: Display graphics
49233: Display text
49234: Select full screen (text or graphics)
49235: Select mixed screen (graphics)
49236: Select page one (text or graphics)
49237: Select page two (text or graphics)
49239: Select hi-res graphics

It's helpful to note that the display switches (49232 and 49233) should be POKEd last to avoid revealing the screen as it changes.

Introduction to Shapes

Shapes are predefined figures that can be quickly and easily drawn on the hi-res screen with the Applesoft DRAW command. Shapes are composed of vectors, which are numbers corresponding to a "plot a dot" decision and a move:

HI-RES SHAPE VECTOR ENCODING SCHEME



Non-plotting Vectors **Plotting Vectors**

Note: Add 4 to non-plotting vector to get equivalent plotting vector.

FIGURE 6-2

A vector consists of a dot either plotted or not plotted on the screen and a move up, down, left or right. Note that the dot is plotted (or not plotted) first and then the movement is made. No other movements (e.g., diagonally) are allowed. This is the reason why diagonal lines have a slight "stairstep" appearance. Also note here that an open circle is used to represent a non-plotted dot. This is unlike the figure used in the *Applesoft BASIC Programming Reference Manual*, which uses just an arrow without any dot at all. The notation used here helps keep the difference between the two types of vectors clear.

A shape can be constructed by drawing these vectors on a sheet of graph paper. As an example, let's define a box with a hole in the top:

SAMPLE BOX DRAWN WITH VECTORS

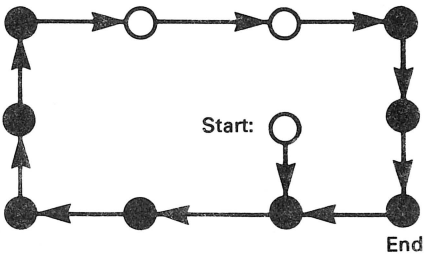


FIGURE 6-3

To keep things simple, the color white will be used for our examples, which causes the dots to be plotted next to each other. Since the dots will overlap each other when they are actually placed on the video screen, the effect will be a delicate white line.

A couple of points worth noting here: it's always good to begin in the middle of your shape, even if there are no dots to be plotted there, to allow the shape to be rotated correctly with the ROT command. Also, although the *Applesoft BASIC Programming Reference Manual* does not explicitly say so, there are two things

you cannot do with vectors: you cannot move up three times in succession without plotting and you cannot move up two times without plotting, and then do a plot. This is because the zero code associated with the move up without plotting vector causes Applesoft to think either it has reached the end of the shape or has reached a part of the shape that is to be ignored.

Once you've properly created your figure, you can begin the process of translating it into a DATA statement by writing the code number next to each vector. The *Applesoft BASIC Programming Reference Manual* uses a method far more complicated than the one shown here. Refer to it if you need more depth in understanding how this process works. For right now, all we need is a clear and simple method that works:

SAMPLE BOX WITH VECTORS & VECTOR CODES

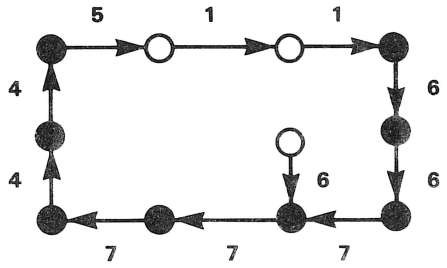


FIGURE 6-4

Next, start at the beginning of your shape and write down the codes in order:

6 7 7 4 4 5 1 1 6 6 7

Now, starting at the left side of the string of numbers, count off three numbers. If the third number is a 1, 2 or a 3, break the string there. If not, back up and break the string into two numbers:

BREAKING THE VECTOR CODE STRING

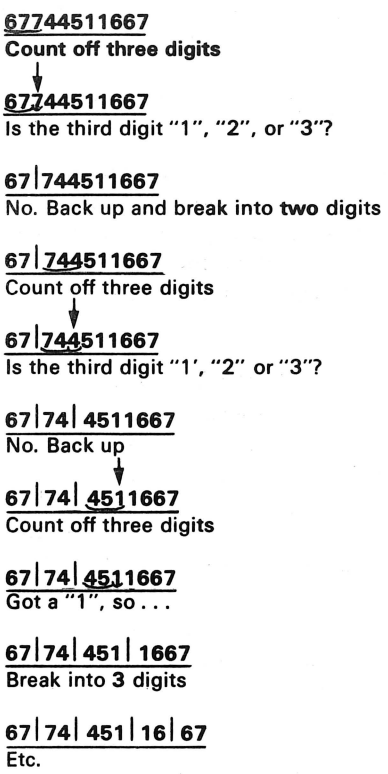


FIGURE 6-5

Convert the code groupings into decimal by multiplying the second number by 8, the third number (if any) by 64 and then adding the results together:

CONVERTING BROKEN VECTOR CODES INTO DECIMAL

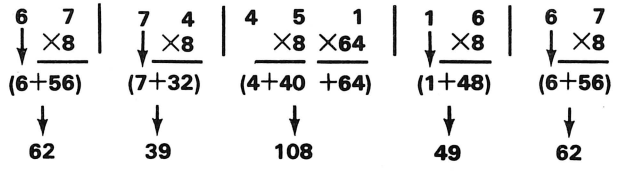


FIGURE 6-6

These numbers are called the *shape definition*. Note how they are condensed into fewer decimal numbers. Add a zero and you can use the numbers directly in a DATA statement:

```
DATA 62,39,108,49,62,0
```

We need one more item before we can use our shape: a *shape index*. The index tells Applesoft how many shapes we are using and where they are located. If you are using one shape, the index is always DATA 1,0,4,0. Just think of income tax form 1040 and you'll find it easier to remember. Here's what the numbers mean:

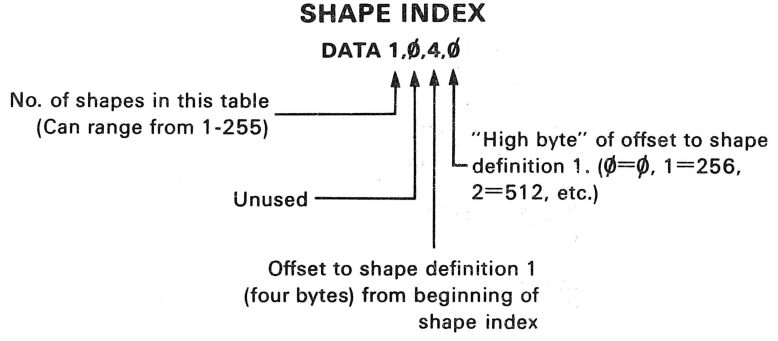


FIGURE 6-7

Suppose that you want to use two shapes and that the second shape is composed of 20 vectors. To do this, you would translate the second shape into an additional DATA statement just like the first and change the index to this:

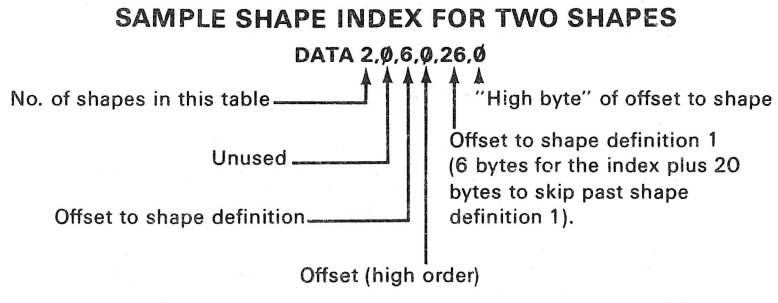


FIGURE 6-8

The shape index and shape definition(s) together comprise the *shape table.* Once the shape table is created, all that is left to do is to load it in memory and tell Applesoft where it is. To load the table in memory from DATA statements, this technique will work:

```
100 DATA 1,0,4,0 : REM  SHAPE INDEX
110 DATA 62,39,108,49,62,0 : REM  SHAPE
    DEFINITION
120 TABLE = 32768 : REM  $8000
130 HIMEM: TABLE
140 ONERR GOTO 160
150 FOR I = 0 TO 1
    : READ BYTE
    : POKE TABLE,BYTE
    : TABLE = TABLE + 1
    : I = 0
    : NEXT
```

As we can see from line 120 above, a good place to put the shape table is high in memory out of the way of the program and then to move the HIMEM pointer down (line 130) to protect the table from being overwritten by the program. The loop in line 150 is an endless loop because index variable I is always reset back to its starting value, 0. This causes the DATA statements to be READ until an out of data error condition occurs. The ONERR statement at line 140 then causes control to continue with the rest of the program.

We must now tell Applesoft where the table has been loaded. Two special memory locations have been set aside for this very purpose: 232 and 233. To use them, we must break the table address into two parts, a "high byte" and a "low byte." The high byte is simply the number of times the address can be divided by 256; the low byte is the remainder:

```
160 TH% = TABLE / 256 : TL% = TABLE - TH% * 256
170 POKE 232,TL% : POKE 233,TH%
```

Note that the low order byte is stored first.

Once this has all been done and hi-res graphics have been initialized with the following mandatory commands:

```
HGR : HCOLOR = 3 : ROT = Ø : SCALE = 1
```

the shape can then be drawn with one simple command:

```
DRAW SHAPE AT LINE,COLUMN
```

Shape Creating Summary:

1. Draw your shape on a piece of graph paper, using vectors and starting in the middle of the shape.

2. Number each vector using the appropriate code.

3. Write the codes down in a string. Make sure you don't have a "000" or a "004" – "005" (illegal moves). Count the number of codes. There should be just as many codes as there are vectors in your drawing.

4. Break the string into groups of two or three digits, depending on whether the third digit is a 1, 2, or 3 (use a three digit group) or a 0,4,5,6,7 (use a two digit group).

5. Convert the groups into decimal numbers by multiplying the second digit by 8 and the third digit (if any) by 64 and adding the results. If you have a result that is greater than 255, you didn't do step 4 correctly. Also, if you have a single digit at the end of the string, just treat it as a decimal number.

6. Put the decimal numbers in a DATA statement and add a zero at the end to indicate to Applesoft where the shape definition ends.

7. Create the shape index and put it in a DATA statement in front of the shape definition(s). The index for a table containing just one shape is always DATA 1,Ø,4,Ø.

8. Decide where you want to load the shape table and READ/POKE the data there. If the table is to be located high in memory, you should move HIMEM down to protect it from your program.

9. Break the address of the table into a low and high order byte pair and POKE them into locations 232 and 233, respectively.

10. Use the DRAW command to put the shape wherever you like.

The Lunar Lander Demonstration

The rest of this chapter will show you how to put the preceding principles to work. We will do this by illustrating how to create and use a Lunar Lander game. When we finish, you'll see how to use shape graphics and the game paddles or a joystick to land a rocket ship on the moon. For now, let's take things one step at a time.

Creating the Lunar Lander Shape

Let's use a simple design:

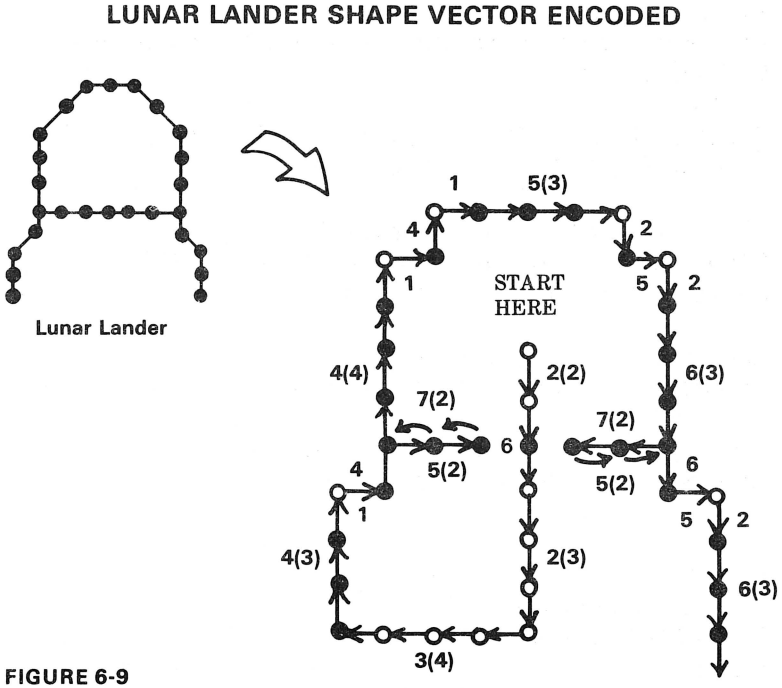


FIGURE 6-9

Note the use of a subscript in the drawing to save time in indicating multiple copies of the same vector code.

When we write the vector codes in a string, we have:

2 2 6 2 2 2 3 3 3 3 4 4 4 1 4 5 5 7 7 4 4 4 4
1 4 1 5 5 5 2 5 2 6 6 6 7 7 5 5 6 5 2 6 6 6

Breaking the string into groups of two and three digits according to the third digit, we have:

22 : 622 : 233 : 33 : 44 : 41 : 45 : 57 : 74 : 44 : 41 : 41
: 55 : 52 : 52 : 66 : 67 : 75 : 56 : 52 : 66 : 6

Multiplying the second digit by 8, the third digit (if any) by 64 and adding the results, we have:

18,150,218,27,36,12,44,61,39,36,
12,12,45,21,21,54,62,47,53,21,54,6

By adding a zero at the end, we can use these numbers as a DATA statement:

```
DATA 18,150,218,27,36,12,44,61,39,36,12,12,
     45,21,21,54,62,47,53,21,54,6,0
```

We can now use this shape definition in a program to display the Lunar Lander on the hi-res screen. The program about to be presented does this in the following manner:

Lines 1080,1120: Initialize variables. To keep the program running fast, most of the numbers used in the program (especially in loops) have been changed to variable names and defined here. Some of the variables defined here are not used yet. They will be later as we enhance the program. Here are the meanings of the variable names:

BEG: Beginning (of anything)
BT: Game button 1
CLRSCRN: Clear the screen
FT: Fifty
N1, N2, N5, N7, N8, N9: Numbers 1, 2, 5, 127, 128, and 96

P1, P2: Hi-res pages 1 and 2
REG: Regular (scale)
SH: Shape number
SP: Select page
SØ, S1, S2, S3, S4: Various soft switches
TABLE: Address of shape table
WHITE: Hi-res color white
Z: Number 0

Lines 1170-1180: Move HIMEM down. This protects the shape table from the program.

Lines 1190-1200: Tell Applesoft where the shape table is.

Lines 1250-1450: Load the shape table. Line 1450 clears the error condition caused by READing past the end of DATA.

Line 1500: Clears the text screen to make sure the bottom four lines of text are empty when we switch to hi-res graphics mode and also makes sure we have the proper color, scale and rotation.

Lines 1550-1580: Select page two and clear it, then select page one and clear it. This does the clearing "invisibly" and exits with page one selected.

Lines 1640-1820: Draw the Lunar Lander on hi-res page one. Line 1640 specifies a screen location near the top, line 1700 displays hi-res page one, line 1780 draws the Lander on the screen and line 1820 makes sure we are displaying hi-res page one in mixed (four lines of text displayed at the bottom) mode.

Lines 1900-1960: Restores HIMEM to its original (before the program was run) setting, waits for a keypress from the user and switches to a clear text screen.

Lunar Lander 1: Display Lander

```
1000 REM ************************************
1010 REM *          LUNAR LANDER 1          *
1020 REM *          DISPLAY LANDER          *
1030 REM *                DEMO              *
1040 REM ************************************
```

```
1050 REM ========================
1060 REM              INITIALIZATION
1070 REM ========================

1080 REM -------------
1090 REM  VARIABLES
1100 REM -------------
1110 BEG = 1
     : BT = 49249
     : CLRSCRN = -3086
     : FT = 50
     : N1 = 1
     : N2 = 279
     : N5 = 5
     : N7 = 127
     : N8 = 128
     : N9 = 96
     : P1 = 32
     : P2 = 64
     : PG = 32
     : REG = 1
1120 SH = 1
     : SP = 230
     : S0 = 49239
     : S1 = 49234
     : S2 = 49232
     : S3 = 49236
     : S4 = 49235
     : TABLE = 32768
     : WHITE = 3
     : Z = 0
1130
     :
1140 REM -------------
1150 REM  POINTERS
1160 REM -------------
1170 OLDHIMEM = PEEK(115) + 256 *
     PEEK(116)
1180 HIMEM: TABLE
     : REM PROTECT TABLE
1190 TH% = TABLE / 256
     : TL% = TABLE - TH% * 256
```

```
1200 POKE 232,TL%
     : POKE 233,TH%
     : REM SET SHAPE PTR
1210
     :
1220 REM ----------------
1230 REM DATA TABLES
1240 REM ----------------
1250 REM ==> SHAPE TABLE

1260 REM SHAPE INDEX
1270 DATA 1,0,4,0
1280
     :
1290 REM SHAPE DEF 1 (LANDER)
1300 DATA 18,150,218,27,36,12,44,61,39,36,12,
     12,45,21,21,54,62,47,53,21,54,6,0
1310
     :
1420 REM ==> LOAD DATA TABLES

1430 ONERR GOTO 1450
1440 FOR I = 0 TO 1
     : READ BYTE
     : POKE TABLE,BYTE
     : TABLE = TABLE + 1
     : I = 0
     : NEXT
1450 POKE 216,0
1460
     :
1470 REM ====================
1480 REM          INIT GRAPHICS
1490 REM ====================

1500 HOME
     : HCOLOR = WHITE
     : SCALE = REG
     : ROT = UP
     : REM MANDATORY STMTS
1510
     :
```

```
1550 FOR I = P2 TO P1 STEP -P1
1560 POKE SP,I
     : CALL CLRSCRN
1580 NEXT
1590
     :
1600 REM -------------------------
1610 REM DRAW LUNAR LANDER
1620 REM -------------------------
1630 REM ==> INITIAL START

1640  X = 140
     :  Y = 10
     :  OY = 10
1650
     :
1690 REM ==> DISPLAY GRAPHICS

1700 POKE S0,Z
  : POKE S1,Z
  : POKE S2,Z
1710
  :
1780 DRAW SH AT X,Y
1820 POKE S3,Z
  : POKE S4,Z
1900 REM ====================
1910 REM          RESTORE SYSTEM
1920 REM ====================

1930 HIMEM: OLDHIMEM
1940
     :
1950 HOME
     : VTAB 22
     : PRINT "DONE!"
     : PRINT "PRESS A KEY TO RETURN TO
       TEXT: ";
     : GET ANS$
     : PRINT ANS$
1960 TEXT
     : HOME
     : PRINT "BYE..."
```

Lunar Lander 2: Lowering the Lunar Lander

Now we come to actual animation. We will add statements that will gradually lower the Lunar Lander by drawing it on screen one and in a little lower position on screen two. Screen one is displayed and then screen two is switched on, giving the illusion that the Lander has moved down a bit. While screen two is being displayed, the original figure on screen one is "invisibly" erased and redrawn a little lower. Page one is then displayed, again giving the illusion that the Lander has moved down. This page flipping process is repeated, until the Lander reaches the bottom of the screen.

Add the following statements to Lunar Lander 1:

```
1660 REM ==> RATE OF FALL & FINISH HEIGHT
1670 RF = 1 : FIN = 100 / RF
1680 :
```

Note: By changing RF, you can control how quickly the Lander falls; however, using values for RF that are less than .52 will cause problems with the sound routine for the Lander.

```
1720 REM ==> MOVE LANDER
1730 FOR I = BEG TO FIN
1740 HCOLOR = BL
     : DRAW SH AT (OX),OY
     : HCOLOR = WH
     : OY = Y
     : OX = X
```

Note: OX *and* OY *refer to old X and old Y. These coordinates must be saved so that the shape can be erased after a new shape is drawn at a different location.* OX *must be enclosed in parentheses above to prevent Applesoft from reading the statement as:*
DRAW SHA TO X,OY

```
1780 Y = Y + RF
     : DRAW SH AT X,Y
     : POKE S3 + N1 * (PG = P2),Z
     : PG = N9 - PG
     : POKE SP,PG
```

```
1790 NEXT
1800
  :
```

*Note: The Lander is made to move down by adding the rate of fall to the Y coordinate. The expression (*PG = P2*) is a* Boolean *expression and returns a 1 if the current page being drawn upon is page two; it returns a 0 if not. The 1 or the 0 control whether N1 (1) will be added to softswitch three, which selects page one or page two.* PG = N9 – PG *toggles the page setting back and forth from page one to page two, causing the shape to be drawn on different pages each time this routine is used.*

Lunar Lander 3: Display Terrain

Now we'll add some ground for the Lander to hit. Because we are working with page flipping, the ground must be drawn on both pages to prevent a visual flicker. Note that the terrain will be drawn before the hi-res pages are displayed, giving a smoother initial viewing. Add the following statements:

```
1520 REM --------------------------------
1530 REM DRAW GRND ON BOTH PGS
1540 REM --------------------------------
1550   FOR I = P2 TO P1 STEP -P1
1560   POKE SP,I
     : CALL CLRSCRN
1570 HPLOT 0,150
     :  HPLOT TO 100,150
     :  HPLOT TO 120,135
     :  HPLOT TO 130,135
     :  HPLOT TO 133,125
     :  HPLOT TO 148,125
     :  HPLOT TO 155,135
     :  HPLOT TO 160,135
     :  HPLOT TO 190,150
     :  HPLOT TO 279,150
1580  NEXT
1590
     :
```

```
1810 REM ==> MAKE SURE ON MTN
1820 POKE SP,P1
     : HCOLOR = BLACK
     : DRAW SH AT (OX),118
     : HCOLOR = WHITE
     : DRAW SH AT (OX),118
     : POKE S3,Z
     : POKE S4,Z
1830
     :
```

(Note: if the rate of fall is adjusted, it could cause the final position of the Lander to go a little further than where we want it to stop. We can correct this by erasing the final position of the Lander and making sure that it is redrawn at the correct location).

Lunar Lander 4: Sound Effects

Sound adds a special flavor to graphics. No matter how simple the sound, it adds new zest to the animation. Here, we will make the Lander beep as it falls, the pitch falling as the Lander falls. We'll close with a higher pitched closing "trill" to add a touch of class to the end of the display. Add the following statements:

```
1350 REM ==> TONE GENERATOR
1360 DATA 32,177,0,32,248,230,142,252,2,32,
     76,231
1370 DATA 142,253,2,32,76,231,142,254,2,138,
     168,174
1380 DATA 252,2,173,48,192,136,208,5,206,
     253,2,240,8,202
1390 DATA 208,245,174,252,2,208,237,173,
     255,2,141
1400 DATA 253,2,206,254,2,208,237,96
1410 :

1450 POKE 216,0
     : SOUND = TABLE - 57
1460
     :
```

Note: The beginning address of the tone generating routine is determined by subtracting the length of the routine from the current setting of TABLE, *which is at the end of the* TABLE.

```
1740 CALL SOUND : FT + I,N5,N1
     : HCOLOR = BL
     : DRAW SH AT (OX),OY
     : HCOLOR = WH
     : OY = Y
     : OX = X
```

Note: The CALL *statement is using a machine code program* POKE*d in previously from the* DATA *statements. Machine code is the only thing that has enough speed to keep up with animation. The three values or parameters following the* CALL *statement control pitch, duration and the number of times to repeat the tone.*

```
1870 REM ==> CLOSING TRILL
1880 FOR I = 0 TO 64 STEP 8
     : CALL SOUND: 100 - I,10,1
     : NEXT
1890
     :
```

Lunar Lander 5: Button Control

This example will show you how to use button 1 on either the game paddle or joystick to "freeze" the action and stop the Lunar Lander in mid-air. Add the following statement:

```
1750 IF PEEK(BT) > N7 THEN WAIT BT,N8,N8
```

Note: The WAIT *command makes the computer "freeze" until the location tested (button 1 in this case) changes. In this statement, if button 1 is pushed,* PEEK(BT) *will become greater than 127 and the action will freeze until bit value 128 (the first N8) of BT is turned off (as indicated by the second N8) by releasing the button. For further information on the* WAIT *command, consult the* ***Applesoft Reference Manual.***

Lunar Lander 6: Joystick/Paddle Control

This example will show you how to use a joystick or a game paddle to "steer" the Lunar Lander as it descends. Game paddles can be read with the PDL(Ø) and PDL(1) functions. A joystick is read the same way; you can think of a joystick as a set of game paddles at right angles to each other. To keep things simple, we'll read PDL(Ø) which also corresponds to the joystick being moved along one axis. Add the following statements:

```
1640 X = INT(270 * RND(1))
     : Y = 10
     : OY = 10
1650
     :
```

Note: This causes the Lander to be placed at a random column when the game starts. This makes it a little more challenging to land it on the mountain.

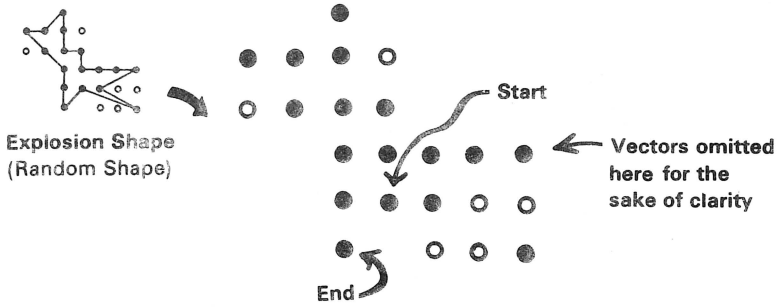
```
1760 X = X + N1 * (PDL(Z) > N7)
     : X = X - N1 * (PDL(Z) < N7)
     : IF X > N2 THEN X = N1
1770 IF X < N1 THEN X = N2
```

Note: The horizontal position of the Lander is incremented if the paddle/joystick is turned toward the right past the halfway point of 127 and decremented if turned toward the left past the halfway point. Wraparound (moving the Lander from one edge of the screen to the other) is provided by the two IF *statements should the Lander stray too far to one side.*

Lunar Lander 7: Explosions

This example will show you how to use a second shape and rotate it quickly to simulate an explosion. It will be used here if the Lander fails to land on the mountain. The explosion shape looks like this:

EXPLOSION SHAPE VECTOR ENCODED



Vectors:	55	05	77	74	43	46	77	61	56	55	56	52	73
Decimal:	45	40	63	39	28	52	63	14	53	45	53	21	31

Vectors:	07	76	6
Decimal:	56	55	6

DATA 45,40,63,39,28,52,63,14,53,45,53,21,31,56,55,6,0

FIGURE 6-10

Add the following statements:

```
1270 DATA 2,0,6,0,29,0

1320 REM SHAPE DEF 2 (EXPLOSION)
1330 DATA 45,40,63,39,28,52,63,14,53,45,53,21,
     31,56,55,6,0
1340
     :

1840 REM ==> EXPLODE IF NOT
1850 IF X > 150 OR X <
     130 THEN HCOLOR = BLACK
     : DRAW SH AT (OX),118
     : FOR R = 0 TO 112 STEP 8
     : ROT = R
     : HCOLOR = WHITE
     : DRAW SH + 1 AT (OX),118
     : CALL SOUND: 255,5,1
     : HCOLOR = BLACK
     : DRAW SH + 1 AT (OX),118
     : NEXT
1860
     :
```

The Complete Lunar Lander Program

To let you see what the program looks like with all the additions and enhancements, here is an updated listing:

```
1000 REM ************************************
1010 REM *          LUNAR LANDER            *
1020 REM *            COMPLETE              *
1030 REM *              DEMO                *
1040 REM ************************************
1050 REM ====================
1060 REM       INITIALIZATION
1070 REM ====================

1080 REM  ------------
1090 REM  VARIABLES
1100 REM  ------------
1110 BEG = 1
     : BT = 49249
     : CLRSCRN = -3086
     : FT = 50
     : N1 = 1
     : N2 = 279
     : N5 = 5
     : N7 = 127
     : N8 = 128
     : N9 = 96
     : P1 = 32
     : P2 = 64
     : PG = 32
     : REG = 1
1120 SH = 1
     : SP = 230
     : S0 = 49239
     : S1 = 49234
     : S2 = 49232
     : S3 = 49236
     : S4 = 49235
     : TABLE = 32768
     : WHITE = 3
     : Z = 0
1130
```

```
      :
1140 REM ------------
1150 REM POINTERS
1160 REM ------------
1170 OLDHIMEM = PEEK(115) + 256 *
      PEEK(116)
1180 HIMEM: TABLE
      : REM PROTECT TABLE
1190 TH% = TABLE/256
      : TL% = TABLE - TH% * 256
1200 POKE 232,TL%
      : POKE 233,TH%
      : REM SET SHAPE PTR
1210
      :
1220 REM ----------------
1230 REM DATA TABLES
1240 REM ----------------

1250 REM ==> SHAPE TABLE

1260 REM SHAPE INDEX
1270 DATA 2,0,6,0,29,0
1280
      :
1290 REM SHAPE DEF 1 (LANDER)
1300 DATA 18,150,218,27,36,12,44,61,39,36,12,
      12,45,21,21,54,62,47,53,21,54,6,0
1310
      :
1320 REM SHAPE DEF 2 (EXPLOSION)
1330 DATA 45,40,63,39,28,52,63,14,53,45,53,21,
      31,56,55,6,0
1340
      :
1350 REM ==> TONE GENERATOR

1360 DATA 32,177,0,32,248,230,142,252,2,32,
      76,231
1370 DATA 142,253,2,32,76,231,142,254,2,138,
      168,174
1380 DATA 252,2,173,48,192,136,208,5,206,
      253,2,240,8,202
```

```
1390 DATA 208,245,174,252,2,208,237,173,255,
     2,141
1400 DATA 253,2,206,254,2,208,237,96
1410
     :
1420 REM ==> LOAD DATA TABLES

1430 ONERR GOTO 1450
1440 FOR I = 0 TO 1
     : READ BYTE
     : POKE TABLE,BYTE
     : TABLE = TABLE + 1
     : I = 0
     : NEXT
1450 POKE 216,0
     : SOUND = TABLE - 57
1460
     :
1470 REM ====================
1480 REM          INIT GRAPHICS
1490 REM ====================

1500 HOME
     : HCOLOR = WHITE
     : SCALE = REG
     : ROT = UP
     : REM MANDATORY STMTS
1510
     :
1520 REM ------------------------------
1530 REM DRAW GRND ON BOTH PGS
1540 REM ------------------------------
1550 FOR I = P2 TO P1 STEP -P1
1560 POKE SP,I
     : CALL CLRSCRN
1570 HPLOT 0,150
     : HPLOT TO 100,150
  : HPLOT TO 120,135
     : HPLOT TO 130,135
     : HPLOT TO 133,125
     : HPLOT TO 148,125
     : HPLOT TO 155,135
```

```
      : HPLOT TO 160,135
      : HPLOT TO 190,150
      : HPLOT TO 279,150
1580 NEXT
1590
      :
1600 REM -------------------------
1610 REM DRAW LUNAR LANDER
1620 REM -------------------------
1630 REM ==> INITIAL START

1640 X = INT((270 * RND(1)))
      : Y = 10
      : OY = 10
1650
      :
1660 REM ==> RATE OF FALL & FINISH HEIGHT

1670 RF = 1
      : FIN = 108/RF
1680
      :
1690 REM ==> DISPLAY GRAPHICS

1700 POKE S0,Z
      : POKE S1,Z
      : POKE S2,Z
1710
      :
1720 REM ==> MOVE LANDER

1730 FOR I = BEG TO FIN
1740 CALL SOUND: FT + I,N5,N1
      : HCOLOR = BL
      : DRAW SH AT (OX),OY
      : HCOLOR = WH
      : OY = Y
      : OX = X
1750 IF PEEK(BT) > N7 THEN WAIT BT,N8,N8
1760 X = X + N1 * (PDL(Z) > N7)
      : X = X - N1 * (PDL(Z) < N7)
      : IF X > N2 THEN X = N1
```

```
1770 IF X < N1 THEN X = N2
1780 Y = Y + RF
     : DRAW SH AT X,Y
     : POKE S3 + N1 * (PG = P2),Z
     : PG = N9 - PG
     : POKE SP,PG
1790 NEXT
1800
     :
1810 REM ==> MAKE SURE ON MTN

1820 POKE SP,P1
     : HCOLOR = BLACK
     : DRAW SH AT (OX),OY
     : HCOLOR = WHITE
     : DRAW SH AT (OX),118
     : POKE S3,Z
     : POKE S4,Z
1830
     :
1840 REM ==> EXPLODE IF NOT

1850 IF X > 150 OR X < 130 THEN
       HCOLOR = BLACK
     : DRAW SH AT (OX),118
     : FOR R = 0 TO 112 STEP 8
     : ROT = R
     : HCOLOR = WHITE
     : DRAW SH + 1 AT (OX),118
     : CALL SOUND: 255,5,1
     : HCOLOR = BLACK
     : DRAW SH + 1 AT (OX),118
     : NEXT
1860
     :
1870 REM ==> CLOSING TRILL

1880 FOR I = 0 TO 64 STEP 8
     : CALL SOUND: 100 - I,10,1
     : NEXT
1890
     :
```

```
1900 REM ====================
1910 REM        RESTORE SYSTEM
1920 REM ====================

1930 HIMEM: OLDHIMEM
1940
     :
1950 HOME
     : VTAB 22
     : PRINT "DONE!"
     : PRINT "PRESS A KEY TO RETURN TO TEXT: ";
     : GET ANS$
     : PRINT ANS$
1960 TEXT
     : HOME
     :
     : PRINT "BYE..."
```

Summary

The Apple is capable of **high resolution** graphics and animation. However, if Applesoft is used to express the program, the degree of sophistication of the animation will be limited.

Shape tables, page flipping and **speed tips** are all necessary to get Applesoft to work fast enough with animation.

There are two hi-res pages in memory, each one taking up 8,192 bytes of RAM.

Through the use of CALLs and soft switches accessed through certain POKEs, it is possible to display hi-res pages without clearing any information stored in them and also to draw and erase on a page without displaying it. It is also possible to display a full screen page 1 and a mixed page two, unlike the respective HGR and HGR2 commands.

Shapes are pre-defined figures composed of vectors. **Vectors** are plotted or unplotted dots, followed by a move. A **shape table** is composed of a shape index and up to 255 shape definitions.

You cannot move up three times in succession without plotting a point, nor can you move up twice without plotting and then do a plot.

There are one-digit numerical codes associated with each of the eight vectors. The codes can be converted into decimal and used in DATA statements.

You must always define the hires color, ROT and SCALE values, or else the shape will not be drawn correctly.

CHAPTER 7

SPECIAL PRINTER TECHNIQUES

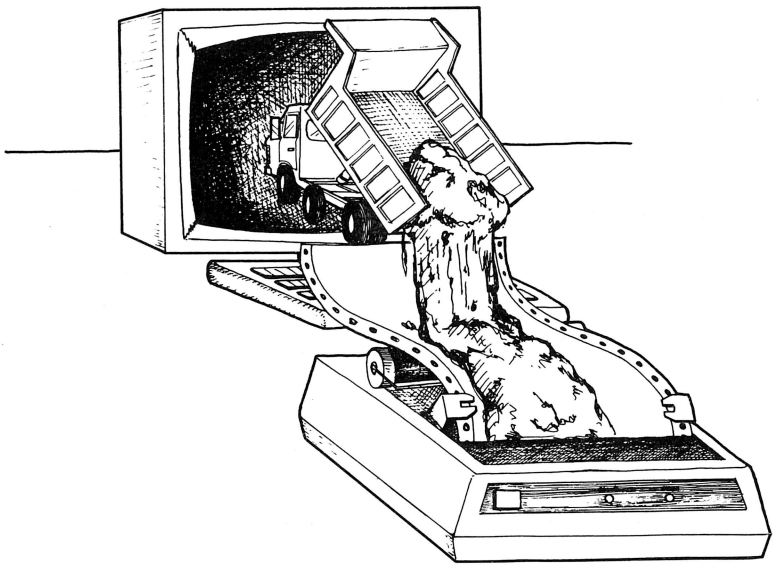
Three Types of Printers

There are three types of printers commonly available for personal computers today:

—Thermal
e.g., **Silentype**

—Dot matrix
e.g., **Epson, Prowriter**

—Letter Quality (daisy wheel)
e.g., **Brother, C. Itoh, Diablo, Qume**



Thermal printers use a specially treated paper that turns black when it is heated. Characters are formed by passing a hot print head over the paper. Thermal printers are compact, lightweight, quiet and easy to operate. They can easily handle hi-res screen dump. Their use is on the decline mainly because of the type of paper they use: it is smaller and more expensive than regular paper, and it contributes to the relatively poor quality of the characters generated on it.

Dot matrix printers form characters by striking tiny pins against a ribbon adjacent to a sheet of paper. The pins form a matrix of small dots on the paper that resemble characters. The quality of the appearance of the letters depends on the density of the dots. Dot matrix printers are fast and economical. They can also handle graphics printing such as hi-res screen dumps. They can use a wide assortment of regular paper, including mailing labels, typing paper, stationery and fan-folded computer paper. Some business users find that dot matrix characters are not of sufficient quality for correspondence purposes. For this reason, dot matrix printers are also known as "near-letter quality" printers. This situation may ease somewhat as printer manufacturers develop character matrices with higher densities.

Letter quality printers use a small hammer to strike a spinning daisy wheel of characters against a ribbon and paper. Characters formed this way, especially when made with the use of film ribbons, are crisp, clear and of superior quality. Letter quality printers are expensive, slow and are generally not capable of generating graphics, such as hi-res screen dumps.

Printing Out Normal Text

Most printers have a simple way of operating with the Apple: simply use PR#1 and PRINT statements to send text to the printer and use PR#Ø to stop:

```
1Ø PR#1
2Ø PRINT "THIS IS A TEST"
3Ø PR#Ø
```

Results:

```
THIS IS A TEST
```

One "inside tip" that can help you avoid a puzzling error condition is to use CALL 1002 after a PR# instruction if you plan on using DOS commands from within your program. For example, this does *not* work:

```
10 PR#1
20 PRINT CHR$(4)"CATALOG"
30 PR#0
```

Results:

CATALOG (prints the word, doesn't do a CATALOG)

But this does:

```
10 PR#1   : CALL 1002
20 PRINT CHR$(4)"CATALOG"
30 PR#0   : CALL 1002
```

Results:

(prints a CATALOG of the disk)

The PR# statements disconnect DOS, but CALL 1002 connects it again. A more detailed explanation of this problem can be found on pg. 102 of *The DOS Manual.*

Another technique that works is:

```
10 PRINT CHR$(4)"PR#1"
```

The problem with this method is that it sends a carriage return to the screen. This may cause the screen to scroll and ruin a text dump that you might be attempting.

Special Printer Commands

Computer printers can go far beyond the printing of normal text. They can emphasize, italicize, proportionally space, double space, subscript, tab, underline, and much more. These features can be activated by using special printer command codes. Each printer has its own set of codes and the instruction manual included with the printer can further explain these commands. A problem immediately arises, however, in trying to translate the instruction manual into something that the Apple can execute.

Here's an example from Appendix B of Epson's *Graftrax Plus Printer Manual*:

Epson Manual:

Dec	Hex	Symbol	Function
15	0F	SI	Shift in. Turns on compressed character mode. Does not work with emphasized mode. Stays on until cancelled by DC2 (18).

Translation:

Command Code	Applesoft	Function
<ESC> "E"	CHR$(27) "E"	Turn on emphasized mode

Example:

```
PR#1
PRINT CHR$(15)"THIS IS A TEST"
```

Epson Manual (Original MX-100):

```
ESC D + n1 + n2 + ..... + nk + NUL
(1<=(n)d<=233, k<=12)
```

Example:

```
(DATA)·ESC D <5>H <A>H <15>H NUL ABC HT
       DEF HT GHI HT JKL CR LF
(PRINT) ABC DEF GHI      JKL
```

Translation:

<ESC> "D"; 1st tab setting; 2nd tab setting; . . . ;last tab setting; CHR$(Ø)

Tab settings may range from 1 to 233; up to 12 tabs may be set.

Example:

```
PRINT CHR$(27)"D";CHR$(5);CHR$(10);
    CHR$(21);CHR$(0);
PRINT "ABC";CHR$(9);"DEF";CHR$(9);"GHI";
    CHR$(9)"JKL"
```

Results:

```
ABC DEF GHI     JKL
```

The point of these examples is that you need to be able to translate what is in the printer manual into something that you can use on your Apple. The following guidelines will help:

1. There are usually just two types of printer command codes: control codes and ESCape codes. An example of a control code is <CTRL O> or CHR$(15), which on the Epson turns on compressed character mode. An example of an ESCape code is <ESC> "E", which on the Epson, turns on emphasized mode.

2. You cannot send the <ESC> character to the printer merely by pressing the ESC key. The Apple will intercept the key and try to interpret it as an ESC I,J,K or M "cursor move" command. Use PRINT CHR$(27) to send the <ESC> character to the printer.

3. Use the ASCII chart in the *Applesoft BASIC Programming Reference Manual* to convert any strange codes like NUL, BEL, ESC, etc., that you may encounter.

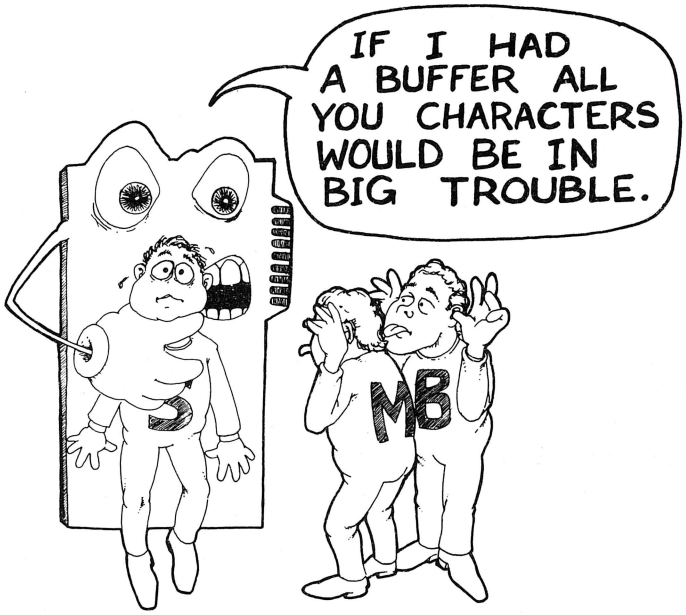
Programming the Printer Interface Card

Sitting quietly inside your Apple, usually in slot 1, is a printer interface card, without which, as they say in the movies, printing would not be possible. It does the following important functions:

1. It receives characters from the Apple that are to be printed and holds them until the printer is ready to pick them up.

This "middleman" technique is necessary because the Apple can print characters at a speed of greater than 19,000 characters per second, while your printer can only print at speeds typically ranging from 10 to 200 characters per second.

A normal printer interface card can hold only one character at a time, so this means that your card will freeze your Apple until the printer gets all of the characters for printing.



Some printer interface cards known as printer buffers are capable of holding more than one character at a time; typically they can hold at least 16,384 characters. This allows your Apple to send all of the characters to be printed to the buffer in a second or so. You then have full control over your Apple and can move on to other programming projects. The printer, on the other hand, can get its characters from the buffer whenever it wishes.

2. A program in the printer interface card watches all characters being sent to it, and when it receives a carriage return, it sends it on to the printer, but it also adds a line feed to advance the paper. Double spacing or no line feeding may be the result of not setting your printer's internal switch settings to acknowledge this fact.

3. Some programs in printer interface cards have advanced features such as the ability to perform a hi-res screen dump. These features are usually activated by sending the program a CTRL I (CHR$(9)). This means you will have trouble sending a CTRL I to the printer for other uses such as a tab command or a graphics character. In addition, there are certain other control characters that activate certain features in a printer interface card. The most notorious of these characters are CHR$(7) through CHR$(13).

   To get around this problem, you can avoid the program in the card and send characters directly to the storage area on the card itself. The printer can then pick the character up without interference.

   The storage area on the card is called a *data latch.* You can put a character there with a simple POKE. POKE 49296,ASC("A") for example, will put the ASCII code for the letter "A" in the Epson APL parallel printer interface card data latch. The addresses for data latches vary from card manufacturer to card manufacturer, but they can usually be found in the back of your card's user's manual.

   There is one last thing you need to know if you are putting characters directly into your interface card. The printer does not know you have put the character in the card unless you specifically tell it. This is done by raising or lowering a strobe signal, which is a little like ringing a doorbell to get someone's attention. Some cards (such as the Epson APL) have a built-in hardware strobe, which means as soon as you put a character in the data latch, the card immediately tells the Epson that the character is there. You don't have to worry about anything else. Other cards require a software strobe, which means another POKE: (The Grappler is used for the following example).

```
POKE 49297,ASC("A") : REM Put character in data latch
POKE 49298,ASC("A") : REM Lower strobe (alerts
    Epson)
POKE 49300,ASC("A") : REM Raise strobe back to
    normal
```

Some cards don't even use a data latch: (The PKASO card is used for the following example.)

```
POKE 51201,ASC("A") : REM Make sure strobe is up
POKE 51200,ASC("A") : REM Lower strobe (alerts
                      Epson)
POKE 51201,ASC("A") : REM Raise strobe back
                      to normal
```

In some cases, the information you need is expressed in an assembly language program in the back or an Appendix of the card's user's manual. Here's an example of how to translate such a program into Applesoft: (Example used here is from the PKASO manual).

Data Output to the Printer:

```
1   WAIT LDA $C090      See if printer is ready.
2        ROL A          Get bit 6 (Ready flag).
3        BPL WAIT       Not yet; keep looping until it is.
4        LDA DATA       Get the character to print.
5        STA $C801      Make sure strobe is up.
6        STA $C800      Lower strobe to alert printer to
                        get char.
7        STA $C801      Raise strobe up again.
```

Note: In line 2, ROL *and not* ROL A *is usually used by most Apple assemblers.*

The Applesoft equivalent of the program would be:

```
1   WAIT 49296,64 : REM     Wait til printer is ready.
4   A = ASC("B") : REM      Get the character ("B")
                            to print.
5   POKE 51201,A : REM      Make sure strobe is up.
6   POKE 51200,A : REM      Lower strobe to alert
                            printer.
7   POKE 51200,A : REM      Raise strobe up again.
```

Note that line 1 of the above Applesoft program does the work of the first three lines of the assembly language program.

Hi-res Screen Dump

After you determine your card's data latch and strobing procedure, you can use the following Applesoft program to do a hi-res screen dump on any dot matrix printer. Special thanks to *Bob Diaz* of *Epson America* who wrote the original version of this program. Note that the screen characters must be POKEd directly into the card to avoid compatibility problems with characters in the range of CHR$(7) – CHR$(13).

This program is set to work with the Epson MX-80 printer and the Epson APL (Apple) printer interface card; remember to adjust the WAIT and POKE addresses in line 200 if you are using a different printer system. Be patient with the program; there will be a long wait after each line is printed . . . Applesoft is just plain slow at times.

Explanation of the Program:

Line 130 goes to a graphics subroutine that draws a figure on hi-res screen one.

Line 150 defines variable names and constants used as Epson command codes.

Line 170 turns on printer output, sets line spacing to seven dots (graphics spacing) and sets the page number to hi-res page one (use 16384 for page two).

Lines 190-200 scan the hi-res screen along the right edge from top to bottom, convert a hi-res screen byte into a bit image character and sends it to the printer. A hi-res page is divided into three zones, each zone is divided into eight sections and each section is divided into eight lines. Line 190 indents the figure to keep it centered on the paper and also prepares the printer to receive 256 "bit image" characters. The beginning of line 200 sends 64 blanks to make the total number of graphics characters sent to the printer add up to 256.

Line 220 restores normal line spacing, video output and text mode.

Finally, note that the hi-res screen is printed *sideways* on the paper to simplify the calculations involved in getting a byte from the screen and printing it out on paper.

```
100 REM ************************************
110 REM *    HI-RES SCREEN DUMP PROG     *
120 REM ************************************
130 GOSUB 250
    : REM PUT FIGURE ON SCREEN
140
    :
150 ESC$ = CHR$ (27)
    : EA$ = ESC$ + "A"
    : EK$ = ESC$ + "K"
    : E1$ = ESC$ + "1"
    : E2$ = ESC$ + "2"
160
    :
170 PR#1
    : CALL 1002
    : PRINT E1$;
    : PG = 8192
180
    :
190 FOR X = 39 TO 0 STEP -1
    :   PRINT SPC(10);EK$ CHR$(0) CHR$(1);
200     FOR I = 1 TO 64
    :   PRINT CHR$(0);
    :   NEXT
    :     FOR ZN = 0 TO 80 STEP 40
    :       FOR SC = 0 TO 896 STEP 128
    :           FOR LN = 0 TO 7169 STEP 1024
    :           WAIT 49601,128,128
    :         POKE 49296, PEEK(ZN + SC +
              LN + X + PG)
    :         NEXT
    :       NEXT
    :     NEXT
    :     PRINT
    :   NEXT
210
    :
```

```
220 PRINT E2$
    : PR#0
    : CALL 1002
    : TEXT
    : END
230
    :
240 REM +-------------------------------------+
250 REM :      FIGURE DRAWING ROUTINE     :
260 REM +-------------------------------------+
270
    :
280 HGR2
    : HCOLOR = 3
    : FOR X = 0 TO 279 STEP 4
    :     HPLOT 140,96 TO X,0
    : NEXT
    : FOR Y = 0 TO 191 STEP 4
    :   HPLOT 140,96 TO 279,Y
    : NEXT
    : FOR X = 279 TO 0 STEP -4
    :   HPLOT 140,96 TO X,191
    : NEXT
    : FOR Y = 191 TO 0 STEP -4
    :   HPLOT 140,96 TO 0,Y
    : NEXT
290 HCOLOR = 0
    : FOR I = 2 TO 20 STEP 2
    :   HPLOT110+I,70+I TO 170-I,70+I TO 170-I,
        120-I TO 110+I,120-I TO 110+I,70+I
    : NEXT
300 RETURN
```

Summary

There are three types of printers commonly available for personal computers today: **thermal, dot matrix and letter quality.**

Information on the Apple is printed with a simple procedure:

```
10 PR#1
20 PRINT "THIS IS A TEST"
30 PR#0
```

CALL 1002 should be used after a PR# statement if DOS commands are to be used from within a program.

Printers are capable of performing special functions such as **underlining, subscripting, italicizing, tabbing, double spacing,** and the like. The features your printer is capable of performing are found in your owner's manual. To decipher the owner's manual, remember that ESC can be expressed as PRINT CHR$(27) and CTRL codes can be expressed as CHR$(0) through CHR$(31). An ASCII chart can help you translate any names that you may not know such as HT, FF, CR, NUL, and BEL.

The **printer interface card** is a necessary middleman to receive characters from your Apple until the printer can get them. A **printer buffer** is a special kind of interface card that returns control of your Apple back to you and yet allows the printer to get characters as it needs them from the buffer.

There is a program within your printer interface card that performs certain extra actions when it receives special characters, usually within the range of CHR$(7) to CHR$(13). You can avoid these extra actions and send characters directly to the printer by POKEing characters directly into the card's **data latch.** Some cards also require you to POKE a software strobe to tell the printer that a character is waiting to be picked up.

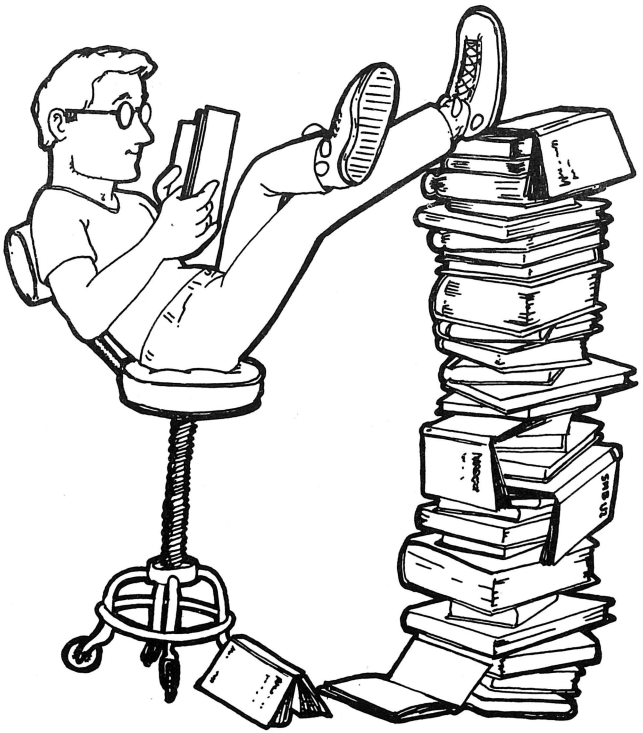
It is possible to use an Applesoft program to print out a copy, or **dump** of the hi-res screen. To do this, however, characters must be POKEd directly into the printer interface card to avoid compatibility problems with the program already in the card.

For Further Reading

The Elementary Apple, William Sanders (DATAMOST, Inc., Chatsworth, CA, 1983) Introduction to several different types of printers and the basics of how to use them.

The Other Epson Manual, Bill Parker (Quality Software, 1984). A thorough examination of the entire series of Epson dot matrix printers. Numerous tips, sample programs and charts. Written specifically for the frustrated Apple user.

Programming the Apple: A Structured Approach, J.L. Campbell and Lance Zimmerman (Robert J. Brady Co., Bowie, MD, 1982). Good overview of various types of printers and how to use them.



CHAPTER 8

PEEKS, POKES, CALLS AND TRICKS OF THE TRADE

Ampersand

The ampersand character (&), when used within Applesoft, causes the Apple to transfer control to a machine code program indicated by the ampersand vector.

Example:

```
100 REM MACHINE CODE TO CLEAR THE SCREEN
110 DATA 32,88,252,96
120 :
130 REM ADDRESS TO STORE MACHINE CODE
140 ADDR = 768
150 AH% = INT(ADDR/256) : AL% = ADDR -
    AH% * 256
160 :
170 REM PUT MACHINE CODE IN MEMORY
180 FOR I = 0 TO 3   : READ BYTE   : POKE
    ADDR + I,BYTE   : NEXT
190 :
200 REM SET UP AMPERSAND VECTOR
210 POKE 1014,AL%   : POKE 1015,AH%
220 :
230 REM AMPERSAND DEMO
240 & : REM THIS CLEARS THE SCREEN
```

Applesoft Program Pointers & Locations:

Beginning of program	: PEEK(103) + 256*PEEK(104)
End of program	: PEEK(175) + 256*PEEK(176)

Beg of simple variable table	: PEEK(105) + 256*PEEK(106) (LOMEM)
Beg of array table	: PEEK(107) + 256*PEEK(108)
End of program and all tables	: PEEK(109) + 256*PEEK(110)
Hi-res page 1 Range	: 8192 – 16383
Hi-res page 2 Range	: 16384 – 24576
End of free space	: PEEK(111) + 256*PEEK(112)
HIMEM	: PEEK(115) + 256*PEEK(116)

DOS

Buffers

With a normal boot-up, DOS 3.3 creates three *buffers* for use in reading and writing files from the disk. The buffers reside just below DOS and are protected by HIMEM which is moved to just below the buffers. Each buffer uses 595 bytes of space, and most users never use the last two buffers. While the MAXFILES 1 command can eliminate unused buffers, you may want to use the buffers to hold temporary values or machine code programs instead. Here are some useful formulas for finding the buffers:

Top of buffers:	TB = PEEK(40192) + 256*PEEK(40193) + 37
No. of buffers:	NB = PEEK(43607)
Btm of buffers:	BB = TB – 595*NB + 1
Beg of any buf:	BA = TB – 595*(buffer no.: 1-3)
Addtnl buffers that can be created:	AB = PEEK(43608)

COMMAND/ERROR TABLES

The command table contains the names of all valid DOS commands; the error table contains the names of all DOS error messages generated when an error condition is encountered. Once you know the locations of the commands and error messages, you can easily change them with a few POKEs, giving you the ability to create friendlier commands (such as "CAT" instead of "CATALOG") and error messages (such as "CUT A NOTCH!" instead of "WRITE PROTECTED"). Here are the locations:

Command Table: (read down)

43140 INIT	43185 EXEC	43233 PR#
43144 LOAD	43189 WRITE	43236 IN#
43148 SAVE	43194 POSITION	43239 MAXFILES
43152 RUN	43202 OPEN	43247 FP
43155 CHAIN	43206 APPEND	43249 INT
43160 DELETE	43212 RENAME	43252 BSAVE
43166 LOCK	43218 CATALOG	43257 BLOAD
43170 UNLOCK	43225 MON	43262 BRUN
43176 CLOSE	43228 NOMON	43266 VERIFY
43181 READ		

(43271 = end of command table)

The ASCII value of the last letter of each command is greater than 128; the other letters are always less. DOS is able to find its way through the command table by scanning each character to see if 128 has been added to it. For example, to change "CATALOG" to "CAT", you would add 128 to the "T" and then "close up" the table to write over the "ALOG" portion:

```
POKE 43220,PEEK(43220)+128
FOR I=43225 TO 43271 : POKE I-4,PEEK(I) : NEXT
```

Error message table: (read down)

43380 LANGUAGE NOT AVAILABLE	43477 DISK FULL
43402 RANGE ERROR	43486 FILE LOCKED
43413 WRITE PROTECTED	43497 SYNTAX ERROR
43428 END OF DATA	43509 NO BUFFERS AVAILABLE
43439 FILE NOT FOUND	43530 FILE TYPE MISMATCH
43453 VOLUME MISMATCH	43548 PROGRAM TOO LARGE
43468 I/O ERROR	43564 NOT DIRECT COMMAND

DOS finds error messages a little differently than the way it finds commands. It *always* goes to the locations listed above. Practically speaking, this means that you cannot replace an error

message with one that is longer than the original. It also means that you don't have to worry about closing up the table over an unused portion of an error message: DOS simply stops when it reaches a letter whose ASCII value is greater than 128. Here's how you would change "WRITE PROTECTED" to "CUT A NOTCH!":

```
MSG$="CUT A NOTCH!"
FOR I=1 TO LEN(MSG$)
  POKE 43412+I,ASC(MID$(MSG$,I,1))
NEXT
B$=RIGHT$(MSG$,1)
POKE 43412+LEN(MSG$),ASC(B$)+128
```

Greeting Program: POKE 40514

Sometimes it is desirable to be able to boot a disk that uses a binary (machine code) or a text (EXEC file) greeting program. While this can be done by booting up an Applesoft program which BRUNs a binary file or EXECs a text file, this method is slow because it requires two programs to be loaded from the disk and executed. The following three steps will show you how to initialize a disk so that it directly boots a binary or text file:

Step 1:

```
POKE 40514,52 <-- Allow binary greeting program
POKE 40514,20 <-- Allow text file greeting program
```

Step 2:
Clear the memory with a NEW command, enter 10 REM, insert a blank disk and enter INIT HELLO (or whatever name you want to give the binary or text file).

Step 3:
DELETE HELLO (or whatever name you choose for the greeting program) and save your binary or text file on the disk under the same greeting name (HELLO in this example). FID can be used to transfer your greeting program from another disk to the one just initialized. Once you have done this, the disk will boot the binary or text file directly.

Last Loaded File

Starting address = PEEK(43634) + 256*PEEK(43635)
(binary only)
Length = PEEK(43616) + 256*PEEK(43617)
(binary only)
Name = FOR I=43637 TO 43666 : PRINT CHR$(PEEK(I));
: NEXT

MON Flags: POKE 43614

The PRINT CHR$(4)"NOMON I,O,C" command turns off the printing of all input, output and commands of a program being executed and "hides" what your program is doing. It has one drawback: the NOMON I,O,C command itself is printed out. This can be avoided by using the POKE equivalent of the command:

```
POKE 43614,Ø : REM NOMON I,O,C (no output)
POKE 43614,112 : REM MON I,O,C (output restored)
```

(I=32, O=16, C=64; just add up the numbers corresponding to the letters that you want to use.)

RWTS (Read or Write a Track/Sector)

Through a few POKEs and a CALL, it is simple to read a sector from a disk and put it in a fixed 256-byte buffer beginning at location 46267. The buffer can be examined with PEEKs, changed with POKEs and written to the disk again.

TYPE	=	45121 (Read = 1, Write = 2)
DRIVE	=	46584 (1 or 2)
TRACK	=	45975 (0-34)
SECTR	=	45976 (0-15)
RWTS	=	45111
ERR	=	72
BUFFER	=	46267-46522

Example: (drive 2, track 17, sector 15)

Read a sector	Write a sector
POKE TYPE,1 POKE DRIVE,2 POKE TRACK,17 POKE SECTR,15 CALL RWTS POKE TYPE,2 POKE ERR,Ø	POKE TYPE,2 POKE DRIVE,2 POKE TRACK,17 POKE SECTR,15 CALL RWTS POKE ERR,Ø

Error Handling

Instead of having your Apple crash when it encounters an error condition (such as when someone leaves a disk drive door open), the ONERR GOTO statement can be used to keep the program going. Once an error has been encountered and the ONERR GOTO is taken, PEEKing at location 222 will tell you what kind of error has occurred so that you can take appropriate action. The following table is a combination of both Applesoft and DOS error conditions:

Ø NEXT WITHOUT FOR	16*SYNTAX ERROR
1*LANGUAGE NOT AVAIL.	22 RETURN W/O GOSUB
2 RANGE ERROR	42 OUT OF DATA
3 RANGE ERROR (again)	53 ILLEGAL QUANTITY
4*WRITE PROTECTED	69 OVERFLOW
5*END OF DATA	77 OUT OF MEMORY
6*FILE NOT FOUND	90 UNDEFINED STATEMENT
7*VOLUME MISMATCH	107 BAD SUBSCRIPT
8*I/O ERROR	120 REDIMENSIONED ARRAY
9*DISK FULL	133 DIVISION BY ZERO
10*FILE LOCKED	163 TYPE MISMATCH
11*SYNTAX ERROR	176 STRING TOO LONG
12*NO BUFFERS AVAIL.	191 FORMULA TOO COMPLEX
13*FILE MISMATCH	224 UNDEFINED FUNCTION
14*PROGRAM TOO LARGE	254 BAD INPUT
15*NOT DIRECT COMMAND	255 CTRL C INTERRUPT

Note: The "" symbol refers to a DOS error message; the rest refer to Applesoft.)*

Game I/O

Reading Paddles/Joystick:

```
PØ = PDL(Ø)
P1 = PDL(1)
```

PDL will return a value ranging from 0 to 255. Each paddle corresponds to one axis of a joystick. It is interesting to note that it is possible to connect *four paddles* or *two joysticks* to the game port, provided that they are read directly from the "extra" locations, PEEK(49254) and PEEK(49255).

Reading Pushbuttons:

```
B1 = PEEK(49249)
B2 = PEEK(49250)
B3 = PEEK(49251)
```

The button readings will remain less than 128 until a button is pushed. It is interesting to note that the Apple is capable of reading three buttons. Also note that paddle numbering begins with zero while button numbering begins with one.

Hi-res Graphics

Shape Table Pointer: 232-233

These two locations point to where you have loaded a shape table. Sample use:

```
ADDR = 768
AH% = ADDR / 256 : AL% = ADDR - AH%*256
POKE 232,AL% : POKE 233,AH%
```

Select Hi-res Page: POKE 230

Selecting a page means being able to alter it in some way, regardless of whether it is displayed or not. It is possible, for example, to display page one while drawing a picture "invisibly" on page two through the use of location 230. Here are the POKEs:

```
POKE 230,32  : REM Hi-res page 1
POKE 230,64  : REM Hi-res page 2
POKE 230,128 : REM Hi-res page "3"
```

Selecting page three actually accesses a part of RAM above the hi-res pages. While this area cannot be displayed, it can hold a hi-res picture and be moved into an actual hi-res page for display purposes.

Clear Hi-res Page: CALL -3086

It is interesting to note that it is possible to clear a hi-res page to black without displaying it. By selecting the page desired (see previous paragraph) and CALLing -3086, the page will be cleared.

Display Page 1: CALL -3100

This command is a quick way to "flip" back to hi-res page one and examine the screen without destruction of data. Unfortunately, there is no equivalent command for page two.

Page Flipping:

Displaying the text and hi-res pages without clearing the screen can be accomplished by selecting the appropriate series of POKEs presented below:

```
TEXT -- POKE 49236,0 : POKE 49234,0 : POKE 49233,0
HGR1-- POKE 49239,0 : POKE 49236,0 : POKE 49235,0
       : POKE 49232,0
HGR1-- POKE 49239,0 : POKE 49236,0 : POKE 49234,0
       : POKE 49232,0 (full screen HGR1)
HGR2-- POKE 49239,0 : POKE 49237,0 : POKE 49234,0
       : POKE 49232,0
HGR2-- POKE 49239,0 : POKE 49237,0 : POKE 49235,0
       : POKE 49232,0 (mixed screen HGR2)
```

The above numbers are called *soft switches* because they are under software control. By POKEing them with any number (0 was used here), they perform certain special functions:

```
49232: Display graphics
49233: Display text
49234: Select full screen (text or graphics)
49235: Select mixed screen (graphics)
49236: Select page one (text or graphics)
49237: Select page two (text or graphics)
49239: Select hi-res graphics
```

It's helpful to note that the display switches (49232 and 49233) should be POKEd last to avoid revealing the screen as it changes.

Reading The Keyboard: CALL -756

The following method of reading the keyboard is quick, simple and does not generate any garbage. It is ideal for single keypress responses that do not need to be saved. Note that clearing the strobe with a POKE 49168 is not needed with this method:

```
100 PRINT "PRESS ANY KEY TO CONTINUE:";
    : CALL -756
```

The key pressed is not lost; however, it can be retrieved and used:

```
100 PRINT "PRESS A KEY:";  : CALL -756
110 CHAR$ = CHR$(PEEK(49152))
120 PRINT CHAR$
```

Reading the keyboard can be done in the middle of a running Applesoft program to see if the user wants to change the action. This is how it can be done by using the keyboard strobe directly:

```
100 HOME
110 VTAB 12 : HTAB 10
120 PRINT "PRESS A KEY:"
130 IF PEEK(49152) < 128 THEN 110
140 VTAB 12 : HTAB 23
150 PRINT CHR$(PEEK(49152))
160 GOTO 110
```

Move Memory

The following is a monitor move routine that is very fast, especially when compared to the speed of moving memory by PEEKing and POKEing. To use it, you must specify the beginning and ending addresses of your source block of memory and the beginning address of the destination where you want to move it. The following example moves hi-res page two to hi-res page one:

```
100 DEF FN HI(X) = INT(X/256)
    : DEF FN LO(X) = X - 256 * FN HI(X)
110
    :
120 SB = 16384 : REM Source beginning
    : SE = SB + 8191 : REM Source end
    : DB = 8192 : REM Destination beginning
130
    :
140 POKE 60,FN LO(SB)
    : POKE 61,FN HI(SB)
    : POKE 62,FN LO(SE)
    : POKE 63,FN HI(SE)
    : POKE 66,FN LO(DB)
    : POKE 67,FN HI(DB)
    : POKE 71,0     <-- Clears the "Y register"
    : POKE 58,44    <-- MOVE routine (lo byte)
    : POKE 59,254   <-- MOVE routine (hi byte)
    : CALL -327     <-- "Go" processor
```

Reset Control: POKE 1012,0

A very simple POKE 1012,0 will cause your Apple to reboot if you hit RESET. It will also undo itself (restore a normal RESET) after the Apple reboots.

Screen Control

Clear Screen

Clear to end of line: CALL -868
Clear to end of page: CALL -958
Clear entire screen : CALL -936

Cursor Moves

→	CALL -484
←	CALL -1008
up	CALL -998 (useful in EXEC files to keep the cursor from "creeping" down as each line executes)
down	CALL -992

Scroll

Scroll up: CALL -912

Sounds

Click Speaker

Merely PEEKing at location 49200 any way you can will cause the speaker to click. Do this enough times in a loop and you will get a buzz. Actual musical tones require the speed of machine code routines. Here's an example of how to produce sound from the speaker:

Example 1:

```
X = PEEK(49200) : REM Make one click
(barely audible)
```

Example 2:

```
FOR I = 1 TO 100 : X = PEEK(49200) : NEXT
: REM Make buzz
```

Example 3:

```
FOR I = 1 TO 100
  X = PEEK(49200) + PEEK(49200) + PEEK(49200)
NEXT : REM Lower pitched buzz
```

Example 4:

```
S = 49200
FOR I = 1 TO 100
 X = PEEK(S) + PEEK(S) + PEEK(S)
NEXT : REM Higher pitched buzz (Applesoft works
faster with variables)
```

Ring The Bell

CALL -1052

Summary

These are just a few of the tricks you can do with some of the built-in functions of the Apple.

By looking at the monitor listing for the Apple in the *Applesoft BASIC Reference Manual*, you can find your own POKEs and PEEKs. Just remember to convert the hex numbers into decimal and you're all set.

For Further Reading

All About Applesoft (Call-A.P.P.L.E., Kent, WA). In-depth analysis and tutorial of Applesoft BASIC, including many programs and utilities written by users.

Apple II Reference Manual, Christopher Espinosa (Apple Computer, Cupertino, CA, 1979). Official technical manual to the inner workings of the Apple II family of computers. A classic.

Apple II Monitors Peeled (Apple Computer, Cupertino, CA, 1981). Official guide to how to use the monitor routines at $F800 to $FFFF (-2048 to -1) and the page zero locations used by the monitor. Many practical examples and shortcuts. For the intermediate to advanced user.

Applesoft Basic Programming Reference Manual (Apple Computer, Cupertino, CA). Official user's manual to the language. Contains many useful charts.

What's Where in the Apple? (Micro Ink, Chelmsford, MA 1981). Atlas of every memory location within the Apple, listing their names and explaining what they do. Alphabetical and numerical listings.

CHAPTER 9

HOW TO USE AN ASSEMBLER

An *assembler* is a program that lets you write programs in a powerful language called *assembly language.*

Advantages and Disadvantages of Assembly Language

Assembly language programs have two main advantages: speed and customization. Assembly language programs run at an incredible speed that no higher level language (like Applesoft) can match. You can also do things with assembly language that are impossible to do with higher level languages.

There are also some disadvantages with assembly language. It is difficult to learn, there are many new concepts, and it requires a more intimate knowledge of how your computer works. Assembly language is also tedious and error prone. There is little error checking available and debugging can become a major time consumer.

A Comparison: Applesoft and Machine Code

Let's take a look at the amazing speed available by comparing an Applesoft program with a machine code equivalent. *Machine code* is the *assembled* or *compiled* version of your assembly language program. Essentially, the assembly program is a dressed up version of machine code, replacing numbers with easy-to-remember expressions.

Here's an Applesoft program that clears the screen by wiping each line from left to right with a small "pad":

```
100 FOR LINE = 1 TO 24
110   VTAB LINE
120   FOR CLMN = 1 TO 40
130     HTAB CLMN
140     IF CLMN > 1 THEN
          HTAB CLMN - 1
    :       PRINT " "; : REM CLEAR PREVIOUS
            INVERSE CURSOR
150     IF CLMN < 40 THEN
        INVERSE : REM IF NOT EOLN, SHOW
        INVERSE CURSOR
160 PRINT " ";
    : NORMAL
170 NEXT CLMN
180 NEXT LINE
```

RUN this program and you'll find that you can watch the cursor move across the screen, erasing characters as it goes.

Now, without clearing any memory, enter in the machine code equivalent of the above program with the following steps:

```
]CALL-151
*300:A9 00 85 06 A9 00 85 07 A5 06 85 25
     20 22 FC
*30F:A4 07 F0 06 88 A9 A0 91 28 C8 A9 20
     C0 27 90 02
*31F:A9 A0 91 28 2C 00 C0 30 15 A5 F1 20
     A8 FC E6 07
*32F:A5 07 C9 28 90 DA E6 06 A5 06 C9 18
     90 C7 2C 10 C0 60
*<CTRL C>
```

Note: To type this program in, do not type the ""; it is shown here to match what you will see on the screen. Type in everything else on the line just as you see it; type the 300 and the ":" and the numbers after it. If you see a space, put one in; if you don't, don't put one in. It's that simple. Press RETURN at the end of the line, just as you would with an Applesoft line. The last line returns you to Applesoft when you enter CTRL-C (don't type in the* < or > symbols) *and press RETURN.*

Now, let's give this program something to erase, so LIST the Applesoft program given previously (it should still be in memory) and then run the machine code program by entering: CALL 768. Zip! If you blinked your eyes, you missed it! If you don't believe it is working the same way as the Applesoft program, you can slow it down with Applesoft's SPEED command. LIST the Applesoft program again, enter SPEED=100 and CALL 768 again. You can now watch the pad move dutifully across the screen. If you want to stop the action, just press a key.

The other advantage of assembly language previously mentioned is customization. You can do things with it that you cannot do with higher level languages. It is almost impossible to read a RAM card from Applesoft, for example; only machine code will do. Higher level languages cannot keep up with higher baud rates available with modems. Unless you use machine code, you will lose information being sent over phone lines.

Choosing an Assembler

To begin using assembly language, you must first get an assembler. These can be purchased in nearly any large computer store that sells Apple software or can be purchased through mail order.

Some of the more popular assemblers available are:

Merlin ($64.95)

Southwestern Data Systems
10761 Woodside Avenue #E
Santee, CA 92071
(619) 562-3670

Macro assembler; powerful line editor, 70-column hi-res character generator; numerous utilities included; supported by the book, *Assembly Lines* by Roger Wagner.

Big Mac ($28.50)

Call -A.P.P.L.E.
(must be a member)
21246 68th Ave., S.
Kent, WA 98032
(206) 872-2245

Forerunner of **Merlin**; macro assembler; source listings are identical with **Merlin**; supported by monthly articles in *Call-A.P.P.L.E.*

ORCA/M ($99.95)

Hayden Software
600 Suffolk St.
Lowell, MA 01853

Probably the most powerful of all macro assemblers available for the Apple. Packed with features not available with any other assembler. Full screen editor. Excellent, high-quality documentation, although it is intended for the intermediate to expert user.

The Assembler ($69.95)

MicroSPARC Inc.
10 Lewis St.
Lincoln, MA 01773
(617) 259-9710

Powerful macro handling capabilities. A companion library of macro definitions, **Macrosoft**, is also available to allow the user to write in a BASIC-like language. (Both **The Assembler** and **Macrosoft** can be purchased together for $99.95)

S-C Macro Assembler ($80)

S-C Software
Box 280300
Dallas, TX 75228
(214) 324-2050

Macro assembler; supported by excellent monthly newsletter, *Apple Assembly Line*, also published by S-C.

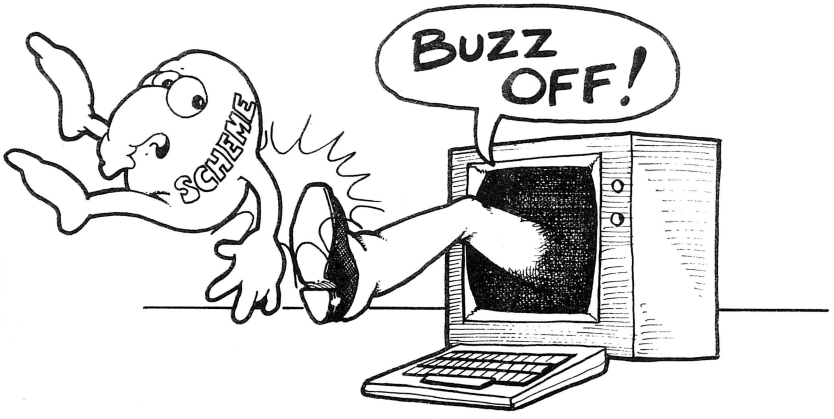
LISA ($119.95)

Sierra On-Line, Inc.
Coarsegold, CA 93614
(209) 683-6858

Extremely fast assembly times; supported by the book, *Using 6502 Assembly Language* by Randy Hyde (DATAMOST).

DOS Tool Kit

Apple Computer
20525 Mariani Ave.
Cupertino, CA 95014



Official assembler of Apple Computer, includes many useful utilities.

Due to the popularity, power and economy of the Big Mac/Merlin class of assemblers, the following examples were done with Merlin. Because of the simplicity of the examples, however, it should not be too difficult to adapt the exercises to other assemblers.

Getting Started

The first thing you must do after you get an assembler is *make a backup copy of it.* When you are working with assembly language, you are working at the very "nuts and bolts" level of the Apple, and mistakes are not tolerated very well by the system. At the very worst, you could accidentally write garbage on your disk, ruining information stored there. A backup copy will prevent the destruction of your original.

Try to make a backup copy using COPYA or one of the commercially available copy programs. If that fails, the manufacturer of the assembler is probably using some sort of copy protection scheme. Merlin uses such a scheme and requires you to boot the original and press the C key while the disk is booting to make a copy. You are limited to three copies and each copy made is a mule (won't reproduce).

Your First Assembly Language Program

Boot up your backup copy of Merlin and look at the final screen. You are presented with a menu of choices, such as C for CATALOG, D for DRIVE CHANGE (i.e., change from drive 1 to drive 2), E for enter the EDITOR, etc. The author of Merlin decided to call this the EXEC mode. We want to write a program, so let's press the E key to enter the editor.

You'll now see a : prompt symbol to tell you that you are in the editor. We want to add lines, so press the A key and RETURN to enter add mode. The number 1 now appears at the left edge of the screen, showing you the line number you are currently adding (creating).

An assembly language line is composed of four fields: Label, Op Code, Operand and Comment. Merlin is set to automatically move the cursor to the beginning of each field when you press the space bar or the right arrow key. Practice moving the cursor back and forth among the four fields with the right and left arrow keys until you get used to the way the cursor "jumps".

Let's try our hand at converting the Applesoft command to clear the screen, CALL -936, into an assembly language instruction. A CALL in Applesoft is the same as a JSR (Jump to Subroutine) in assembly language. We need to create a JSR -936 instruction, then. We do this by *hitting the space bar once* to make the cursor skip to the second field (there is no label in this instruction). We can now type JSR. *Hit the space bar to go to the next field.* Now type in -936 and press RETURN. Your instruction should look like this:

```
1       JSR -936
```

It should *not* look like these:

```
JSR       -936                    (forgot to skip the label field)
JSR-936                           (began in wrong field; no
                                  space after JSR)
          JSR-936                 (forgot to put a space after
                                  the JSR)
                    JSR -936      (began in wrong field)
```

If you make a mistake and you haven't finished the line by pressing RETURN, you can simply backspace and type over the error. If you have entered the line, press RETURN again to exit add mode and enter NEW. This will erase the program completely, just like NEW does in Applesoft. You can then start all over again with a clean slate. Later on, you'll be shown less drastic ways to correct a mistake.

Finally, we need to add a "finishing touch" to mark the end of the program, an RTS instruction. RTS means ReTurn from Subroutine and until you become more advanced, you must end every program with this instruction. (RTS is needed because assembly language programs are actually glorified subroutines; you must always return to the language that called (ran) the program). Press the space bar once to skip to the op code field, type RTS and press RETURN. Your program should now look like this:

```
1         JSR -936

2         RTS
```

That's it! Press the RETURN key to get out of add mode and to return to edit mode. You should see the : prompt symbol again. You can now instruct Merlin to assemble your program by typing the three letters ASM and pressing the RETURN key.

```
ASM <RETURN>
```

Merlin will now ask, Update source (Y/N)? For now, all you need to know is to always answer N to this question. Your program has now been assembled from the source code you typed in (the JSR and the RTS instructions) into object code (also called *machine code*). Because we didn't tell Merlin where to put the object code, it automatically put it at location $8000 (the "$" means the numbering system) or 32768 decimal. This means that we should be able to CALL 32768 and our little screen clearing program will run! *NOTE: Due to space constraints, the 48K version of Merlin automatically puts object code at* $5000 *(20480) instead of* $8000.

But first, we have to quit Merlin so that we can return to Applesoft to run the program. Press the Q key and then the RETURN key to quit the editor and then simply press Q again to quit Merlin. Go ahead and CALL 32768 and watch your first program work.

Enhancing Your Program

Now, let's return to our program and make some enhancements. If you are using the RAM card (64K) version of Merlin or Big Mac, all you have to do to return to the EXEC mode of Merlin or Big Mac is enter ASSEM. If you are using a non-RAM card (48K) version of Merlin or Big Mac, enter CALL 1016 to return to EXEC mode. (1016 is the address of the CTRL Y vector).

Press E to enter the editor again and enter L to list your program (it should still be there). Now let's add a line to the bottom of the program to turn off the listing of the cross-referenced symbol table at the end of an assembly. The symbol table is just a list of all the labels used in a program. Since we haven't used any labels, we don't need this listing. Turning off the listing doesn't hurt anything and will give us a shorter, clearer print out.

To add lines to the end of your program, enter "A" to enter add mode. You should now see "3", meaning that you're about to enter line number 3 in the program. Tap the space bar once to move the cursor to the op code field and type LST OFF (remember to put a space between the LST and the OFF). Hit RETURN at the end of this line, hit RETURN again to exit from add mode, then enter edit mode and enter L. Your program should look like this:

```
1           JSR -936
2           RTS
3           LST OFF
```

Now, enter ASM (and N to Update source?) and you'll see a much cleaner listing of your program. Notice also the object code listed on the left side of the screen:

```
8000: 20 58 FC 1          JSR -936
8003: 60       2          RTS
```

Object code is always expressed in the hexadecimal number system. The 8000 is the address where the object code is being assembled. The 20 is the machine code form of the assembly language instruction JSR. (Note that assembly language is not the same thing as machine code.) To see how -936 equals 58 FC, ask Merlin or Big Mac to convert -936 into hexadecimal for you by simply typing in -936 and pressing RETURN. You'll get the answer:

```
 -936 <-- You type this
$FC58 <-- Merlin gives you this
```

Do you see that by cutting the hex number in half, forming two bytes, and by switching the bytes that you arrive at the same number as in the object code listing?

FC58
(cut in half)

FC 58
(switch bytes)

58 FC

The three bytes we have examined, then, occupy memory ranges 8000-8002:

```
20          58          FC
↑           ↑           ↑
8000        8001        8002
```

This next instruction (60 or RTS will begin at 8003 and in fact, that's what the object code listing tells us:

```
8000: 20 58 FC 1          JSR -936
8003: 60       2          RTS
```

Not all programs begin at $8000. A lot begin at $300 or other places, so you must be able to tell Merlin where you want your program to assemble. This is done with the ORG and OBJ instructions. ORG means Origin address and OBJ means Object code address. It is not helpful to go into more detail on what these terms mean; they are quite complicated. For now, just follow the discussion and see what effect they have on your program.

Let's change our program so that it assembles at $300. This is a popular place to put programs and is also known as "page 3".

We must insert the ORG and OBJ instructions *Above* line 1, so we use the I command: I1 (and press RETURN). You will now see a new line 1 being displayed. Press the space bar and type in the following:

```
1          ORG $300
2          OBJ $300
3          (Hit the space bar once and hit RETURN
            to give yourself a blank line here.)
```

Note that we put a blank line at line 3 to make the printout look a little clearer. When you see line 4, don't type anything, just press RETURN and you will exit add mode and return to edit mode. List your program now and you should see this:

```
1          ORG $300
2          OBJ $300
3
4          JSR -936
5          RTS
6          LST OFF
```

ASM your program and you should see this:

```
                    1           ORG $300
                    2           OBJ $300
                    3
0300: 20 58 FC      4           JSR -936
0303: 60            5           RTS
```

Everything looks the same as before except that the addresses on the left edge of the listing now begin at the address specified, $300. *You should note that under normal conditions, Merlin and Big Mac will allow you to specify an OBJ only within page 3 or above $8000.* If you try to do this anyway, you will get a MEMORY FULL error. This is done to protect the rest of memory, which is used for your source program.

Editing Your Program

Let's learn how to correct typing errors in your program. First, let's put in some labels to give us something with which to practice. Insert the following lines above line 4 (use the command I4 and press RETURN). (Remember: these are labels so this time don't hit the space bar before typing them.)

```
4 LABEL1
5 LABEL2
6 LABEL3
```

List the program and you should see:

```
1           ORG $300
2           OBJ $300
3
4 LABEL1
5 LABEL2
6 LABEL3
7           JSR -936
8           RTS
9           LST OFF
```

Let's delete line 5. Simply enter D5. Your program will look like this (note how the assembler automatically renumbers the lines):

```
1           ORG $300
2           OBJ $300
3
4 LABEL1
5 LABEL3
6           JSR -936
7           RTS
8           LST OFF
```

Insert LABEL2 above line 5 again. Now, let's delete a range of lines with D4,5. Your program should look normal again (without the labels). Insert the labels back into the program again.

Now let's change all occurrences of the word, LABEL into CATS. C is used to indicate the change command, so enter C"LABEL "CATS" and press RETURN. Merlin will ask you: All or some (A/S)?. It wants to know if you want to change every occurrence of LABEL to CATS throughout the entire program. If you do, press the A key. If you want Merlin to stop at every thing it changes and ask you if you want to make the change permanent, enter S. If you choose the S option, you may accept the change by pressing Y or reject it by pressing the ESC key. You may terminate the changing at any time by entering CTRL C.

Now let's change the words CATS into other words on a line-by-line basis. To do this, we want to edit each line containing CATS, so enter E4,6. (You could also have done this by entering E"CATS"). Merlin will display each line and give you the opportunity to type over CATS with anything else you like.

If you wish to insert letters within a line, enter CTRL I (the left arrow key will get you out of insert mode). If you wish to delete letters within a line, enter CTRL D. Pressing CTRL N will move the cursor to the end of the line, CTRL B will return the cursor to the beginning. CTRL F will find (move the cursor to) the next character that you type.

When you are done editing the line, press RETURN. Note that you don't need to put the cursor at the end of the line before hitting RETURN. All visible characters will be saved. You can cut off characters after the cursor, however, by entering CTRL Q (for quit) instead of pressing RETURN.

Inside the 6502

The brain of your Apple is the small black chip known as the "6502." You can see it if you take the lid off your Apple and look at the green board (the motherboard) inside. The 6502 chip is marked as such on top and on the motherboard with small white digits.

The 6502 has three storage registers that can hold any number from 0 to 255. Each register has a name: A (the accumulator), X and Y. The essence of assembly language programming is to load one of these registers with a number, manipulate it somehow in the register and then store it back out to main memory.

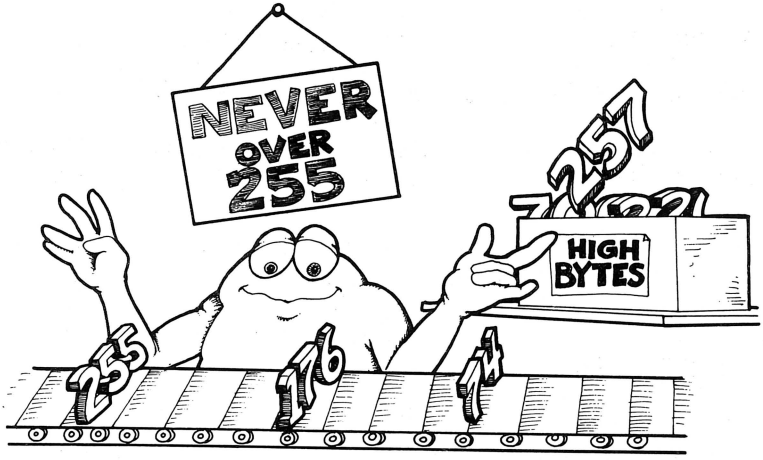
Let's take a simple example. Let's load the accumulator with the number 1 and store it in memory. From edit mode, enter NEW to erase any program in memory, enter "A" to enter add mode and enter the following program:

```
1       LDA #1
2       STA 6
3       RTS
4       LST OFF
```

Note the # symbol in line 1 and the lack of a # symbol in line 2. The # symbol is how you tell the assembler to get the actual number instead of the contents of an address.

For example, assume I have a shoe box. I take a nice, black felt-tip pen and write a 6 on the outside. This is now box number 6. I then take a sheet of paper and write a 1 on it and put it in the box. We now have the number 1 (#1) inside box number 6 (6).

Suppose I have another box that I have labeled "1." If I want to take the contents of box 1 out and put it in box 6, I would say LDA 1 and not LDA #1. So then the # would refer to the piece of paper while a number without the # symbol would refer to a box that could hold the piece of paper. In a similar manner we can refer to the loading of shoe boxes in assembly language, except that we call the shoeboxes a more computer-like term, "bytes."



ASM the above program, quit Merlin and CALL 32768 to run it. While it looks like nothing has happened, you must remember that the program wasn't supposed to do much either. Enter PRINT PEEK(6) to make sure that a 1 has indeed been stored in location (byte) number 6. Return to Merlin by entering ASSEM.

Loading Big Numbers: High and Low Bytes

Now suppose we have a number larger than 255. How do we load that into a register? Since we can't squeeze a large number into one register, we must split it up and put it in two registers! Thus, we must use two bytes to hold numbers greater than 255.

The number is split up by dividing it by 256. The quotient is called the "high byte," the remainder is called the "low byte."

Fortunately, we don't have to do any division; the assembler will do it for us if we use a special notation:

#<number	(this is the low byte of the number)
#>number	(this is the high byte of the number)

So, if we had a number like 260, we'd have:

#<260	(= 4 because 260/256 = 1 with a remainder of 4)
#>260	(= 1 because 260/256 = 1)

We could then load up two registers with this number:

LDA #<260	(this is the same as saying, LDA #4)
LDY #>260	(this is the same as saying, LDY #1)

Note: The Apple Toolkit assembler reverses this order, using > for the low byte and < for the high byte.

Using Labels

One other point before we can make a useful application. Just as in Applesoft, you can use variable names in assembly language to replace actual values:

For example, this

```
PRINT "HI"
```

can be replaced with

```
STRING$ = "HI"
PRINT STRING$
```

In assembly language, we can use *labels* to replace actual values also:

For example, this

```
8000: A9 04     1               LDA #<$8004
(04 is the low byte of $8004)

8002: A0 80     2               LDY #>$8004
(80 is the high byte of $8004)
```

```
8004: C8 C9     3               ASC "HI"
(C8="H", C9="I" in ASCII)
```

can be replaced with

```
8000: A9 04     1               LDA #<STRING
8002: A0 80     2               LDY #>STRING
8004: C8 C9     3 STRING        ASC "HI"
```

In the second example, you can see that Merlin is smart enough to know that STRING is a label that equals $8004. It is then able to treat STRING as a variable name of sorts and work with the number that it represents. Labels are used because it's easier to refer to a part of a program by a name than it is to try to figure out the address. Also note the use of the "pseudo" op code ASC to tell the assembler that "HI" is a string of ASCII characters to be inserted in the object code.

We can put all of this together to create a short assembly language program that will print out a string:

```
1               LDA #<STRING
2               LDY #>STRING
3               JSR $DB3A       PRINT OUT STRING
4               RTS             THIS IS THE END
                                OF THE PROGRAM
5 STRING        ASC "THIS IS A TEST"
6               HEX 00
7               LST OFF
```

Line 3 in this example uses a routine already inside Applesoft at address $DB3A. This routine is called STROUT and its purpose is to output a string. STROUT requires only that the low byte of the address of the string you want printed is in the A register (line 1), the high byte of the address of the string is in the Y register (line 2) and that there is a zero at the end of the string (line 9). Note the use of comments in lines 3 and 4 to help explain what the program does.

To make things a little clearer, we can create another label (lines 1 & 5):

```
1 STROUT   = $DB3A
2
3          LDA #<STRING
4          LDY #>STRING
5          JSR STROUT      PRINT OUT STRING
6          RTS
7 STRING   ASC "THIS IS A TEST"
8          LST OFF
```

Note that you may have seen line 1 expressed as STROUT EQU $DB3A in other assemblers. Either form may be used on Merlin; the = sign is used here for clarity.

Control Structures In Assembly Language

You'll recall from the chapter on structured programming that any program in any language can be written with only three control structures: sequence, decision and loop. It's time now to see how to create these structures with 6502 assembly language.

Sequence

Sequence is the easiest structure to show; all of the examples used so far are examples of sequential programming. Coding that does not involve branching to different areas of the program is sequential. Some examples:

```
LDA #1
LDX #2
LDY #3
STA 1
STX 2
STY 3
```

Here we load up each of the 6502's registers and store them back out again in locations 1,2 and 3.

Decision

A decision is usually made in 6502 programming by comparing two values and then branching, depending on the results. For example, we can take the trivial example of loading X with the contents of location 36 and seeing if it was a 2. If it is, we'll branch to some other part of the program:

```
        LDX 36
        CPX #2          (This means compare the contents
                         of X with 2)
        BEQ PART2       (Branch if X equals 2)
PART1   RTS             (Quit if X does not equal 2)
PART2   LDY #4          (Do this if X does equal 2)
        RTS
```

Loop

Looping is also done with a comparison and a branch, but the branch taken is to the beginning or end of the loop. For example, let's look at a loop that loads X with Ø and then adds 1 to X until X = 1Ø. (This is the same thing as saying FOR X = 1 TO 1Ø in Applesoft and is also known as a "trailing decision" loop):

```
        LDX #Ø          (Make sure X reg contains zero)
LOOP    INX             (Increment X: add 1 to X)
        CPX #11         (Have we gone too far?)
        BNE LOOP        (No, keep going)
        RTS             (Yes, quit)
```

A WHILE WEND type of loop with a leading decision that performs the loop only if X is less than 11 looks like this:

```
        LDX #Ø          (Initialize X)

LOOP    CPX #11
        BEQ DONE
        INX
        JMP LOOP        ("Jump" back and do the loop
                         again)
DONE    RTS
```

Basic Techniques

How to Convert 1-Byte PEEKs to Assembly Language:

```
LDA NUMBER      ;A = PEEK(NUMBER)
```

How to Convert 2-Byte PEEKs to Assembly Language:

```
LDA NUMBER      ; A% = PEEK(NUMBER)/256
LDX NUMBER+1    ; X% = NUMBER - A%*256
```

How to Convert POKEs to Assembly Language:

```
LDA NUMBER      ;POKE ADDRESS,NUMBER
STA ADDRESS
```

How to Convert CALLs to Assembly Language:

```
JSR ADDRESS     ;CALL ADDRESS
```

How to Read a Keypress:

```
LOOP    LDA $C000       Read the keyboard
        BPL LOOP
        BIT $C010       Clear the keyboard strobe
```

Note: The A register will contain the character with the high bit on when the loop is exited.

How to Read a Line, Commas And All:

```
GETLN5 = $FD6F
       JSR GETLN5
```

Note: Upon exit, the string will be stored in the input buffer ($200 or 512) and the length of the string will be in the X register. When you are done with your program, use JMP $E003 (return directly to Applesoft) instead of an RTS to prevent your Apple from trying to execute the string.

How to Print a Number (0 - 65535) in Decimal:

```
LINPRT =    $ED24
        LDX #<NUMBER
        LDA #>NUMBER
        JSR LINPRT
```

How to Print a Decimal Number in Hex:

```
PRNTAX =    $F941
        LDX #<NUMBER
        LDA #>NUMBER
        JSR PRNTAX Print A & X regs in hex
```

How to Print a String:

```
STROUT =    $DB3A
        LDA #<STRING
        LDY #>STRING
        JSR STROUT
        RTS
STRING ASC "THIS IS A TEST"
        HEX 00
```

How to Add Two 1-Byte Numbers:

```
NUMBER1 = 6     lo byte first
NUMBER2 = 7     lo byte first

RESULTS   = 8
          LDA NUMBER1
          CLC          Make sure carry is clear
          ADC NUMBER2
          STA RESULTS
```

How to Add Two 2-Byte Numbers:

```
NUMBER1 = 6              lo byte first
NUMBER2 = 8              lo byte first
RESULTS  =10
         LDA NUMBER1
         CLC             Make sure carry is clear
         ADC NUMBER2
         STA RESULTS
         LDA NUMBER1+1
         ADC NUMBER2+1
         STA RESULTS+1
```

How to Subtract Two 1-Byte Numbers:

```
NUMBER1 = 6              Lo byte first
NUMBER2 = 7              Lo byte first
RESULTS  =8
         LDA NUMBER1
         SEC             Must be done before a
                         SBC inst.
         SBC NUMBER2
         STA RESULTS
```

How to Subtract Two 2-Byte Numbers:

```
NUMBER1 = 6              Lo byte first
NUMBER2 = 8              Lo byte first
RESULTS  =10
         LDA NUMBER1
         SEC
         SBC NUMBER2
         STA RESULTS
         LDA NUMBER1+1
         SBC NUMBER2+1
         STA RESULTS+1
```

Sample Applications

Print the Beginning and End of an Applesoft Program in Memory:

```
TXTTAB  = 103
STREND  = 109
COUT    = $FDED
LINPRT  = $ED24
          LDX TXTTAB          Beginning of program pointer
          LDA TXTTAB+1
          JSR LINPRT
          LDA #" "            Put a space between the
                              answers
          JSR COUT
          LDX STREND          End of program pointer
          LDA STREND+1
          JSR LINPRT
          RTS
```

CATALOG the Disk:

```
BASIC   = $E003
STROUT  = $DB3A
          LDA #<STRING
          LDY #>STRING
          JSR STROUT
          JMP BASIC
STRING    HEX 84            CTRL D
          ASC "CATALOG"
          HEX 00            Used by STROUT to mark
                            end of string
```

Change Disk Drive Number (2 to 1 or 1 to 2):

```
          LDA #3
          SEC
          SBC $AA68
          STA $AA68
```

Print Starting Address and Length of Last Binary File In Hex:

```
PRNTAX    = $F944
COUT      = $FDED
          LDX $AA72
          LDA $AA73
          JSR PRNTAX
          LDA #" "
          JSR COUT
          LDX $AA60
          LDA $AA61
          JMP PRNTAX
```

Print Name of Last File Loaded or Run (any type!):

```
COUT      = $FDED
          LDX #0
LOOP      LDA $AA75,X
          JSR COUT
          INX
          CPX #30
          BNE LOOP
          RTS
```

Print Name of Disk:

```
COUT      = $FDED
          LDX #10
LOOP      LDA 46000,X
          JSR COUT
          DEX
          CPX #$FF
          BNE LOOP
          RTS
```

DOS/Applesoft Problem

There is a compatibility problem between DOS and Applesoft when you attempt to BRUN a program that uses COUT to print characters, as many of the programs above do. BRUNning may cause your Apple to hang. If this occurs, you must change your program so that the very first thing it does is save the contents of DOS location $AA59. Then, just before you exit the program, you must restore $AA59. Here's an example:

```
        LDA $AA59       Grab this byte
        STA $FE         and save it in free space
COUT    = $FDED
        LDX #10
LOOP    LDA 46000,X
        JSR COUT        Print name of last file loaded
        DEX
        CPX #$FF
        BNE LOOP
        LDA $FE         Retrieve the byte
        STA $AA59       and restore it again
        RTS
```

Summary

This chapter is a *very* brief introduction to **assembly language** and the use of an **assembler**. Both the language and the assembler are capable of far more than what you've seen here and many tricks and shortcuts could have been taken, but the examples were presented with the beginning assembly language programmer in mind. To really learn assembly language programming, it is suggested that you purchase one of the books intended to accompany the matching assembler and listed previously under the section entitled, "Choosing An Assembler."

An **assembler** is a commercially prepared program that allows you to write assembly language programs. Without an assembler, you cannot write in assembly language.

Assembly language gives you speed and the ability to do things that are difficult to do from higher level languages. It suffers from the disadvantage of being tedious and requiring time to debug faulty programs.

An assembly language instruction consists of four fields: **Label, Op Code, Operand** and **Comment.** An assembly language program is also known as a source file or source program. It assembles or compiles into object code or machine code.

In comparing Applesoft commands to assembly language instructions, we find that CALLs are the same as JSRs, PEEKs are the same as LDAs, POKEs are the same as STAs and variable names are similar to labels.

ORG determines at which address the source listing will be assembled *without physically placing the object code anywhere,* while OBJ controls where the object code will be physically placed in memory.

The "brain" of the Apple is the **6502 chip.** It has three main registers in it (A, X and Y) that are used by assembly language programmers. Each register can hold a number from 0 to 255. Characters can be stored in the registers in the form of ASCII numbers.

The Apple cannot store numbers greater than 255 into a single memory location or byte. Numbers greater than 255 can be broken up into two bytes, however, by dividing the number by 256. The quotient is called the **high byte,** the remainder is called the **low byte.**

Pseudo op codes like ASC, EQU, = and HEX do not mean anything to the 6502 chip, but instead are used to tell the assembler to perform a function that would be difficult or laborious to do by hand.

Now you have had enough of an introduction to assembly programming to decide whether or not it is your cup of tea. There is a good deal you can do with Applesoft, but to really control your computer, learning assembly/machine code programming is necessary.

For Further Reading

Also look at the list of books in the "**Choosing an Assembler**" section in this chapter.

6502 Assembly Language Programming, Lance Leventhal (Osborne/McGraw-Hill, Berkeley, CA, 1979). No-nonsense, straightforward approach to 6502 assembly language programming. Complete, detailed and accurate. Numerous and detailed charts and diagrams. Sample applications on a wide variety of topics. For 6502 microprocessors in general. A classic.

An Introduction to Microcomputers — Volume 0: The Beginner's Book, Adam Osborne (Osborne & Associates, Berkeley, CA, 1979). Detailed, yet simple examination of how microprocessors work. Numerous illustrations.

An Introduction to Microcomputers — Volume 1: Basic Concepts, Adam Osborne (Osborne & Associates, Berkeley, CA, 1976). A continuation of Volume 0. Goes into great depth on how the machine code instructions work.

Apple Machine Language Don and Kurt Inman (Reward Books, Reston, VA, 1981). A true beginner's introduction to machine code programming. Very user friendly. The emphasis is on machine code and not assembly language.

Microelectronics (Scientific American, 1977). Excellent explanation of how the hardware in personal computers works. Numerous large, full color photographs. A reprint of articles from Scientific American.

Understanding Microprocessors, Don Cannon and Gerald Luecke (Texas Instruments Learning Center, Dallas, TX, 1979). An excellent, illustrated introduction to how the hardware of microprocessors works. Beginning to intermediate level.

CHAPTER 10

PROGRAM DEVELOPMENT AIDS

This program will show you how to use some utility programs that will make your programming in Applesoft more manageable. We will take a look at:

- RDLN, an assembly language routine you can type in that allows you to input strings containing commas.
- ASA, or Ampersand Structured Applesoft (enclosed herein) that gives you a multi-line IF THEN ELSE, REPEAT UNTIL, and WHILE WEND control structures.
- An introduction to how to use APLUS, a structured Applesoft pre-processor and other important aids from Sensible Software.
- An introduction to BASIC', a powerful structured version to Applesoft, from Delta Micro Systems.

RDLN

RDLN, pronounced Read Line, is a short routine that allows you to input strings containing commas, colons, semicolons and other characters that normally return an ?EXTRA IGNORED error message from Applesoft. This utility was mentioned in Chapter 5 (Text Files), where it was explained that it could be used to read files containing these characters.

To create this program, CALL -151 to enter the monitor and type in each line of the object code listed on the left edge of the following printout (disassembly) of the program. For example, the first line looks like this:

```
0300: 20 B1 00   233     JSR CHRGET    Skip ":" separator
(--Type this--)  ( ---------- Ignore this ----------)
(machine code)             (assembly language)
```

You would then type in:

```
0300: 20 B1 00    (and press the RETURN key)
```

Note that you should type exactly what you see: if you see a blank, put one in; if you don't, don't put one in. It's that simple.

The first few lines would look like this (remember to press RETURN after each line):

```
0300: 20 B1 00
0303: 20 E3 DF
0306: 20 6C DD
```

When you finish typing in the program, enter <CTRL C> <RETURN> to return to Applesoft and enter BSAVE RDLN, A$300,L45. This will save the program on disk. The assembly language source file listed below gives you an example of how to use it.

Here is the complete program listing for you to use (Merlin was the assembler used):

```
                        1 ***************************************************
                        2 *     RDLN (Read Line) by Bill Parker, 7/29/83    *
                        3 *  Adapted from Apple Assembly Line, 12/83, p 17  *
                        4 *    This program allows the entry of a string of *
                        5 *   characters, commas and all, from Applesoft.   *
                        6 *  Example:                                       *
                        7 *     PRINT CHR$(4);"BLOAD RDLIN,A768"            *
                        8 *     RDLN = 768                                  *
                        9 *     CALL RDLN:REC$ : REM Input a line           *
                       10 ***************************************************
                       11
                       12              ORG $300   Note: This is RELOCATABLE
                       13
                       14 * Page 0 locations
                       15 CHRGET     =   $B1
                       16 FRESPC     =   $71
                       17 VARPNT     =   $83
                       18 *
                       19 * Input Buffer
                       20 INBUFR     =   $200
                       21 *
                       22 * Applesoft Routines
                       23 CHKSTR     =   $DD6C
                       24 GETSPA     =   $E452
                       25 INLIN      =   $D52C
                       26 MOVSTR     =   $E5E2
                       27 PTRGET     =   $DFE3
                       28
                       29 *
                       30 * Parse RDLN command
                       31 *
0300: 20 B1 00         32          JSR CHRGET     Skip ":" separator
0303: 20 E3 DF         33          JSR PTRGET     Find or creat var. descriptor
0306: 20 6C DD         34          JSR CHKSTR     But make sure var is string
                       35 *
                       36 * INput a string into the variable chosen
                       37 *
3039: 20 2C D5         38          JSR INLIN      Read a line from input device
                       39 *
```

```
                        40 * Compute string lenth & make room for it in memory
                        41 *
030C: E8                42 LOOP     INX                 ;Always starts
                                                        at $FF
030D: BD 00 02          43          LDA  INBUFR,X       Got a "0"
                                                        (EOL marker)
0310: D0 FA             44          BNE  LOOP           ;No, Keep
                                                        checking
0312: 8A                45          TXA                 ;Yes, get ready
                                                        to make room
0313: 48                46          PHA                 ;Save string
                                                        length
0314: 20 52 E4          47          JSR  GETSPA         Make room in
                                                        string area
                        49 *
                        50 * Plug in length and pointer to string in
                        51 * variable descriptor
                        52 *
0317: A0 00             53          LDY  #0
0319: 91 83             54          STA  (VARPNT),Y     Length of
                                                        string
031B: C8                55          INY
031C: 15 71             56          LDA  FRESPC         Addr (lo) of
                                                        storage
031E: 91 83             57          STA  (VARPNT),Y
0320: C8                58          INY
0321: A5 72             59          LDA  FRESPC+1       Addr (hi) of
                                                        storage
0323: 91 83             60          STA  (VARPNT),Y
                        61 *
                        62 * Move string from input buffer to string storage area
                        63 *
0325: A2 00             64          LDX  #<INBUFR
0327: A0 02             65          LDY  #>INBUFR
0329: 68                66          PLA                 ;Retrieve
                                                        string length
032A: 4C E2 E5          67          JMP  MOVSTR         (Uses A,X,Y as
                                                        shown above)
                        68          LST  OFF            Don't print
                                                        symbol table


--End assembly-


45 bytes


Errors: 0
```

ASA

ASA, or Ampersand Structured Applesoft, is a "poor man's" structured Applesoft. You can have it for the price of this book; simply type it in from the program listed below (use the instructions given previously in RDLN if you need help).

ASA gives you a multi-line IF THEN ELSE structure, a repeat until loop and a while wend loop. The structures are implemented by using the & character as shown below. Also, to keep the program as simple and as short as possible, Applesoft tokens were used instead of the words normally associated with these structures:

ASA	Usual Usage

& IF <condition> THEN
  <statements>
& ELSE
  <statements>
& END
	IF <condition> THEN
	  <statements>
	ELSE
	  <statements>
	IFEND

& RUN <statements> & STOP IF <condition>	REPEAT <statements> UNTIL <condition>
& ON <condition> <statements> & CONT	WHILE <condition> <statements> WEND

```
1 ************************************************
2 * ASA - Ampersand Structured Applesoft 6/11/83 *
3 *            Version 1.0 - By Bill Parker      *
4 *                                              *
5 *      Purpose: Allows the use of structured   *
6 *   programming in Applesoft through the use of *
7 *            three control structures:         *
8 *    1. A true multi-line IF THEN ELSE:        *
9 *       & IF condition THEN                    *
10 *        <stmts>                               *
11 *      & ELSE                                  *
12 *        <stmts>                               *
13 *      END                                     *
```

```
14 *      2. A REPEAT UNTIL loop:                          *
15 *         & RUN                                         *
16 *              <stmts>                                  *
17 *         & STOP IF condition                           *
18 *      3. A WHILE DO loop:                              *
19 *         & ON condition                                *
20 *              <stmts>                                  *
21 *         & CONT                                        *
22 *     This program is RELOCATABLE. Uses two pg.         *
23 *  Ø locations: IFCTR=25Ø, ONCTR=251. Features          *
24 *         a self-initializing ampersand hookup.         *
25 *           To create: Type in object code and          *
26 *                BSAVE ASA, A$8ØØØ, L43Ø                *
27 *      To start: Simply BRUN ASA anywhere you           *
28 *                      have room.                       *
29 ***************************************************
30
31 * Definitions
32    AMPERTKN   =  175
33    CONTTKN    =  187
34    ENDTKN     =  128
35    IFTKN      =  173
36    ONTKN      =  18Ø
37    RUNTKN     =  172
38    STOPTKN    =  179
39
40 * Page zero locations
41    CHRGET     =  $B1
42    CURLIN     =  $75
43    FAC        =  $9D
44    IFCTR      =  25Ø
45    ONCTR      =  251
46    TXTPTR     =  $B8
47
48 * Page 3 locations
49    AMPERVEC   =  $3F5
50
51 * Applesoft routines
52    ADDON      =  $D998
53    FRMEVL     =  $DD7B
54    NESTERR    =  $DDØB
55    NEWSTT     =  $D7D2
56    PRSYNERR   =  $DEC9
57    SYNCHR     =  $DECØ
58
59 * Monitor routines
6Ø    RETURN     =  $FF58
61
```

```
                        62 *================================*
                        63 *  BEG OF PROGRAM -- This is a "header" section *
                        64 *  that clears the IF and ON counters (which keep  *
                        65 *    track of loop nesting levels) and initializes the    *
                        66 *  ampersand vector no matter where this program is  *
                        67 *                    BRUNed.                     *
                        68 *================================*
                        69
8000: A9 00             70              LDA  #0          Clear &IF and
                                                         &ON counters
8002: 85 FA             71              STA  IFCTR
8004: 85 FB             72              STA  ONCTR
8006: 20 58 FF          73  SETAMPER     JSR  RETURN      Put current loc
                                                         in stack
8009: BA                74              TSX              ;Get stack
                                                         pointer
800A: BC 00 01          75              LDY  $100,X      Get loc hi
800D: CA                76              DEX              ;Move  stack
                                                         ptr up
800E: BD 00 01          77              LDA  $100,X      Get loc lo
8011: 18                78              CLC
8012: 69 16             79              ADC              #CK4IF-
                                                         SETAMPER-2
8014: 90 01             80              BCC  :1          Lo byte > $FF?
8016: C8                81              INY              ;Yes-inc
                                                         hi-byte
8017: 8D F6 03          82  :1          STA  AMPERVEC+1  Put BOP (lo) in
                                                         & vector
801A: 8C F7 03          83              STY  AMPERVEC+2  Put  BOP  (hi)
                                                         in & vector
801D: 60                84              RTS              ;Return to
                                                         Applesoft
                        85 *================================*
                        86


                        87 *================================*
                        88 *     Begin parsing for &IF THEN, &ELSE, &END,    *
                        89 *           &RUN, &STOP, &ON, &CONT               *
                        90 *================================*
                        91
801E: C9 AD             92  CK4IF       CMP  #IFTKN      Got IF token?
8020: F0 1F             93              BEQ  IF
8022: C9 45             94  CK4ELSE     CMP  #'E'        Got ELSE?
8024: D0 15             95              BNE  CK4END
8026: 20 B1 00          96              JSR  CHRGET
8029: C9 4C             97              CMP  #'L'
802B: D0 0E             98              BNE  CK4END
802D: 20 B1 00          99              JSR  CHRGET
8030: C9 53            100              CMP  #'S'
8032: D0 07            101              BNE  CK4END
```

```
8034: 20 B1 00  102              JSR  CHRGET
8037: C9 45     103              CMP  #'E'
8039: F0 54     104              BEQ  ELSE
803B: C9 80     105 CK4END       CMP  #ENDTKN    Got END
                                                 token?
803D: F0 6A     106              BEQ  END
803F: D0 6B     107              BNE  CK4RUN     Check for
                                                 other tokens
                108 *
                109 * Process & IF command
                110 *
8041: 20 B1 00  111 IF           JSR  CHRGET     Get off IF token

8044: 20 7B DD  112              JSR  FRMEVL     Evaluate cond
                                                 & put in FAC
8047: A5 9D     113 CK4TRUE      LDA  FAC        Eval. to Ø?
8049: F0 03     114              BEQ  FALSE      Yes, condition
                                                 is false
804B: 4C B1 00  115 TRUE         JMP  CHRGET     No, skip THEN
                                                 & resume
804E: 20 B1 00  116 FALSE        JSR  CHRGET     Skip until &
                                                 token
8051: C9 AF     117              CMP  #AMPERTKN  is found
8053: D0 F9     118              BNE  FALSE
8055: 20 B1 00  119              JSR  CHRGET     Look at char
                                                 after &
8058: C9 AD     120              CMP  #IFTKN     Is it IF?
805A: F0 1F     121              BEQ  IF1        Yes
805C: C9 45     122 CK4ELSE1     CMP  #'E'       Is it ELSE?
805E: D0 15     123              BNE  CK4END1
8060: 20 B1 00  124              JSR  CHRGET
8063: C9 4C     125              CMP  #'L'
8065: D0 0E     126              BNE  CK4END1
8067: 20 B1 00  127              JSR  CHRGET
806A: C9 53     128              CMP  #'S'
806C: D0 07     129              BNE  CK4END1
806E: 20 B1 00  130              JSR  CHRGET
8071: C9 45     131              CMP  #'E'
8073: F0 13     132              BEQ  ELSE1
8075: C9 80     133 CK4END1      CMP  #ENDTKN    Is it END
                                                 token?
8077: F0 06     134              BEQ  END1
8079: D0 D3     135              BNE  FALSE      Not correct &
                                                 command
807B: E6 FA     136 IF1          INC  IFCTR
807D: D0 CF     137              BNE  FALSE
807F: A5 FA     138 END1         LDA  IFCTR
8081: F0 09     139              BEQ  SKIP1      Correct nest
                                                 end?
8083: C6 FA     140              DEC  IFCTR      No, keep going
8085: B8        141              CLV
```

```
8086: 50 C6     142              BVC  FALSE      Branch always
8088: A5 FA     143 ELSE1        LDA  IFCTR      Correct nest?
808A: D0 C2     144              BNE  FALSE      No, keep going
808C: 4C B1 00  145 SKIP1        JMP  CHRGET     Got to end OK;
                                                 return
                146 *
                147 * Process & ELSE command
                148 *
808F: 20 B1 00  149 ELSE         JSR  CHRGET     Get off ELSE
8092: C9 AD     150              CMP  #IFTKN
8094: F0 06     151              BEQ  IF2
8096: C9 80     152              CMP  #ENDTKN
8098: F0 06     153              BEQ  END2
809A: D0 F3     154              BNE  ELSE       Skip to correct
                                                 nest
809C: E6 FA     155 IF2          INC  IFCTR
809E: D0 EF     156              BNE  ELSE
80A0: A5 FA     157 END2         LDA  IFCTR
80A2: F0 05     158              BEQ  END
80A4: C6 FA     159              DEC  IFCTR
80A6: B8        160              CLV
80A7: 50 E6     161              BVC  ELSE
80A9: 4C B1 00  162 END          JMP  CHRGET
                163
                164 *============================*
                165 * CHECK FOR &RUN/STOP (REPEAT UNTIL) LOOP
                166 *
80AC: C9 AC     167 CK4RUN       CMP  #RUNTKN
80AE: F0 06     168              BEQ  AMPRRUN
80B0: C9 B3     169              CMP  #STOPTKN
80B2: F0 22     170              BEQ  AMPRSTOP
80B4: D0 68     171              BNE  CK4ON      Not RUN or
                                                 STOP, keep
                                                 chking
                172 *
                173 * Process & RUN command
                174 *
80B6: 20 B1 00  175 AMPRRUN      JSR  CHRGET     Get off RUN
                                                 token
80B9: F0 03     176              BEQ  *+5        End of stmt?
80BB: 4C C9 DE  177              JMP  PRSYNERR   No-bad
                                                 &RUN cmd
80BE: 68        178              PLA             Save current
                                                 rtn addr
80BF: AA        179              TAX
80C0: 68        180              PLA
80C1: A8        181              TAY
80C2: A5 76     182              LDA  CURLIN+1   Save line no of
                                                 current
80C4: 48        183              PHA             ;line being
                                                 executed
```

```
80C5: A5 75     184              LDA  CURLIN
80C7: 48        185              PHA
80C8: A5 B9     186              LDA  TXTPTR+1    Save current position
80CA: 48        187              PHA              ;of where we are in the
80CB: A5 B8     188              LDA  TXTPTR      ;Applesoft program
80CD: 48        189              PHA
80CE: A9 AC     190              LDA  #RUNTKN     Save RUN token as an
80D0: 48        191              PHA              ;identification byte
80D1: 98        192              TYA              ;Return orig rtn addr
80D2: 48        193              PHA              ;on top of all
80D3: 8A        194              TXA              ;these values
80D4: 48        195              PHA
80D5: 60        196              RTS              ;Return to Applesoft prog
                197 *
                198 * Process & STOP command
                199 *
80D6: 20 B1 00  200 AMPRSTOP     JSR  CHRGET      Get off STOP token
80D9: A9 AD     201              LDA  #IFTKN      On an IF token now?
80DB: 20 C0 DE  202              JSR  SYNCHR      Err msg if not
80DE: 20 7B DD  203              JSR  FRMEVL      Eval condition after IF
80E1: A5 9D     204              LDA  FAC         (cond stored in FAC)
80E3: D0 25     205              BNE  EXITLOOP
80E5: BA        206 LOOPAGIN     TSX
80E6: BD 03 01  207              LDA  $103,X      Check ID byte for RUN
80E9: C9 AC     208              CMP  #RUNTKN
80EB: F0 03     209              BEQ  *+5
80ED: 4C 0B DD  210              JMP  NESTERR
80F0: BD 04 01  211              LDA  $104,X      Restore previously saved
80F3: 85 B8     212              STA  TXTPTR      values from AMPRRUN
80F5: BD 05 01  213              LDA  $105,X
80F8: 85 B9     214              STA  TXTPTR+1
80FA: BD 06 01  215              LDA  $106,X
80FD: 85 75     216              STA  CURLIN
80FF: BD 07 01  217              LDA  $107,X
8102: 85 76     218              STA  CURLIN+1
8104: E8        219              INX
```

```
8105: E8        220             INX
8106: 9A        221             TXS
8107: 4C D2 D7  222             JMP  NEWSTT       Rtn to A/S
                                                  prog & resume
810A: BA        223 EXITLOOP    TSX               ;Time to quit
                                                  looping
810B: BD 03 01  224             LDA  $103,X       Make sure
                                                  loop is
810E: C9 AC     225             CMP  #RUNTKN      properly
                                                  nested
8110: F0 03     226             BEQ  *+5
8112: 4C 0B DD  227             JMP  NESTERR
8115: 8A        228             TXA               ;Drop stack
                                                  pointer down
8116: 18        229             CLC               ;to delete
                                                  previously
8117: 69 07     230             ADC  #7           ;saved values
                                                  under
                                                  AMPRRUN
8119: AA        231             TAX
811A: 9A        232             TXS
811B: 4C D2 D7  233             JMP  NEWSTT       Rtn to A/S
                                                  prog & resume
                234
                235 * ============================
                236 *CHECK FOR &ON/CONT (WHILE DO) LOOP
                237 *
811E: C9 B4     238 CK4ON       CMP  #ONTKN
8120: F0 07     239             BEQ  GotOn
8122: C9 BB     240             CMP  #CONTTKN
8124: F0 60     241             BEQ  GOTCONT
8126: 4C C9 DE  242 BUMCMD      JMP  PRSYNERR     Finally: not
                                                  valid & cmd
                243 *
                244 * Process ON/CONT loop
                245 *
8129: A5 B8     246 GotOn       LDA  TXTPTR       Back TXTPTR
                                                  up to
812B: 38        247             SEC               ;beg of stmt
812C: E9 06     248             SBC  #6
812E: 85 B8     249             STA  TXTPTR
8130: A5 B9     250             LDA  TXTPTR+1
8132: E9 00     251             SBC  #0
8134: 85 B9     252             STA  TXTPTR+1
8136: 68        253 SAVE2STK    PLA               ;PCL
8137: AA        254             TAX
8138: 68        255             PLA               ;PCH
8139: A8        256             TAY
813A: A5 76     257             LDA  CURLIN+1     Cur. A/S line
                                                  no.
813C: 48        258             PHA
```

```
813D: A5 75     259           LDA  CURLIN
813F: 48        260           PHA
8140: A5 B9     261           LDA  TXTPTR+1
8142: 48        262           PHA
8143: A5 B8     263           LDA  TXTPTR
8145: 48        264           PHA
8146: A9 B4     265           LDA  #ONTKN      ID to insure
                                               right nest
8148: 48        266           PHA
8149: 98        267           TYA
814A: 48        268           PHA
814B: 8A        269           TXA
814C: 48        270           PHA
814D: A0 07     271 EVALON    LDY  #7          Move TXTPTR
                                               up to cond
814F: 20 98 D9  272           JSR  ADDON       Add Y to
                                               TXTPTR
8152: 20 7B DD  273           JSR  FRMEVL      Eval condition
8155: A5 9D     274           LDA  FAC         Do loop?
8157: F0 01     275           BEQ  DROPSTAK    No
8159: 60        276           RTS              ;Yes: rtn to
                                               Aplsoft
815A: BA        277 DROPSTAK  TSX              ;Bury prev.
                                               saved
815B: 8A        278           TXA              ;loop values
815C: 18        279           CLC
815D: 69 07     280           ADC  #7
815F: AA        281           TAX
8160: 9A        282           TXS
8161: 20 B1 00  283 SKIPON    JSR  CHRGET      Skip ON loop
                                               body
8164: C9 AF     284           CMP  #AMPERTKN   until correct
                                               CONT
8166: D0 F9     285           BNE  SKIPON      is reached
8168: 20 B1 00  286           JSR  CHRGET
816B: C9 B4     287           CMP  #ONTKN
816D: D0 04     288           BNE  :2
816F: E6 FB     289           INC  ONCTR
8171: D0 EE     290           BNE  SKIPON
8173: C9 BB     291 :2        CMP  #CONTTKN
8175: D0 EA     292           BNE  SKIPON
8177: A5 FB     293           LDA  ONCTR
8179: F0 05     294           BEQ  CONT        Correct CONT
                                               reached
817B: C6 FB     295           DEC  ONCTR
817D: B8        296           CLV
817E: 50 E1     297           BVC  SKIPON
8180: 20 B1 00  298 CONT      JSR  CHRGET      Get off CONT
                                               token
8183: 4C D2 D7  299           JMP  NEWSTT      and rtn to
                                               Aplsoft
```

```
                    300 *
                    301 * Process & CONT command
                    302 *
8186: BA            303 GOTCONT   TSX                ;Make sure we
                                                     have
8187: BD 03 01      304           LDA  $103,X        properly
                                                     nested cmds
818A: C9 B4         305           CMP  #ONTKN
818C: F0 03         306           BEQ  *+5
818E: 4C 0B DD      307           JMP  NESTERR
8191: BD 04 01      308 GETFMSTK  LDA  $104,X        Retrieve prev.
                                                     saved
8194: 85 B8         309           STA  TXTPTR        values
8196: BD 05 01      310           LDA  $105,X
8199: 85 B9         311           STA  TXTPTR+1
819B: BD 06 01      312           LDA  $106,X
819E: 85 75         313           STA  CURLIN
81A0: BD 07 01      314           LDA  $107,X
81A3: 85 76         315           STA  CURLIN+1
81A5: 8A            316           TXA                ;Bury this
                                                     loop's
81A6: 18            317           CLC                ;values
81A7: 69 07         318           ADC  #7
81A9: AA            319           TAX
81AA: 9A            320           TXS
81AB: 4C D2 D7      321           JMP  NEWSTT        Return to
                                                     Aplsoft
                    322           LST  OFF           Turn off
                                                     symbol table
                                                     listing
```

--End assembly--

430 bytes

Errors: 0

Programming Aids By Sensible Software

For some time, I have used software written by Sensible Software and have found their products to be very useful, of high quality and economically priced. This is a brief introduction to three of their fine products for enhancing Applesoft: Edit-Soft (a line editor), APLUS (structured form of Applesoft and "pretty lister"), and B.E.S.T. (optimizer and cross referencer).

EDIT-SOFT:

Edit-Soft is an Applesoft line editor that is easy to use and filled with features. To run it, you simply RUN the HELLO program on the disk, which will install your choice of the 48K or the 64K version of Edit-Soft. If you select the 48K version, Edit-Soft will load in memory between DOS and its buffers. The 64K version loads into the RAM card.

Let's take a look at a sample session, using Edit-Soft to create a program that asks for a password and does a CATALOG if the password is correct. After we RUN HELLO, Edit-Soft will tell us that it is ready and will return us to Applesoft.

What next? Let's ask Edit-Soft.

```
&HELP
```

HELP TABLE

Commands For Use In Editor Or Applesoft

&	ENTER EDITOR
&###	EDIT LINE ###
&A###,###	AUTO (START###, STEP###)
&C	CTL-CHR IN INVERSE
&F###	FIND (SEARCH FROM ###)
CTRL-F	CONTINUE SEARCH
&H	HELP--LIST COMMANDS
&K	KILL--REMOVE EDITSOFT
&S###	SPLICE LINE ###

Macro Commands:

&B	SAVE MACRO DEFINITIONS
&L	LOAD MACRO DEFINITIONS
&M	DEFINE MACRO
&R	REPLACE MACRO
&D1/D2	DEFINITION TABLE,PG 1 OR 2

ANY KEY TO CONTINUE

CTL-S	LOWER CASE
CTL-X	EXIT EDITOR
SHFT-CTRL-M	CTL-CHAR OVERRIDE
ESC-RETURN	ENTER ENTIRE LINE
→	INSERT
←	DELETE
ESC	GO TO BEGINNING OF LINE
CHAR	CHARACTER SEARCH
CTL-C	COMPRESS
CTL-E	GO TO END OF LINE
CTL-V	VERIFY END OF BUFFER
CTL-L	PRINT [
CTL-K	PRINT \
CTL-J	PRINT _

Now that we know what the commands are, let's see what the macros look like:

```
&D1
```

CTL-O	CATALOG
CTL-T	TEXT:HOME:POKE-16300,0
CTL-L	LIST
CTL-P	PRINT
CTL-D	PRINT CHR$(4)"
CTL-R	RUN
CTL-V	CALL -151
CTL-A	12,U
CTL-E	33,R
CTL-Z	24,D

We can now begin to write a program. Let's use the autonumber feature:

```
&A100

100 REM
110 REM * Edit-Soft Demo *
120 REM **********<CTRL-SHIFT-
    M><M>
```

```
130 INPUT "ENTER PASSWORD: ";PW$
140 IF PW$ < > "ABCD" THEN 130
150 PRINT CHR$(4)"CATALOG"

&100
100 REM ****************
```

Note: We edited line 100 after writing the program so that we could tell how long it should be. Line 110 shows that entering lower case can be done by using CTRL S as a shift toggle. Line 120 shows that a <CR> can be embedded at the end of a REM *to space down the next line. When you enter the quotation mark in line 130, a character count is displayed on a status line at the top of the screen so that you can format strings properly while you're on the screen. In line 150, we used the CTRL D macro to quickly print the first half of the statement. Pretty nice!*

APLUS:

APLUS enhances Applesoft to allow you to use structured programming to simplify your programming efforts. APLUS also includes a "pretty lister" which gives you a neatly formatted printout of your program (APLUS programs only) when you use the &LIST command.

To create an APLUS program, simply program in Applesoft as you normally would . . . with or without your choice of an Applesoft line editor. You are allowed to use certain additional commands like WHEN ELSE, UNLESS FIN, etc., to give your program structure. By using the &LIST command instead of LIST, the printout will be correctly formatted, showing off the control structures. The program can be SAVEd and LOADed just as a normal Applesoft would. To run the program, however, you must first LOAD it into memory and use &CONVERT to convert the APLUS commands into normal Applesoft.

Here's an example of excerpts from an APLUS game demonstration program from the system disk and its Applesoft converted equivalent *(Remember: this is not the entire program):*

```
1000 "DO INITIALIZE"
1010 UNTIL (SL= > 10OR SR= > 10)
1020 : "DO DRAW GAME-BOARD"
1030 : UNTIL (LH< > 0OR RH< > 0)
1040 : : "DO CHANGE-PLAYER-DIRECTION?"
1050 : : "DO CALC-PLAYER-MOVES"
1060 : : "DO MOVE-PLAYERS"
1070 : : "DO SKILL-DELAY"
1080 : ::FIN
1090 : "DO ANALYZE WIN"
1100 ::FIN
1110 "DO GIVE FINAL RESULTS"
1120 END

  ==============================
2290 "TO INITIALIZE"
2300 : SL= 0:SR= 0:REM SCORES
2310 : TEXT :HOME
2320 : VTAB 10:HTAB 14
2330 : FLASH
2340 : PRINT "BARRICADE"
2350 : NORMAL
2360 : VTAB 17:HTAB 6
2370 : PRINT "PRODUCED BY SENSIBLE
       SOFTWARE"
2380 : VTAB 23
2390 : PRINT "COPYRIGHT 1979 -- ALL
       RIGHTS RESERVED"
2400 : SKILL= 1400
2410 : "DO DELAY"
2420 : HOME
2430 : PRINT "THE OBJECT OF BARRICADE IS
       TO FORCE"
2440 : PRINT "YOUR OPPONENT TO HIT A WALL
       BEFORE YOU":PRINT "DO."
2450 : PRINT :PRINT
2460 : PRINT "HOW GOOD ARE YOU (0=NOVICE,
       9=EXPERT)"
```

```
2470 : INPUT SKILL
2480 : SKILL= INT (10- SKILL)
2490 : IF SKILL< 0OR SKILL> 9THEN SKILL= 9
2500 : SKILL= 30* (SKILL- 1)
2510 ::FIN

 ============================
2520 "TO DELAY"
2530 : FOR D= 1TO 3000:NEXT D
2540 ::FIN
```

Applesoft Converted Equivalent:

```
1000 GOSUB 2290:::::::::::::::
1010 REM NTIL(SL = > 10 OR SR = > 10)
1020 GOSUB 2060::::::::::::::::::
1030 REM NTIL(LH < > 0 OR RH < > 0)
1040 GOSUB 1130::::::::::::::::::::::::::::
1050 GOSUB 1420::::::::::::::::::::
1060 GOSUB 1720::::::::::::::::
1070 GOSUB 2550::::::::::::::::
1080 IF NOT (LH < > 0 OR RH < > 0) GOTO 1030
1090 GOSUB 1840::::::::::::::::
1100 IF NOT (SL = > 10 OR SR = > 10) GOTO 1010
1110 GOSUB 1980::::::::::::::::::::::
1120 END
2290 REM ::::::::::::::
2300 SL = 0:SR = 0: REM SCORES
2310 TEXT : HOME
2320 VTAB 10: HTAB 14
2330 FLASH
2340 PRINT "BARRICADE"
2350 NORMAL
2360 VTAB 17: HTAB 6
2370 PRINT "PRODUCED BY SENSIBLE
     SOFTWARE"
2380 VTAB 23
2390 PRINT "COPYRIGHT 1979 -- ALL RIGHTS
     RESERVED"
2400 SKILL = 1400
```

```
2410 GOSUB 2520::::::::::
2420 HOME
2430 PRINT "THE OBJECT OF BARRICADE IS
     TO FORCE"
2440 PRINT "YOUR OPPONENT TO HIT A WALL
     BEFORE YOU": PRINT "DO."
2450 PRINT : PRINT
2460 PRINT "HOW GOOD ARE YOU (0=NOVICE,
     9=EXPERT)"
2470 INPUT SKILL
2480 SKILL = INT (10 - SKILL)
2490 IF SKILL < 0 OR SKILL > 9 THEN SKILL = 9
2500 SKILL = 30 * (SKILL - 1)
2510 RETURN ::
2520 REM :::::::::
2530 FOR D = 1 TO 3000: NEXT D
2540 RETURN ::
```

B.E.S.T.:

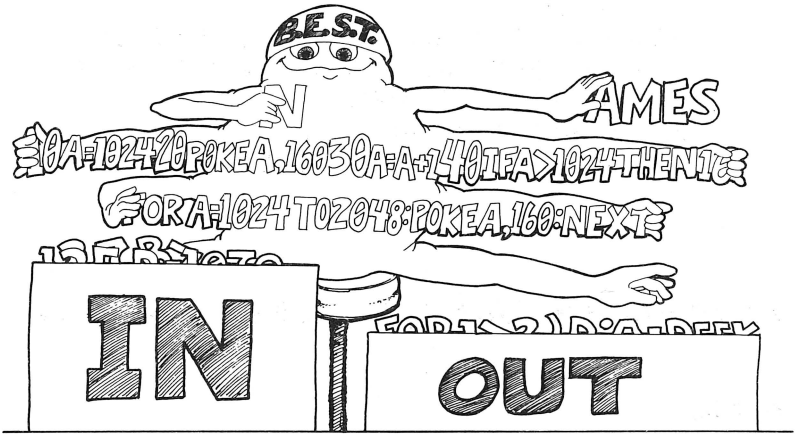
B.E.S.T. stands for BASIC Enhanced Software Tools and is a handy utility for optimizing, cross referencing, renumbering and merging Applesoft programs.

To optimize a program means to change it so that it runs faster and is more compact. Cross-referencing prints out a list of variables, and a list of line numbers and where they are used. This is useful in debugging, spotting logic errors and in identifying unused sections of code.

We will use B.E.S.T. here to let you see what the previous program looks like when optimized. To use B.E.S.T., BRUN B.E.S.T. LONG (the LONG refers to the "long" version which has more space-consuming features). You will be presented with a help menu of commands and their meanings. This menu can be displayed at any time by typing in &HLP.

Next, we load the above program, taking care to remember that it is the normal Applesoft conversion of the APLUS program. We use the B.E.S.T. command &MEM and B.E.S.T. tells us that the program is 2,630 bytes long. Now let's see what B.E.S.T. can do.

We will first optimize the variable names by shortening them to one character (reassignment of names begins with "A" and proceeds through each letter of the alphabet). Shortening the names will reduce the size of the program, cutting the amount of time needed by GOTO statements to search through the program for their destination line numbers. Variable optimization is performed with the &VOP command, which quickly tells us that the program size is now 2,472 bytes. That's about a 6 percent savings in space, but we can do even better than that.



We will use B.E.S.T.'s other optimization option, &ROP, which removes REMarks, combines smaller lines into big ones and renumbers the program with as small as possible line numbers. This reduces program size even more and cuts the time it takes to go from one line number to the next. &ROP quickly does its work and tells us that the program is now 1,422 bytes long, nearly half of the program's original size!

This is what the optimized program looks like:

```
1 GOSUB 48
2 GOSUB 47
3 GOSUB 6: GOSUB 23: GOSUB 39: GOSUB 52:
  IF NOT (A < > Ø OR B < > Ø) GOTO 3
```

```
4 GOSUB 40: IF NOT (C => 10 OR E => 10)
  GOTO 2
5 GOSUB 46: END
48 C = 0:E = 0: TEXT : HOME : VTAB 10: HTAB
   14: FLASH : PRINT "BARRICADE": NORMAL
   : VTAB 17: HTAB 6: PRINT "PRODUCED BY
   SENSIBLE SOFTWARE": VTAB 23: PRINT
   "COPYRIGHT 1979 -- ALL RIGHTS
   RESERVED":X = 1400: GOSUB 51: HOME
   : PRINT "THE OBJECT OF BARRICADE IS TO
   FORCE": PRINT "YOUR OPPONENT TO HIT A
   WALL BEFORE YOU": PRINT "DO.": PRINT
   : PRINT
49 PRINT "HOW GOOD ARE YOU (0=NOVICE,
   9=EXPERT)": INPUT X:X = INT (10 - X): IF
   X < 0 OR X > 9 THEN X = 9
50 X = 30 * (X - 1): RETURN
51 FOR D = 1 TO 3000: NEXT D: RETURN
```

Programming Aids By Delta Micro Systems

BASIC'

Delta Micro Systems publishes a powerful pre-processor for Applesoft called BASIC' (pronounced "basic prime"). BASIC' is sort of a scaled down version of Pascal, although it has some features that Pascal does not. For example, its REPEAT UNTIL loop may have the UNTIL anywhere within the loop instead of only at the end as is the case with Pascal. This gives added flexibility in controlling loop exits. It also does away with the need for terminators like NEXT, WEND, IFEND, etc. by using a convenenient auto-indent feature instead.

BASIC' works by giving you access to a line editor to create a structured form of Applesoft. This program is saved in a text file which is automatically printed out on paper, properly formatted with page headings and the date! You can use BASIC's translator to convert the text file into a standard Applesoft program.

The entire package is easy to use and allows you to write beautiful Applesoft programs. It also is just plain fun to use. Following are some examples, showing what a typical BASIC' prime source file looks like, how BASIC' prints it out, and how it converts into Applesoft. The program is taken from a demonstration in the BASIC' user's manual and is designed to print out a range of ASCII characters.

BASIC' Source Code:

```
!=
!>BASIC' EXAMPLE
!>DELTA MICRO SYSTEMS
!=
PRINT "ASCII Character Tables" ! Heading
REPEAT
   INPUT "From,To: ";N,M
   UNTIL N = Ø AND M = Ø ! Double zeroes to quit
   DO TABLE
END
PROC TABLE ! ASCII characters for codes N
through M
   PRINT "Code","Char" ! Print headings
   FOR I = N TO M
     UNTIL I > 127 ! Upper limit
     PRINT I, ! Code
     IF I < 32 ! Control code
       CASE I
         # 13 ! Carriage return
           PRINT "<RTN>"
         # 27 ! ESCAPE
           PRINT "<ESC>"
         ELSE ! All others
              PRINT "<CTRL> ";CHR$(64+I)
     ELSE ! Printable characters
       PRINT CHR$(I)
```

"Pretty Listing" of BASIC' Source Code:

```
B A S I C' V1.2 #1 ASCII DEMO.T        31-JUL-83 PAGE 1

      LN# REF
```

```
 1      ! ========================
 2      !          BASIC' EXAMPLE
 3      !       DELTA MICRO SYSTEMS
 4      ! ========================

 5 200 PRINT "ASCII Character Tables"  ! Heading
 6 201 REPEAT
 7 201     INPUT "From,To: ";N,M
 8 202     UNTIL N = 0 AND M = 0       ! Double
       zeroes to quit
 9 203     DO TABLE
10 205 END

       ! ========================


11 400 PROC TABLE          ! ASCII characters for
       codes N through M
12 400    PRINT "Code","Char"     ! Print headings
13 400    FOR I = N TO M
14 401       UNTIL I > 127          ! Upper limit
15 402       PRINT I,                    ! Code
16 403       IF I < 32              ! Control code
17 404          CASE I
18 404             # 13          ! Carriage return
19 405                PRINT "<RTN>"
20 407             # 27                ! ESCAPE
21 408                PRINT "<ESC>"
22 410             ELSE                ! All others
23 411                   PRINT "<CTRL> ";
                      CHR$(64+I)
24 412          ELSE        ! Printable characters
25 413             PRINT CHR$(I)
```

```
B A S I C' V1.2 #1 ASCII DEMO.T    31-JUL-83 PAGE 2

LN# REF

PROCEDURE CROSS REFERENCES:

TABLE        # 400 11 : *MAIN* 9
```

Translated Applesoft Code:

```
200 PRINT "ASCII Character Tables"
201 INPUT "From,To: ";N,M
202 IF N = 0 AND M = 0 THEN 205
203 GOSUB 400
204 GOTO 201
205 END
400 PRINT "Code","Char": FOR I = N TO M
401 IF I > 127 THEN 415
402 PRINT I,
403 IF NOT (I < 32) THEN 413
404 IF NOT (I = 13) THEN 407
405 PRINT "<RTN>"
406 GOTO 412
407 IF NOT (I = 27) THEN 411
408 PRINT "<ESC>"
409 GOTO 412
410 GOTO 412
411 PRINT "<CTRL> "; CHR$ (64 + I)
412 GOTO 414
413 PRINT CHR$ (I)
414 NEXT I
415 RETURN
```

Output:

```
RUN
ASCII Character Tables
From,To: 10,15
Code            Char
10          <CTRL> J
11          <CTRL> K
12          <CTRL> L
13          <RTN>
14          <CTRL> N
15          <CTRL> O
16          <CTRL> P
17          <CTRL> Q
```

Summary

We examined three methods of aiding your program development work in Applesoft: through the use of the & character (ASA by Bill Parker), through the use of added commands in an Applesoft program (APLUS, by Sensible Software) and through the use of a sophisticated **pre-processor** (BASIC' by Delta Micro Systems).

There are many aids available for your work with Applesoft. Here are just a few of the more popular ones:

Ampersand Utilities:

Amper-Array, Chart, Screen, Sampler I/SDS ($49.95 ea.)
Amper-Magic/ADS ($75. vol. 1, $35. vol. 2)
Ampermanager/CA ($22.50, must be a member)
Ampersoft/MS ($49.95)
Routine Machine/SDS ($64.95)

Applesoft Editors:

A.C.E./SDS($39.95)
Applesoft Editor Package/PSP ($40)
Edit-Soft/SS ($39.95)
ES-CAPE/SC ($60)
GALE/MS ($49.95)
Global Program Line Editor/CA ($38.50, must be a member)

Applesoft Pre-Processors:

APLUS/SS ($39.95)
BASIC'/DMS ($129)

Applesoft Optimizers:

B.E.S.T./SS ($40.00)

Key to above publisher code:

ADS = Anthro-Digital Software
Box 1385
Pittsfield, MA 01202

CA = Call -A.P.P.L.E.
21246 68th Ave. S.
Kent, WA 98032
(206) 872-2245

DMS = Delta Micro Systems
Box 15952
New Orleans, LA 70175
1-800-535-1814 (toll free)

MS = Micro-Sparc
Box 325
Lincoln, MA 01773

PSP = Peters Soft-Products
Box 694
Didsbury, Alberta
CANADA
T0M 0W0

SDS = Southwestern Data Systems
10761 Woodside Ave #E
Santee, CA 92071
(619) 562-3670

SS = Sensible Software
6619 Perham Drive
West Bloomfield, MI
(313) 399-8877

CHAPTER 11

STRUCTURED LANGUAGES

This is just a brief look at some truly structured languages for those who are interested in going beyond Applesoft. We'll be taking a look at Apple UCSD Pascal and a "new kid on the block" called C.

Applesoft

Here is a sample Applesoft program that runs an empty loop for 10,000 iterations. We'll time it and use it as a comparison against our two structured languages:

```
100 REM THIS APPLESOFT PROGRAM
110 REM RUNS AN EMPTY LOOP FROM
120 REM 1 TO 10,000

130 LET I = 1
140 PRINT "START. "; CHR$ (7);"
      I = ";I
150 FOR I = 1 TO 10000: NEXT
160 PRINT "STOP. "; CHR$ (7);"
      I = ";I
```

Execution time: 10.5 secs.

Notice the typical "unattractive" features of BASIC that make it hard to read: lines broken in the middle, spacing between characters in odd places, and all letters are in upper case. It also runs relatively slow. Compare this with the following Pascal program:

Pascal

This Pascal program runs an empty loop 10,000 times.

```
Program Foo;
Var I: Integer;
Begin
  I := 1;
  Writeln('Start.      ',Chr(7),'I = ',I);
  For I :=1 To 10000 Do;
  Writeln('Stop.       ',Chr(7),'I = ',I);
End.
```

Execution time: 6.0 secs.

Here, we can see that clarity has been greatly enhanced by proper spacing, using upper and lower case and avoiding line breaking. There also has been an increase in speed: it runs nearly twice as fast as Applesoft. Some features that are different from BASIC are: statements must end with a ;. Variable names (called identifiers) must be declared as to their type (avoiding the need to end a variable name with things like $ or %). I := 1 means assign 1 to I, I = 1 means compare I to 1. Strings are encased (delimited) by apostrophes and not quotes; there is no NEXT in a FOR loop, but there is a Do; Writeln means PRINT; and Chr is used instead of CHR$.

C

Now, let's take a look at a C equivalent:

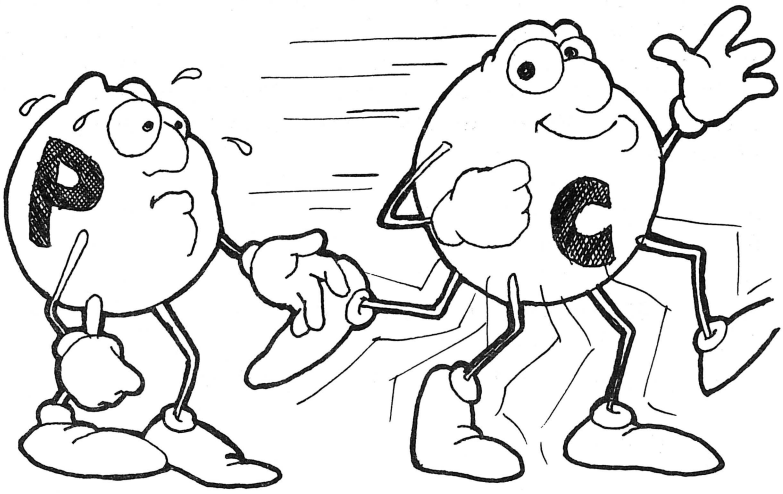
This C program runs an empty loop for 10,000 iterations.

```
main()
{
        int i;
        i = 1;
        printf("Start.\007 I = %d.\n", i);
        for(i = 1; i != 10000; ++i);
        printf("Stop.\007 I = %d.\n", i);
}
```

Execution time: .75 sec.

A C program is similar in structure, in many respects, to a Pascal program, but it runs considerably faster. This is because a C program is compiled into "native code," while UCSD Pascal is compiled into a slower running "p-code." Native code is the Apple's own 6502 machine code which can be executed directly. P-code is a pseudo-machine code, similar to BASIC "tokens," which must go through the time consuming step of being "interpreted" each time the program is run. P-code does have its advantages, however; it is more compact and it can be executed on different types of computers using appropriate p-code interpreters.

You can see that C has some different ways of expressing things: printf means print formatted or a sort of PRINT USING statement; / is used to send a special character to the C compiler, such as 007, which is the ASCII code for a bell or beep, and n, which is the command for a "new line" or <CR>; %d tells C where to put a decimal number in the string to be printed; finally, the braces ({}) are used to mark the beginning and end of a structure.



In case you are curious, here are some details about C: It was invented by a person by the name of Dennis Ritchie in the 1970's at Bell Laboratories. It's called "C" because its predecessors were called "BCPL" and "B." It was originally developed for use on a PDP-11 minicomputer under the UNIX operating system.

C is considered to be the language of the future. It has all the structure and elegance of Pascal, but few of the limitations. C has recently become available for Apple programmers with the release of the Aztec C compiler from Manx Software Systems. This is an excellent implementation of C and works within the DOS 3.3 environment. A CP/M (Z-80 card required) version and a ProDOS version are also available or will be soon.

You will be interested to know that when you run the Aztec C program (which is done simply by BRUNning a certain program), part of Apple DOS is rewritten, turning your Apple into a UNIX-like machine. (UNIX is an operating system used on large, powerful computers.) You have total access to any of your DOS 3.3 disks and can manipulate standard DOS files with an impressive array of powerful commands. A word processor (to write and edit source programs), an assembler and a relocating linking loader are included. The Aztec C version, running under DOS 3.3 is well thought out and easy to use. The added power it gives you as a programmer really makes it something to consider.

Of course, when working with these more sophisticated languages, you must give up one of the strong points of BASIC: immediate response during program testing. All things considered, it sure is convenient to be able to RUN a program and immediately see whether or not it runs. With structured languages, the programs must be compiled first, before it can be run. This can be quite time-consuming, but in the long run, it makes for better programs and easier programming sessions. And that's the whole name of the game, isn't it?

Summary

Two of the most popular structured languages currently available on the Apple are **Pascal** and **C**. They offer elegance, clarity, speed and the capability of being run on different types of computers. However, they must be compiled and they do not offer the same sort of spontaneity that BASIC does.

Bibliography

Borgerson, Mark, *A BASIC Programmer's Guide to Pascal*, New York, NY: John Wiley & Sons, 1982. ($9.95)

Cooper, Michael, Doug and Clancy. *Oh! Pascal*, New York, NY: W.W. Norton, 1982. ($15.95) Nearly 500 pages of textbook-like explanations, sample programs and diagrams. Modern, sometimes off-beat approach. Used as a textbook for teaching beginning-level programming at some universities.

Fox, Michael, David and Waite, *Pascal Primer*, Indianapolis, IN: Sams Books, 1981. ($16.95)

Koffman, Elliot, *Problem Solving and Structured Programming in Pascal.* Reading, MA: Addison & Wesley, 1981. ($13.95) Very clear and compact work. Part of a series on structured programming.

Ledgard, Henry, John Hueras, and Paul Nagrin, *Pascal With Style: Programming Proverbs*, Rochelle Park, NJ: Hayden, 1979. ($6.95)

Ledgard, Henry and Andrew Singer, *Elementary Pascal*, New York, NY: Vintage Books, 1982. ($12.95)

Lewis, T.G., *Pascal Programming for the Apple*, Reston, VA: Reston Publishing, 1981. ($112.95)

Peckham, Herbert and Arthur Luehrmann, *Apple Pascal: A Hands On Approach*, New York, NY, McGraw-Hill, 1981. ($14.95). A must-have for beginning Apple Pascal programmers. More than 400 pages, very clear.

Zaks, Rodnay, *Introduction to Pascal (Including UCSD Pascal)*, Berkeley, CA: Sybex, 1980. ($12.95)

BDS C User's Group, Box 287, Yates Center, KS 66783.

Dr. Dobb's Journal, 1263 El Camino Real, Menlo Park, CA 94025. A colorful and authoritative monthly magazine that carries many useful programs and utilities.

Dwyer, T., *C and the Personal Computer*, Reading, MA: Addison-Wesley, expected in 1985.

Kernighan and Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

Manx Software Systems, Box 55, Shrewsbury, NJ 07701, (201) 780-4004. Sells the 6502/DOS 3.3 Aztec C compiler for $199. CP/M and ProDOS versions also available.

Purdum, Jack, *C Programming Guide*, Indianapolis, IN: Que Corporation, 1983. Excellent, very readable book. Draws on examples from a BASIC programmer's standpoint. Very complete coverage.

Index

C

D

E

F

G

H

N

O

P

R

S

T

U

V

W

#

INTERMEDIATE APPLE

Perfect for the BASIC programmer who is ready to move on.

THE INTERMEDIATE APPLE will take you from being a fledgling Applesoft programmer and show you important principles that can help you handle more complicated programming problems. You'll learn how to structure your program "one step at a time," reducing big problems into smaller, more manageable ones. Your programming will become clearer and easier to understand in the process.

The many benefits of structured Applesoft programming include:

- **Programs are easily understood.**
- **Errors are reduced.**
- **Programs are simple to maintain.**
- **Coding is faster, speeding up program development.**
- **Easy transition to other high level languages.**
- **Coding large programs is simplified.**

In addition, you'll learn about flow diagrams, algorithms, text files, enhanced graphics, special printer techniques, and many more tricks of the trade. So if you're ready to take that intermediate step, THE INTERMEDIATE APPLE is ready for you!

ISBN 0-88190-241-1

0 48831 00150

DATAMOST INC™

20660 Nordhoff Street, Chatsworth, CA 91311-6152
(818) 709-1202

Bill Parker

INTERMEDIATE APPLE