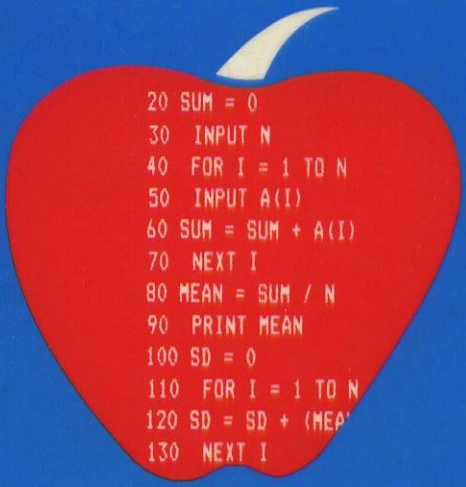
Applesoft BASIC

A TEACH-YOURSELF INTRODUCTION



B.M. PEAKE


Scanned by cvxmelody
http://www.cvxmelody.net/AppleUsersGroupSydneyAppleIIDiskCollection.htm

Applesoft BASIC

A TEACH-YOURSELF INTRODUCTION

B.M. PEAKE

Edward Arnold

First published in the United Kingdom 1983 by
Edward Arnold (Publishers) Limited
41 Bedford Square
London WC1B 3DQ

ISBN: 0 7131 3498 4

Printed and bound by Thomson Litho Ltd, East Kilbride, Scotland.

CONTENTS

FOREWORD

This text arose from a need to introduce students with little or no background in programming to using the Apple II Plus microcomputer system. This was to be achieved with the minimum of teacher supervision. It is designed as an introduction to Applesoft BASIC and describes, in order of increasing complexity, the major statement types and concepts in this computer language. This order follows one which has over many years proved to be successful for introducing students to other high level languages on main frame computers. Thus strings and functions, for example, are not specifically discussed until well into the text, in contrast to many current texts on BASIC for microcomputers which introduce them at the beginning. *No previous knowledge of any computer language is assumed.* Direct access to an Apple II computer is desirable but not essential during the study of the text.

The only way to become proficient in a computer language is to write and execute programs yourself. Hence along with examples to illustrate each programming topic, there are numerous exercises which the reader is encouraged to at least attempt. The exercises often consist of two parts: the first (in normal type) involves answering questions or writing a program while the second part (*in italics*) involves entering and executing the program written in the first part of the exercise.

Model answers to the exercises are provided where appropriate in Appendix IV. There are often many different ways in which to write a program to solve a given problem. Hence the model answers should be regarded as only a *guide* and are not necessarily the most efficient or elegant way to solve the problem. In particular, the use of multi-statement lines (which often leads to more efficient programs) is generally avoided. This should lead to greater clarity and understanding of the answers. No attempt is made to incorporate any structure into the programs which is possible to a limited extent in BASIC.

Considerable thought was given to the merits of including a chapter on graphics in such an introductory text. Even the simple aspects of this topic could well be beyond the ability or interests of some readers. However, graphics is an area of computing which is becoming increasingly important and one for which the Apple is well suited. For these reasons, a substantial chapter on this topic is included towards the end. This material is somewhat independent of the remainder of the text and hence could well be passed over on a first reading.

No attempt has been made to provide a comprehensive coverage of all aspects of Applesoft BASIC. However, I believe that the important features necessary for the use of this language on the Apple computer for many common applications have been covered.

Every effort has been made to eliminate errors in the text material and in programs given as model answers. However, I would welcome from readers notification of any errors, comments or suggestions for improvements.

I would like to thank my students for patiently testing many aspects of the text during its preparation; Miss Sally Wood for her typing efforts; and my colleagues, particularly Messrs Greg Hormann, Bruce McMillan and John Barr, for their advice and encouragement.

Finally, my sincere thanks to my family, Margaret, Jonathan and Simon, for their patience, tolerance and understanding over many months during the preparation of this text.

Barrie M. Peake
Dunedin, New Zealand
September 1982

Addendum

An updated version (Apple IIe) of the Apple II Plus microcomputer was released in February 1983. This new system contains significant improvements, extensions and simplifications to the hardware. In particular there is provision for 80 columns of either upper and lower case letters on the video display. There is also an expanded key-board with some extra keys which provide new functions and operations. However all software written in Applesoft BASIC including that described in this text, will run with no change (or minimal change) on the Apple IIe model.

B.M.P.

Chapter 1

Starting to Use the Apple II Plus Microcomputer System

Welcome to the Apple II Plus microcomputer! The main component essential for the operation of this system is the Apple II Plus microcomputer unit which contains a microprocessor chip (MOS Technology 6502), power supplies, memories and other electronic circuitry, including that for connecting (interfacing) the unit to external devices. All these components are mounted on printed circuit boards and enclosed in a plastic case together with a keyboard. Also included in a typical system configuration (see Figure 1) is a display screen (video monitor), external devices for storing program instructions and data such as one or more disk drives or a cassette recorder, and a printer for output.

FIGURE 1

All these components are collectively referred to as *hardware*, as distinct from the program instructions and data which you enter into the system yourself, which is called *software*. An intermediate category of *firmware* exists to describe those program instructions and data which are stored permanently (normally by the manufacturer) in the electronic memory of the microcomputer unit. In the Apple II Plus system such firmware includes the operating system known as the *Monitor*.

Additional components include circuit boards placed inside the microcomputer unit which allow the use of other computer languages such as

PASCAL or provide interfaces to external devices such as games controllers, a graphics sketch pad, other input/output (I/O) devices, or laboratory instruments.

Ex. 1.1

Identify the components in your system. Locate the positions of the power ON-OFF *switches. For the main microcomputer unit this is located at the rear left hand side by the mains cord. Note that the disk drives have no power* ON-OFF *switches of their own but are instead powered directly from the main unit. Turn on the power switches.*

SWITCHING ON

The POWER light on the keyboard (bottom left hand), will come on and remain on until the system is switched off. Note that this light is not a key that can be depressed.

The title APPLE][should appear on the top of the screen. A few clacking noises followed by a continuous whirring sound will be heard from the disk drive and the red IN USE light on this device will come on. The instructions for producing this screen output are an example of firmware, which remains permanently in memory even when the computer is turned off.

If there is *no* disk in the disk drive, this noise will continue until the RESET key (top right hand of keyboard) is pressed. This gives rise to a 'beep' sound and will cause the APPLE][title to disappear leaving only a *prompt* ( ]for Applesoft BASIC) and blinking *cursor* ■ at the bottom left hand of the screen. The Apple is now ready to receive instructions.

If there *is* a disk in the disk drive, this noise will cease after ~ 6 sec, without having to key RESET. This is because the *disk operating system* (DOS) which is a program stored on all initialised disks (see Chapter 9), has been read from the disk and stored in the computer memory. This process is referred to as bootstrapping or 'booting' DOS. A variety of messages may now appear on the screen and these will be discussed in Chapter 9. In your first attempts at using the Apple and until you want to store a program or data, it is suggested that you do not use a disk.

SWITCHING OFF

NEVER TURN OFF THE APPLE IF THE DISK DRIVE IS IN USE (RED LIGHT ON) WITH A DISK IN PLACE.

This action *could* destroy the disk as well as any program or data files being transferred to or from it at that time. If this disk drive light cannot be turned off by other means, then any possible damage to the disk can be *minimised* by opening the disk drive door and then switching the power off.

Otherwise, just turn off the power ON-OFF switches on the main unit, video monitor and any other external devices such as a printer.

Keyboard (See Figure 2)

This is the main device used on an Apple II system for inputting information into the microcomputer.

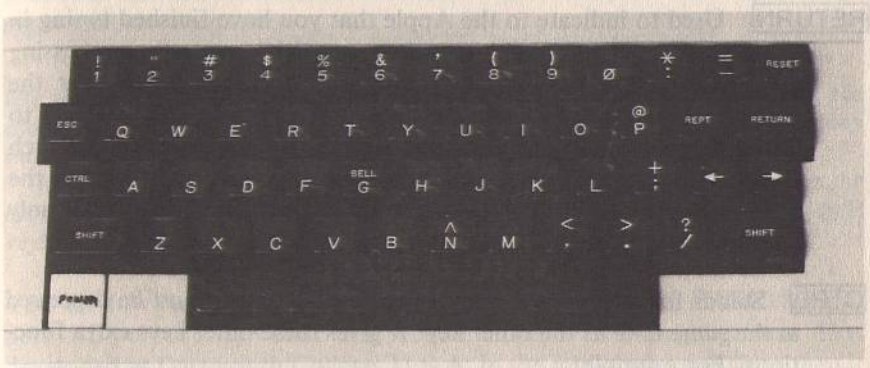


FIGURE 2
Apple II Keyboard

This has many similarities to a normal typewriter keyboard. Use of either one of the two SHIFT keys enables the upper symbols on a given key to be used. Only capital letters are available unless a special lower case circuit board is inserted into the main unit. When using the SHIFT key, depress it *first* before any other keys and *keep it depressed* while typing the other required key.

Additional keys and a brief description of their functions, beginning at the top right hand part of the keyboard (see Figure 2), are:

RESET When depressed, all processing stops. Upon release, a special series of firmware instructions are executed. These will *generally* reset all controls etc. to how they were immediately before and should leave any existing program in memory.

* Multiplication operator.

∅ This symbol represents numeric zero and has the stroke through it in order to clearly distinguish it from an alphabetic O.

ESC Stands for 'ESCape' and when used in conjunction with one of eleven other specific keys, will cause either the cursor to move in certain directions or the screen to be cleared in a specific manner. In particular ESC @ causes the screen to be cleared and the blinking cursor to be placed in the top left hand corner. To use this key, depress and *then release it*, before typing the other key.

REPT Stands for 'REPeaT' and when depressed along with any other key, will cause that key's character to appear repeatedly on the screen at a rate of about 10 times per second.

RETURN Used to indicate to the Apple that you have finished typing in a given line(s) of information and you want it sent to the processing unit. This key is used after almost all commands. Note that the RESET key is located immediately above this key and it is very easy to mistakenly depress RESET rather than RETURN. The results of such an action can be disastrous and to this end, there is a switch inside the main Apple II Plus unit which can be set so that the RESET key is only operative when the CTRL key is simultaneously depressed.

CTRL Stands for 'ConTRoL' and like the SHIFT key it must be depressed at the *same time* as the other key. It gives these other keys extra functions. For example:

CTRL BELL/G causes the speaker to 'beep'.

CTRL C will cause the computer to stop whatever it is doing. This will abort listings and display the message: BREAK on the screen. If the program is waiting for an input, use CTRL C RETURN.

CTRL S temporarily stops listings of programs.

CTRL X cancels the most recent command. It prints the non-keyboard character \ after the cancelled command and then returns the Applesoft prompt and cursor to a new line ready to receive the next command.

← Backspace key. Each time this is depressed the cursor will move back one space *only* in the line currently being typed.

→ Retype key. Each time this is depressed it moves the cursor one character to the right in the current line without removing that character.

^ This character (top half of the N key) is used to indicate exponentiation (raising to a power). Thus X^2 is entered on the keyboard as X^2.

< > Less-than and greater-than symbols respectively and used in logical expressions (see Section 4.2).

/ Division operator.

Ex. 1.2

It is suggested that you practice keying in some of these standard characters using the following example. Start with the prompt and cursor on the left hand side of the screen and note how the cursor automatically advances one place to the right on keying in each new character:

AQ? = I ∅ XYZ(*use* REPT *key*)ZZZZZ1X3 – 4/678?SD !.;QU:J#% * L – "B

Use the ← and → keys to replace the 678 *by* 666 *and then continue with the characters. What happens to the display after the first line is full? Count how many characters are included in a full line.*

Observe the effect of keying RETURN *and then keying* ESC @ *(this requires the depression and release of* ESC *before simultaneously depressing the* SHIFT *and* @/P *keys).*

You are now ready to begin learning some simple aspects of BASIC (*B*eginners *A*ll-purpose *S*ymbolic *I*nstruction *C*ode). This language was first developed at Dartmouth College, U.S.A. in the mid-sixties and since then it has been widely adopted in various forms on many different microcomputer systems. It is available on the Apple II Plus system as Integer BASIC and Applesoft BASIC. Applesoft BASIC is available as firmware on the Apple II Plus system and it will be the only form of BASIC considered in this text.

Comments on the use of disks and disk drives, and a printer will be left until needed later in the text (see Chapter 9 and Appendix I respectively).

Chapter 2

Simple Direct BASIC Statements

2.1 PRINT Statement

It is necessary in any computational device to be able to input information and output the result(s). In the Apple the normal method of input is by depressing specific keys on the keyboard and the output appears on the monitor screen. Later we will also see how the printer can be used for output and the disk drives for input and output.

We have already seen in Ex. 1.2, that when a character is keyed in from the keyboard it is automatically outputted to the screen even without the user having to depress the RETURN key. It would obviously be desirable to be able to specify and control which particular character or combination of characters should be printed out on the screen. There is a specific BASIC instruction (statement) for such an operation called the PRINT statement and it takes various forms including:

PRINT Y and PRINT "X".

Y can be a *number* or *variable*, or a numerical *arithmetic expression* which gives a number as a result. X can be some data consisting of one or more alphabetic or numeric characters (collectively called *alphanumeric characters*). Such a collection of characters is called a *character string* and as such is always enclosed in quotation marks (" "). These will be discussed in more detail later in Section 5.4.

Some examples of the use of this statement include:

	Output
`PRINT 1.23`	1.23
`PRINT "A"`	A
`PRINT "A NICE DAY"`	A NICE DAY
`PRINT 1+1`	2
`PRINT 2^4`	16

Ex. 2.1.

(a) What output on the screen would you expect from execution of the following PRINT statements:

```
PRINT 6.28
PRINT 3+4.28
PRINT "3 +4.28"
PRINT "EX 1.2"
```

(b) Write PRINT statements to output on the screen the following lines of information:

```
2
APPLESOFT
HELLO
3.148
TEST VALUE = ?
Result of adding 2.86 to 6.27
— — — — — — —
1.23456789
```

Key each of the above statements in (a) and (b) into the Apple and observe the output on the screen. Remember to press RETURN *in order to tell the computer that you have finished inputting all the information and you would like execution to commence. If you make a mistake, key either* CTRL X *or* RETURN *and start again. Note (if you have not done so already) the response printed on the screen of*

?SYNTAX ERROR

if you misspelt the PRINT *statement by typing say* PRNT 2. *Practice with the back space (←) and retype (→) keys correcting such an error before pressing the* RETURN *key. Observe the effect of typing blanks between certain letters in the* PRINT *statement e.g.* P RINT 3.148. *What happens when you press* RETURN *a second time without entering a new* PRINT *statement? Is the original* PRINT *executed again?*

2.2 Specification and Format of Numbers

The manner in which numbers are represented and the range of values which can be used on the Apple depends on the type of number *and* the type of language being used. For Applesoft BASIC, numbers are printed with a maximum of nine digits although they may be stored in memory with more than nine digits accuracy. The different types of numbers are as follows:

Integer Numbers: these are whole numbers with no decimal points and may be in the range −32767 to 32767. Note that there is no gap or comma to indicate the thousands place. When an integer value outside this range is used an appropriate error message will appear on the screen. When outputted, any negative integer values have a − sign, while positive integer values have no sign.

Integer Number	*Output Form*
+6	6
−1	−1
0	0

Decimal Numbers: these can be of two types:

(1) *Fixed Point*: these are numbers in the range .01 to 999999999.2 and are outputted in this form with a − sign if negative. Any trailing zeros are supressed on output.

Fixed Point Number	*Output Form*
−0.02	−0.02
+99.9	99.9
100.00	100

(2) *Floating Point*: these are numbers that lie outside the above range of fixed point numbers but within the range of -1×10^{38} to 1×10^{38}. Any number smaller than about 3×10^{-39} is set to zero. They are outputted in what is called *scientific notation* in the form

(sign) X.XXXXXXXXE(sign)YY

where X... is the mantissa and YY is the exponent to the power of ten. Only the − sign is included at the beginning if the number is negative, but both + and − signs are included in front of the exponent part. Leading and trailing zeros are never outputted. If there is only one digit to be outputted after all the trailing zeros have been removed, then no decimal place appears. The exponent always contains two digits even if the first one is a zero.

Floating Point Number	*Output Form*
−0.00051	−5.1 E−04
+0.0010	1 E−03
2.860×10^{12}	2.86 E+12
0.0	0

Note that one can also *input* a floating point number in a modified scientific notation format e.g. -0.28×10^{-3} could be keyed in as −2.8 E−4 or −0.28 E−3. In input, only one digit in the exponent need be used. Although up to 38 digits may be keyed in, only the first 10 are usually used with the 10th digit being rounded off.

Ex. 2.2.

1. Predict the output that you would expect from execution of the following PRINT statements:

```
PRINT +1                    PRINT -3.18

PRINT 16087                 PRINT -0.001E12

PRINT 123456789123          PRINT 0.000

PRINT 45780.0               PRINT 1.00E-56
```


Check your predictions by keying in and executing these statements. Remember to press RETURN *after entering each line.*

2. Write PRINT statements to print out the following numbers:

2, −3, 25853, 85696, zero, 0.018, -3.14×10^{-12}

2.3 Variables

In order to generalise an instruction given to the computer, it is often necessary to replace a specific number by a symbol (*variable*). There are certain rules regarding the allowed format of the names of variables in Applesoft BASIC as follows:

1. All variable names must begin with an *alphabetic* character and may (but not necessarily) be followed by any *alphanumeric* character (e.g.):

Valid	*Invalid*
R	2R
RA	R+
R3	/R

2. A variable name may be up to 238 characters long but only the first two characters are used to distinguish between two different variable names. Thus, for example, the variable names RABBIT and ROCK would be interpreted by the computer to refer to different variables whereas RABBIT and RAKE would be considered to refer to the same variable.

3. There are certain reserved words and symbols which cannot be used as variable names because they are used in Applesoft BASIC to refer to specific operations or other functions. For example, GOTO is a reserved word used to transfer control within a program (see Chapter 3) and hence variable names such as GOTO or BGOTO are not allowed.

You may sometimes find that a supposedly unaccountable error message is in fact due to accidental use of a reserved word. Those interested can find a full list of reserved words in Appendix B of *The Applesoft Tutorial*, or Appendix G of ref [2]. (For a list of references, see Appendix III of this text.)

As for numbers, there are different types of variables and for Applesoft BASIC these fall into three categories:

1. *Integer* variables which have integer values in the range −32767 to 32767. These are indicated by attaching a % symbol after the last character in the variable name. For example, B% and TB% are valid integer variable names. Integer variables are most important for saving memory space.

2. *Real* variables can have up to 9 decimal digits and exist in the range -1×10^{38} up to 1×10^{38}. No special symbol is attached to a real variable name. Hence in the absence of such a symbol, one can assume that the variable is a real one.

3. *String* variables which refer to a collection of characters (from 0 up to 255) and which are treated as *literal* data. They are indicated by attaching a $ symbol after the last character in the variable name. For example B$ or BLAT$ would be valid string variable names. Strings and literal data will be discussed more fully in Section 5.4.

   Obviously the symbols % and $ are reserved symbols and so cannot be used in general variable names unless they are meant to refer to integer or string variables respectively.

Each of these different types of variable is assigned a specific place in the computer memory which will contain its value, and to which the computer will refer whenever that variable name is referenced.

Ex. 2.3

Give reasons for deciding on which of the following variable names are valid and indicate the variable type of all these names that are valid:

1A23, %A, X, 22/3, A12%3, D.B., USA, ROBERT'S$, NAME%, APOLLO-MOONCRAFT$, ADOG, ADOG$.

2.4 Arithmetic Operators

These are similar to their normal algebraic form with the exception of exponentiation:

Algebraic	BASIC
+	+
−	−
×	*
÷	/
exponentiation	^

An *arithmetic expression* is constructed by combining any of these operators with the appropriate numbers or variable names. As in algebra, brackets ( ) can be used to divide up various sections of an expression. For example:

	Algebraic expression	BASIC
(1)	1+2	1+2
(2)	X1 − 10.8	X1 − 10.8
(3)	366.8×9	366.8 * 9
(4)	$\frac{\sqrt{10}}{1.23}$	10^0.5/1.23
(5)	$\frac{1+2}{4+5}$	(1+2)/(4+5)

Note how in (4) the $\sqrt{\ }$ was replaced by raising (exponentiation) to the power of 0.5. An alternative expression for a square root operation is the function SQR ( ). This function will be discussed later in Section 7.1.

Note also how the algebraic expressions involving division in (4) and (5) although taking two lines in algebraic form, are written as single lines in BASIC.

Although algebraic expressions such as (5) may be unambiguous, when one comes to write out the corresponding BASIC expressions, one must be very careful of the order of the operations. Thus if (5) was written as 1+2/4+5, the numeric answer which the Apple would give upon executing this expression would be 6.5 instead of the answer obtained from execution of the statement (1+2)/(4+5) of 0.333333333. To account for this apparent anomaly, one must realise that in BASIC and indeed most other computer languages, there is an order of priority (hierarchy) for execution of arithmetic operations as follows:

Operator	
( )	Highest priority
^	↓
* /	↓
+ −	Lowest priority

The highest priority operation is executed first and if two or more operators have the same priority, then they are executed in order from left to right. Because the brackets have highest priority then any expression within the brackets, even if it involves operations of a lower priority, will be executed before any expressions outside the brackets.

This order of priority can have interesting and unforseen consequences. Thus for example (5) above, execution of the BASIC expression (1 + 2)/(4 + 5) would follow the sequence of steps:

	Step	*Result*
1.	Addition within left bracket	3/(4 + 5)
2.	Addition within right hand bracket	3/9
3.	Division	0.333333333

However, if you had been tempted to write down this arithmetic expression in BASIC as 1 + 2/4 + 5 then a quite different sequence of steps and hence final result would have occurred:

	Step	*Result*
1.	Division	1 + 0.5 + 5
2.	Addition on left	1.5 + 5
3.	Addition	6.5

This example clearly indicates how important it is to remember the hierarchy of arithmetic operations and to make liberal use of brackets to ensure

that the computer executes the operations *in the order that you as a programmer intended.*

Note that *all* operations must be included in the BASIC arithmetic expression including those which are often left out in algebra. For example, the product of three variables A, B and C which would be written in algebra as simply ABC *must* be written in BASIC as A*B*C. The expression ABC in BASIC would be considered as simply a single variable. Another example is the algebraic expression $2X^2$ which would be coded in BASIC as 2*X^2 rather than 2X^2 which would be considered as an invalid variable name because it starts with a numeric character.

It has already been noted in Section 2.1 that any arithmetic expression may be incorporated into the PRINT statement. Such an expression may contain variable names as well as numbers providing the value of these variables are known. Otherwise the Apple assumes they are zero.

Ex. 2.4

1. Predict the order of execution of the arithmetic operations in the BASIC expression:

   1 + 4*2^3/16 −(3 + 2)

   and hence predict the numeric answer. Decide whether this BASIC expression is a correct coding of the algebraic expression:

   $$\frac{1 + (4\times 2)^3}{16 - 3 + 2}$$

   Check your conclusion by manually evaluating this expression.

2. Indicate which of the following expressions:
   (i) are valid or invalid BASIC expressions, and
   (ii) accurately code the associated algebraic expressions:

	BASIC *expression*	*Algebraic expression*
(a)	(A + B)/C	$\frac{a + b}{c}$
(b)	(A + 2B)/C	$\frac{1 + 2b}{c}$
(c)	A*X*X + ((2*B*X) + C)	$ax^2 + 2bx + c$
(d)	1/ 1 + (3*Y)^0.5	$\frac{1}{1 + \sqrt{3Y}}$

3. Write out BASIC arithmetic expressions to perform the following arithmetic operations:
   (a) $(1.01 + 3.06 \times 10^{-12})^{1/2}$
   (b) $\frac{3 + 10.6}{8.9}$
   (c) $\frac{2}{1 + 3^2}$

   (d) $4/3\pi r^3$ ($\pi = 3.14$)
   (e) $1 + \frac{A}{1} + \frac{A^2}{2^2} + \frac{A^3}{3^3}$
   (f) $\frac{M_1 M_2}{M_1 + M_2}$
   (g) $\frac{2ab}{c + d^{0.38}}$
   (h) $\frac{5}{x + 2} - \frac{2(x - 8)}{x^2 - 4}$

4. *Using the* PRINT *statement, execute your* BASIC *expressions for 3(a), (b) and (c) above. Check that your results are in agreement with what is implied in the algebraic form.*

Note that all arithmetic operations are performed as if all numbers and variables are real. If arithmetic operations are performed which result in numbers greater than the upper limit of the range of real numbers, then the error message ?OVERFLOW will appear on the screen. Also division by zero will give the message ? DIVISION BY ZERO. To recover from such an error message, ensure that the cursor has moved to the start of a new line and then enter in a new statement. If the result is less than the lower range of real numbers, then the result is set to zero and execution is continued with *no error message.*

Ex. 2.5

Execute the following statements on your computer. Account for the output on the screen.

i. `PRINT 1/3`
ii. `PRINT 1/0`
iii. `PRINT 1/0.1E-38`

2.5 Assignment Statements

To assign a specific numeric or string value to a given variable, use an *assignment* statement:

variable name = numeric or string value
or variable name = arithmetic expression

The = in this context may have a subtle and important difference from its normal use in algebra. It does *not necessarily* mean that the two sides of either of these forms of the assignment statement are equal. Instead it implies that the present contents of the memory location reserved for that particular variable are to be replaced by the value of the right hand expression. Thus the = operation is more properly called a *replacement operator.* The assign-

ment statement N = N+1 which is algebraic nonsense, is perfectly valid in BASIC. It implies that 1 is to be added on to whatever value currently exists in the memory location associated with the variable name N. In other words N is replaced by N+1.

The variable on the left hand side of an assignment statement is normally of the same type as the variable or constant (number) on the right hand side. However, they *can* be different and this sometimes leads to unexpected effects. For example, in the assignment statement:

```
X% = 1.23
```

the variable X% is of type integer (as specified by % sign) but the 1.23 is a real (fixed point) number. For such statements in Applesoft BASIC, all positive real numbers are rounded down to the nearest whole number, while all negative real numbers are rounded down to the nearest (negative) whole number. Thus in the above example, X% would be assigned the value of 1 while execution of the assignment statement:

```
X% = -0.96
```

would result in X% taking the value of −1. Clearly no similar problems arise when integer variables or numbers are assigned to real variables. Such truncation effects, along with those associated with the accuracy with which real variables are stored, can lead to inaccuracies in numeric calculations particularly those involving many steps.

The assignment of string variables is more complicated in that the collection of literal data characters is placed within quotation marks and the complete entity is called a *string*. For example A$ = "THIS IS A STRING" will assign the literal data THIS IS A STRING to the contents of the memory location associated with the string variable name A$.

Some examples of valid and invalid assignment statements are as follows:

Valid	*Invalid*
N = 1.0E12	A + B = C + D
A = B−XY	(Only one variable name may
CT = DOG + (BIRD)^0.5	appear on the left.)
B$ = "PI = 3.14"	B$ = 1.318
	(Right hand expression is not
	a character string.)

In some other forms of BASIC, the word LET must be inserted before the variable name e.g. LET A=1. However this feature is optional in Applesoft BASIC and is generally omitted for brevity.

Ex. 2.6

Which of the following assignment statements are valid in Applesoft BASIC?

(a) X = A*X + Y
(b) ANSWER = MEAN/N
(c) I = 0.51 B ^ 2
(d) X% = 2/3 − 1
(e) DATE = 13/6/1904
(f) DAY$ = MONDAY
(g) AUGUST = MONTH$
(h) COUNTRY$ = "NEW ZEALAND"
(i) Y = 2 ^ 0.5
(j) A + 3 = B ^ 2

Predict the value assigned to X% after execution of statement (d). Comment on whether you would expect it to be different from that assigned to X from execution of the assignment statement:

```
X = 2/3 − 1
```

2.6 Error Messages

By this stage you may well have encountered different types of error messages. Although the wording of these is designed to be as self explanatory as possible, you are referred to either Appendix E of *The Applesoft Tutorial* or Appendix C of ref. [2] for further explanation.

Chapter 3

Running a BASIC Program

3.1 Execution Modes

So far in our discussion of simple BASIC statements, we have executed each PRINT statement immediately on completion of keying it in, by pressing RETURN. This is called the *immediate* or *direct execution mode*.

Obviously single line statements are very limiting in what they can achieve. It would clearly be advantageous to store in the computer memory a whole series of instructions (statements) which is collectively called a *program*. A command would be given to the Apple to begin running (execution of) one of these instructions and then to automatically work through in a systematic fashion under user control, some or all of the remaining instructions. This is called the *deferred execution* or *programming mode* and is by far and away the most common and useful of the two possible execution modes in Applesoft BASIC.

This mode requires the assignment of a statement number *at the beginning of every statement,* e.g.

```
10 A = 2
20  PRINT A
```

The only exception occurs when more than one statement appears in the same line, in which case only the first statement in the line has to be numbered. Multistatement lines will be discussed more fully in Section 6.2. *In discussing the form of any new Applesoft* BASIC *statements in the remainder of the text, we will assume that they are always preceded by a statement number.* The statement number can be any positive integer in the range from 0 to 63999.

The use of a statement number to label an instruction will change the way in which the Apple reads in and executes the statement. The statement number is keyed in after the Applesoft prompt]. There can be any number of blank spaces, including none at all, between the prompt position and the first digit of the statement number. The statement itself is then keyed in and again there can be any number of blank spaces between the statement number and the statement. For ease of reading it is usual to key the statement number in immediately after the prompt and leave at least one blank space between it and the statement.

Ex. 3.1

Key NEW RETURN *to clear memory of any previous* BASIC *statements and then enter the statement:*

```
20  PRINT A
```


If the statement number and statement are longer than the right hand limit of the first line, the cursor (■) will automatically move down to the beginning of the next line. The remainder of the statement should then be keyed in. It is not necessary to repeat the statement number or include any other symbol to indicate continuation. Up to 6½ lines (or 255 characters) can be used for a single statement. A beep is heard on entering the 248-th character.

Upon completion of keying in the statement, press RETURN to indicate to the computer that the statement is complete as you have been doing already in the immediate execution mode. However, notice now that the Apple will not immediately execute this statement by printing out the value of the variable A. Instead it is stored in memory and the prompt and cursor appear at the start of a new line ready for entry of another BASIC statement or command.

Ex. 3.2

Continue entering this program by keying the statement:

```
10 A = 2
```

When all the statements in a program have been keyed in or indeed after completion of one or any number of statements, you may find out what statements have been stored in memory so far by keying the command:

LIST RETURN

after the appearance of the prompt].

Ex. 3.3

Key LIST RETURN

The screen should now appear as:

```
120 PRINT A

]10 A=2

]LIST

10 A = 2
20  PRINT A

] ■
```

Notice that although you keyed in stmt. 20 before stmt. 10, the computer always lists (and indeed generally executes) statements in ascending order of statement number. Also that irrespective of the number of blank spaces that

you may have keyed in between the letters, the computer begins the statement number in column 1 and separates it by 1 or 2 spaces from the statement itself.

At this stage, any errors that might have been in any statement can be corrected by keying in that statement number again, followed by the correct statement. This new statement will obliterate the old (incorrect) one providing the two statement numbers are the same. If they are not, then the new statement will be entered into memory and the statement it is supposed to replace will also remain.

Ex. 3.4

Replace the existing stmt. 10 by a new statement which assigns the value of 3 to A. *List the amended program.*

Imagine you now want to again change the value of A *this time to 4, but you inadvertantly miskey the statement number and instead key:*

```
1 A = 4
```

List the new program and account for any changes.

To completely *delete* a statement, key in a *null* line i.e. just the statement number only, and then RETURN.

Ex. 3.5

Delete stmt. 1 in your program.

Alternatively the command DEL n,m can be used to delete all lines with *line* numbers from n to m inclusive. Further details of the DEL command are given on p. 49 of ref. [2]. New statements can be inserted between two successive statements between n_1 and n_2 by giving them statement numbers which are between n_1 and n_2. Obviously one cannot insert any statement between two statements with consecutive integer numbers e.g. 10 and 11. Hence it is good programing practice to write the statement numbers in your initial program in steps of 10 beginning at 10. This allows up to 9 new statements to be inserted between any two original statements. More than this number of new statements can of course be inserted by changing the statement numbers of either or both of the original statements.

Other methods for correcting statements are discussed in Appendix II.

Ex. 3.6

Insert the statement A = 1/A *between stmts. 10 and 20 in your program by choosing any statement number between 10 and 20.*

The LIST command will cause the whole program to be listed on the screen. If it is more than 40 lines long, then only the last 40 lines will remain on the screen. To stop the listing at some intermediate point, key CTRL S. To continue the listing, press any key. Alternatively you may wish to list only a single statement by using the command: LIST followed by the statement (line) number. Or you can list all statements with statement (line) numbers between the two limits n and m by keying the command: LIST n,m. Further details of the LIST command and its other features are detailed on p.48 of ref. [2].

Having established that your program is correct, it can now be run (*executed*) by keying the command:

RUN RETURN

after the prompt]. This will cause the Apple to begin running at the lowest numbered statement and then to progressively execute subsequent statements in order of ascending statement number. Exceptions can occur when statements are encountered within the program which direct execution to some other statement number. This important phenomenon called *branching* will be covered in the next chapter. If a statement number is specified after the RUN command and before RETURN is keyed, then execution will begin at this statement number rather than at the beginning.

Ex. 3.7

Execute your program and note the output on the screen. Change the program so that the quantity 1/4 is calculated instead of the present 1/3.

The Apple will continue running until it finishes all the statements or alternatively it encounters an error. In the latter case it will often print out an error message and then stop. If the prompt] appears on a new line, and if you can identify the cause of the error, then corrections can be keyed in and the execution commenced at the *beginning* of the program again using the RUN command. In this way, one can progressively and quickly work through the whole program checking for satisfactory execution (a procedure referred to as *debugging* a program).

Ex. 3.8

Insert the following statement into your program.

```
10 A = 0
```

List and then run your program. Account for the output and change the program for A = 5.

After successful execution, the prompt should appear and any new Applesoft command can now be entered into the computer. The program

will remain in memory and if a new program is keyed in, only those old statements with the same statement numbers as the new ones will be removed. This can lead to unexpected results since execution of what you imagine is solely the new program may in fact also include execution of statements in the previous program.

Ex. 3.9

Enter the following new program:

```
10 A = 5
30 PRINT A ^ 3
```

List and run. Account for the two numbers that are printed as output whereas you would expect only one number (125).

To get around this potential problem, before keying in any new programs, use the command:

NEW RETURN

which will clear the memory of all program steps. This is done automatically by the Apple when it is first powered up and so it was not necessary to use this command when we began entering our first program.

3.2 INPUT Statement

The program:

```
10 A = 1
15 A = A ^ 2
20 PRINT A
```

is limited by the fact that if we wish to calculate and print out the value of A^2 for any number other than $A = 1$, we must enter a new stmt. 10 and then re-execute. It would be advantageous to generalise even this simple program by writing it in such a way that it is written for any value of A which can then be input by the user at the time of execution.

This can be achieved by using the INPUT statement which in its simplest form is

INPUT variable name

When the computer encounters this statement during execution, a question mark will appear on the screen which indicates that the computer requires entry of data. This data is keyed in followed by RETURN and the computer will then continue normal execution of the program.

Ex. 3.10

Enter (remember to first type NEW RETURN *before keying in the program) the following statements:*

```
10 INPUT A
20 PRINT 1 / A
```

Test the program for A = *2, 3 and 4.*

3.3 Remarks

It is often desirable to include remarks or comments at various points in a program in order to provide information on such matters as what is being calculated in what part of the program, and the form of the input and output. In BASIC this facility is provided by the REM statement (REMark):

REM contents of remark

One or more blanks or none at all, may be placed between the statement number and REM or between REM and the remark. Any characters whatsoever (including all blanks) can be placed in the remark section. Up to 255 characters may be used and of course this will often entail using more than one line. Again as for other statements, it is not necessary to repeat the REM for continuation of the remark contents at the beginning of new lines. REM statements can be placed anywhere in the program and are completely ignored by the computer during execution. An example of the REM statement is:

```
120 REM **THIS IS AN EXAMPLE !
```

As the use of BASIC develops and becomes more widespread, attempts are being made to standardise the layout of programs. These often involve the use of REM statements such as at the beginning of the program for giving information on the nature of the program as in the following example:

```
10  REM  ** PROGRAM TITLE : FARM COSTS
20  REM  **
30  REM  ** WRITTEN BY J. SMITH
40  REM  ** DATE WRITTEN SEP.1 ST,1982
50  REM  **
```

Although not necessary for the successful *execution* of a program, such information is very helpful to identify a program and to provide other useful information about it. You are encouraged from the outset to develop the habit of using liberal numbers of REM statements throughout your programs.

The reader is referred to ref. [8] for further comments on the use of REM statements and other helpful methods of improving the layout and readability of BASIC programs.

3.4 Stopping and Ending a Program

Sometimes you may wish to stop the execution of a program at one or more places within it. This can be achieved by using the STOP statement anywhere in the program:

STOP

When the computer encounters this statement it will stop all execution and print out the message:

] BREAK IN statement no XXX

where XXX is the stmt. no. of the STOP statement.

Execution can be restarted at the next instruction after such a STOP statement by keying the command:

CONT

This command will also recommence a program halted by CTRL C or an END statement (see below). It is often useful to insert several STOP statements at various points in a new program during its development in order to tell how far the computer has progressed in the execution of the program.

All programs should have as a final statement, the END statement:

END

When the computer encounters this statement it assumes all execution is to cease. No message such as for the STOP statement is printed out.

Ex. 3.11

Insert a STOP *statement between lines 10 and 20 and an* END *statement in your existing program. Now execute the complete program to obtain 1/A.*

With even these simple BASIC statements, you are now in a position to write programs which will solve a wide range of numerical problems. The following exercises will give you practice at writing, entering and executing such programs.

Ex. 3.12

Write multistatement programs incorporating INPUT and/or PRINT statements to perform the following calculations:

1. The values of 1/A and 1/B (A = 3.6, B = 6.8).
2. The sum of all integers between 1 and 6.
3. The area of a rectangle given the lengths of the two sides (10.8cm, 3.6cm).
4. The average speed of a car given the distance of the journey and the travelling time (328km, 5 hrs).
5. The volume of a sphere for a given radius as expressed by the formula $V = (4/3)\pi r^3$ (Set π = 3.14) (6.8cm).
6. The income tax payable on a given annual salary if the rate of tax is 41% ($25 600).

7. Perform the following conversions:
   (a) metres to (i) cm and (ii) mm (1.26 m)
   (b) pounds weight to kilograms (1-lb = 0.45 kg) (3.82-lb)
   (c) degrees fahrenheit (F) to degrees centigrade (C) according to the formula
   $$C = \frac{5}{9}(F - 32) \qquad (98.1F)$$
8. Calculate the change from $10 for purchase of an item costing less than an equal to this amount ($6.83).
9. A farmer decides to build a fence with one post every 1.25 m. Calculate how many posts and the total length of wire that he will need for a fence of 5 strands (wires) and 460 m long. (Don't forget that it is necessary to have a post at both ends!).
10. The wavelength (λ) and frequency (υ) of electromagnetic radiation are related by the expression:
    $$\lambda \upsilon = c$$
    where c is the speed of light (3×10^8 m s^{-1}). Calculate the frequency given the wavelength (254×10^{-9} m).
11. Round a decimal number off to its nearest whole number (67.23, 6.51).
12. The temperature (T) of n moles of an ideal gas at pressure (P), and volume (V) given by the formula:
    $$T = \frac{PV}{nR}$$
    where R is a constant = 8.14 ($P = 101 \times 10^3$, $V = 1.2 \times 10^{-3}$, n = 0.056).
13. The distance (s) travelled by a projectile in a given time (t) according to the formula:
    $$s = ut + \tfrac{1}{2}at^2$$
    where u is the initial velocity and a is the acceleration
    ($u = 300$ m s^{-1}, $a = 9.8$ m s^{-2}, $t = 250$ s).
14. The nth term in an arithmetic progression according to the formula:
    $$\text{nth term} = a + (n - 1)d$$
    where a is the value of the first term and d is the constant difference (6th term in the series 1 3 5 7 . . .).
15. The sum (s) of the first n terms in the series discussed in problem (14) above from the formula:
    $$s = \frac{n}{2}\{\, 2a(n - 1)d \quad \} \qquad (n = 20)$$
16. Calculate the mean (m) and standard deviation (S_d)
    $$S_d = \sqrt{\frac{\Sigma (x_i - m)^2}{n}}$$
    of three numbers (3.65, 3.52, 3.48).

Key each of your programs into the Apple and test with the sample input data given in brackets at the end of each problem. Remember to key NEW *before entering each new program.*

Chapter 4

Going, Deciding and Looping

4.1 GOTO Statement

There are many situations in programs where it is necessary to direct execution away from the next sequential statement to some other statement in the program. Alternatively, you may wish to repeat a program over and over again. Both these functions can be achieved using several different BASIC statements and one of these is the GOTO statement:

GOTO stmt. no. n

An example of the use of this statement might be the simple program that we developed in the previous chapter viz.:

```
10  INPUT A
20  PRINT 1 / A
30  END
```

If we wished to repeat this program for new data, we could key the RUN command for each new piece. Alternatively, we could save all the time involved in keying in the RUN command to recommence execution for each piece of data, by inserting a GOTO 10 statement between the PRINT and END statements. This will cause the program to *loop* or *branch* back to the start again and input another value for A. Thus the program stays continually in execution mode and indeed will never encounter the END statement which would signify the end of execution. The only way that one could escape from this execution mode (apart from turning the mains power off and then on again which would cause you to lose the program from memory) would be to use CTRL C or RESET (as described on p. 9, Ch. 1).

Ex. 4.1

Enter the program:

```
10  INPUT A
20  PRINT 1 / A
30  GOTO 10
40  END
```

and execute for A = 2.3, 4.6 and 6.8. Unless the input is zero or a character, this program will keep looping within itself for as long as the power is turned on for the computer. To cease execution, key CTRL C RETURN *and the Applesoft prompt should appear on the screen, indicating the end of execution.*

4.2 Decision Statements

An important part of any computer program is the ability to be able to vary the instructions that are executed, depending upon certain conditions or decisions that arise within the program.

This feature is accomplished in Applesoft BASIC using the IF statement which has the general forms:

i. IF condition THEN instruction.
ii. IF condition GOTO stmt. no. n.
iii. IF condition THEN stmt. no. n.

The condition is a type of logical expression which has the value of 1 if the expression is true and 0 if it is false. If the condition is *true*, then in form (i) the instruction is executed, while in forms (ii) and (iii) the program passes execution on to the instruction labelled by stmt. no. n. If the condition is *false*, then none of these actions occur and instead the next instruction on the line after the IF statement is executed, e.g.

```
100  IF (A - B) ^ 0.5 > 2 THEN A = B
200  IF A + B <  > X + Y THEN  PRINT A + B
```

The instruction executed after the IF/THEN statement can itself be another IF/THEN statement, e.g.

```
150  IF A > B THEN  IF A = 0 THEN 100
```

This combination of IF/THEN statements can often be rewritten more clearly in terms of the logical operators AND, OR and NOT which can be used to combine two or more condition expressions into a single one. These operators obey the normal rules of logic as follows:

AND: If the conditions are both true then the resulting single condition is true (1).
If one condition is true and the other false, or both are false, then the one resulting single condition is false (0).

OR: If one condition or both conditions are true, the resulting single condition is true (1).
If both conditions are false, the resulting single condition is false (0).

NOT: Changes the logic of an expression to the opposite state.

The AND and OR logical operators are useful for combining two or more conditions into one statement. Thus the above combined IF/THEN stmt. 150 could be more clearly written as:

```
150  IF A > B AND A = 0 THEN 100
```

In our program to calculate 1/A, if we only wish to calculate this quantity when A > 0 and < 100 then we could include both conditions in an IF statement which in turn could be combined with the PRINT statement to give the program:

```
10  INPUT A
20  IF A > 0 AND A < 100 THEN  PRINT 1 / A
30  GOTO 10
40  END
```

Note that it is not necessary to put brackets around each of the conditions in the IF statement, i.e. (A > O) AND (A < 100). However, brackets can be used particularly if they lead to clarity in either expression.

As an example to illustrate the use of the IF statement, consider our program:

```
10  INPUT A
20  PRINT 1 / A
30  GOTO 10
40  END
```

We have already noted in Ex. 3.8 that if we accidentally input zero as the value for A, then we get the error message ? DIVISION BY ZERO IN 20 printed on the screen. We could avoid this situation by inserting the following IF statement between stmts. 10 and 20.

```
15  IF A = 0 THEN 10
```

This will have the effect of bypassing statement 20 if A = 0 and instead returning to input a new value for A. If A ≠ 0 then the statement on the next line (stmt. 20) is executed.

Ex. 4.2

Enter and execute the above program with the IF *statement.*

The composition of the condition part of the IF statement requires further consideration. The equality sign (=) in stmt. 15 is an example of *a relational operator*. Note that its function in this context is as a true test of equality as distinct from its use as an assignment operator (p. 19). Other relational operators include:

<	less than
>	greater than
<> or ><	not equal to
>= *or* =>	greater than or equal to
<= or =<	less than or equal to

Using these operators one may construct a condition statement to compare all possible conditions of a number and an arithmetic expression, or two arithmetic expressions.

String expressions and variables may also be compared in this way.

Ex. 4.3

1. Comment on whether the following IF statements are valid forms in Applesoft BASIC:

   i. `IF A > B GOTO 230`
   ii. `IF X = 3 OR Y2 = 3 THEN 500`
   iii. `IF H <  OR  > 3 THEN I = I + 1`
   iv. `IF H AND X > 3 GOTO 20`

2. Predict the output from execution of the program

```
10  INPUT A
20  INPUT B
30  IF A > B THEN B = A
40  PRINT A
50  PRINT B
```

   for (i) A = 10 and B = 20 and (ii) A = 20 and B = 10.

   Check your predictions by entering and executing the program for this data.

3. Write a program to print out the largest value of three numbers which are entered one at a time from the keyboard. *Execute the program for the three numbers 3.1, −2.8, 0.*

4. Modify the program in (3) to also print out the smallest value in the series. *Execute for the above series of numbers.*

5. Write a program to calculate the sum, and sum of the square of a series of numbers entered singly from the keyboard. On encountering a value of zero for one of these numbers, the program should print out the values of both sums and then be ready to repeat the calculation for another series of numbers. *Test on the series of numbers given above for (3).*

6. A person deposits a certain amount of money in a bank, then makes a number of withdrawals, and after each withdrawal would like to know the following information:

   i. the amount of money remaining in the bank,
   ii. whether this remaining amount is less than 10% of the original deposit.

   Write a program which will input the initial amount and then perform the above tasks.

   Execute the program for the test data:

   initial amount *$200*
   withdrawals *$101.96, $22.48, $0.96, $36.79, $27.46*

7. The competitors in an athletic event are attempting to complete a course under a given qualifying time. Write a program which will accept as decimal fractions this qualifying time, a common start time and the individual finishing times. For each competitor, it should calculate the elapsed time and whether or not it is below the qualifying time.

Enter and test your program with the data:

Qualifying time	*40.60*			
Start Time	*2.48*			
Finishing Times	*41.48,*	*42.05,*	*43.16,*	*44.50*

8. Write a program to calculate the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

given a, b and c as input. Use the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Use an IF statement to avoid the possibility that $b^2 - 4ac$ may be negative and hence $\sqrt{b^2 - 4ac}$ cannot be calculated.

Test for the equations $x^2 - x - 6 = 0$ *and* $x^2 - 3x + 3 = 0$.

The assignment in Applesoft BASIC of the numeric value of 1 to *true* statements and 0 to *false* statements means that these values can be treated as ordinary numbers and combined in arithmetic-like statements. For example, execution of the statement:

```
PRINT A = 3
```

will result in the value 1 appearing on the screen if A = 3, and 0 if A<>3.

This type of statement is sometimes useful for indicating whether a condition is true or false. The condition expression often involves the AND and OR logical operators.

e.g.

```
10 A = 1
20 B = 10
30  PRINT (A = 0 AND B = 10)
```

Ex. 4.4

Predict the output from execution of this program. If the AND operator in stmt. 30 was replaced by OR, would the output change?

Check your predictions by entering and executing the two programs.

4.3 Loops and the FOR/NEXT Statements

A program to calculate the sum of all integer numbers between 1 and 10 could be written in its simplest form, as the single statement

```
10  PRINT 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

Obviously such a statement would become impossibly long for the sum of numbers say between 1 and 100. To allow for such situations, we can use the IF and GOTO statements to write a general program for calculating the sum from 1 to any upper limit N which is input to the program:


```
10  INPUT N
20 SUM = 0
30 I = 1
40 SUM = SUM + I
50 I = I + 1
60  IF I < = N GOTO 40
70  PRINT SUM
80  END
```

You should examine this program carefully and trace through the steps that are executed for several cycles because it illustrates a number of important features which are common to any program involving loops.

The extent of the loop in this program is determined by the IF statement and extends from stmt. 40 to stmt. 60. *Before* entering this loop the variable SUM which will be used to store the running sum, is set to 0. The loop counter I is initialised to 1 corresponding to the first time through the loop. On entering the loop for the first time, stmt. 40 will cause the current value of I to be added to SUM i.e. SUM = 0+1 = 1. I is then incremented by 1 to give a value of 2. A check is made to see if this new value exceeds the upper limit N which has been entered in the first statement. If I does not exceed N, then execution returns to the beginning of the loop and the procedure is repeated until I exceeds N. Execution within the loop is then terminated and an exit is made to stmt. 70 which prints out the value of SUM.

The important aspects in this example which should be noted are the concept of looping, incrementing of a loop counter, and the initialisation before entering the loop of running variables used within the loop. Most programs of any complexity will involve these features.

Stmts. 30, 50 and 60 can be replaced by FOR/NEXT statements. These are concerned with loops in BASIC and take the general form

```
FOR variable name = initial value TO final value
  .    .
  .    .
NEXT variable name
```

The same variable name must be used in both statements. Variable names or arithmetic expressions which give a numeric value on evaluation can also be used for the initial and final values in the FOR statement. Any number of any type of statement can be placed between these two statement types.

Our program for calculating the sum between 1 and N can be rewritten using FOR/NEXT statements as follows:

```
10  INPUT N
20 SUM = 0
30  FOR I = 1 TO N
40 SUM = SUM + I
50  NEXT I
60  PRINT SUM
70  END
```

Compared to the earlier form of this program, only one statement has been saved by using the FOR/NEXT statements. However, the program is now clearer to follow and during program writing, one can avoid having to think carefully about incrementing the loop counter and the logic required in the statement at the end of the loop.

In this example, I is automatically incremented by 1 at the *end* of the loop whenever the NEXT I statement is executed. A test is also automatically made at this time to see if I has now exceeded the upper limit N. If it has not, then execution passes back to stmt. 40. If it does exceed N then execution passes to the next statement *after* NEXT I.

The *default value* i.e. the value the program will assume in the absence of any specification, for the increment is always 1. This can be changed to any other value including non-integer or negative values by extending the FOR statement to include a STEP term as follows:

FOR variable name = initial value TO final value STEP increment

Thus, if in our present example, we wanted to only count the sum of odd numbers between 1 and N, then we could replace stmt. no. 30 with:

```
30 FOR I = 1 TO N STEP 2
```

which would cause the loop counter I to take only the values 1, 3, 5 . . . . N (or N − 1 if N is even).

If the increment is a negative quantity i.e. a decrement, the loop counter will decrease and the initial value will now be larger than the final value. Because the test of whether the loop counter exceeds the final value occurs at the *end* of the loop, a loop defined for FOR/NEXT statements will always be executed at least *once* irrespective of the relative magnitude of the initial, final and increment values.

Ex. 4.5

1. Modify the above program to count the sum of all even numbers between 1 and N. Include extra statements to print out the value of I and SUM each time the loop is executed.

   Enter and run the program for N = 5.

2. Write a program to calculate the sum of all numbers between two limits given as input.

   Enter and run the program for the range 3 to 8.

3. A program which will allow you to explore the use of various relative magnitudes and signs for the initial (N), final (F) and increment (I) values in a FOR statement is

```
10 INPUT N
20 INPUT F
30 INPUT I
40 FOR X = N TO F STEP I
50 PRINT X
60 NEXT X
70 GOTO 10
80 END
```

   Enter this program into the computer and execute for a range of combinations of N, F *and* I *including*

   (−2, +2, 1) (+2, −2, −1) (−2, +3, 0.5)
   (−3, −3, 1) (+2, +2, 0)

4. Write a program to calculate the factorial function:

   $$N\,! = N \times (N-1) \times (N-2) \times \ldots\ldots 1$$

   Execute for N = 4.

There are many occasions when you may wish to execute a loop within one or more other loops. This procedure is called *nesting* of loops and Applesoft BASIC allows for up to 10 levels of nested loops. Each loop must have its own FOR and NEXT statements.

A simple example of the use of nested loops would be in a program to calculate the sum of N terms of the form:

$$1 + (1+2) + (1+2+3) + \ldots (1+2+ \ldots . N)$$

Two summations are involved in this calculation and these could be performed in a program involving two nested loops:

```
10 INPUT N
20 SUM = 0
30 FOR I = 1 TO N
40 FOR J = 1 TO I
50 SUM = SUM + J
60 NEXT J
70 NEXT I
80 PRINT SUM
90 END
```

Note the inner J loop with its own NEXT statement which is completely enclosed within the outer I loop (also with its own NEXT statement *after* the NEXT J statement). It is important to ensure that the end of an inner loop is *never* placed outside the end of an outer loop. If this occurs, the Apple will be confused and the ERROR message ? NEXT WITHOUT FOR ERROR IN stmt. no. will appear on execution. In this example for each value of I, the inner loop is executed I times.

Ex. 4.6

1. *Enter and run the above program for N = 3. Modify the program to include statements to print out* I *and* J *within the inner* J *loop and note the order of these values for execution when N = 4.*

2. Predict the output from execution of the program:

```
10 N = 3
20 M = 2
30 FOR I = 1 TO N STEP 2
40 FOR J = 1 TO M
50 PRINT I
60 PRINT J
70 NEXT J
80 NEXT I
90 END
```

Check your predictions by entering and executing this program.

3. Write a program using nested loops to calculate values of the expression $(a+b)^c$ for all possible combinations of a, b, c in the range 1 to N. Include statements to also print out the values of each combination of a, b and c. Predict the expected output for N = 2.

Check by execution of the program for this value.

4.4 ON/GOTO Statement

A variation of the IF/THEN type conditional statement discussed in Section 4.2 is the statement:

ON n GOTO stmt. no.(1), stmt. no.(2), ...

where n is an integer constant, variable, or an arithmetic expression which gives an integer on evaluation. This statement causes a branch in execution to the statement specified by the n-th item in the list of statement numbers following GOTO. Thus if n = 1, execution passes to stmt. (1) and so on, e.g.

```
10 ON X + 1 GOTO 100,110,120
```

If n = 0 or is greater than the number of listed alternative statement numbers, no branching occurs and execution passes to the next statement after the ON/GOTO statement.

This type of condition statement is particularly useful for specifying an option from a list. For example, if one wanted to branch to one of three different calculation steps beginning with the stmt. nos. 110, 120 or 130 depending on the wish of the user, one might use the statements:

```
10 INPUT N
20 ON N GOTO 110,120,130
110 . . . .
    . . . .
119 GOTO 200
120 . . . .
    . . . .
129 GOTO 200
```



```
130 . . . .
    . . . .
200 . . . .
```

The value of N would be input by the user and could take the values 1, 2 or 3 depending on which calculation is required. Note in this example how at the end of each of the first two sets of calculations, a GOTO statement is used to avoid automatically executing the next set of calculations.

Ex. 4.7

Write a program incorporating an ON/GOTO statement to calculate the quantities n+1, n or 1-n depending on whether n = 1, 2 or 3 respectively.

Enter and execute this program.

4.5 Flowcharts (Logic Diagrams)

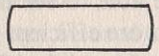
At this stage in learning BASIC, you will probably have realised that the stages you go through in writing a program follow the sequence:

1. Define the problem.
2. Decide on what calculation steps are needed and how you will go about these.
3. Decide what BASIC statements will perform these steps.
4. Combine these statements into a logical sequence to form a BASIC program.

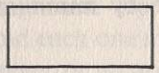
For some problems, steps (1), (2) and (3) can all be done in the mind and the BASIC program written out directly. However, in most other situations, and certainly for problems involving any branching or looping complexity, it is necessary to write down some or all of the calculation steps before deciding on the required BASIC statements. To assist in this step, a procedure has been developed which is known as *flow charting*. It is simply a graphical means of showing the sequence of steps of the program.

Special graphical symbols are used for the various types of steps. Some of the more common ones include:

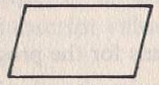
Begin or end



Assignment statements



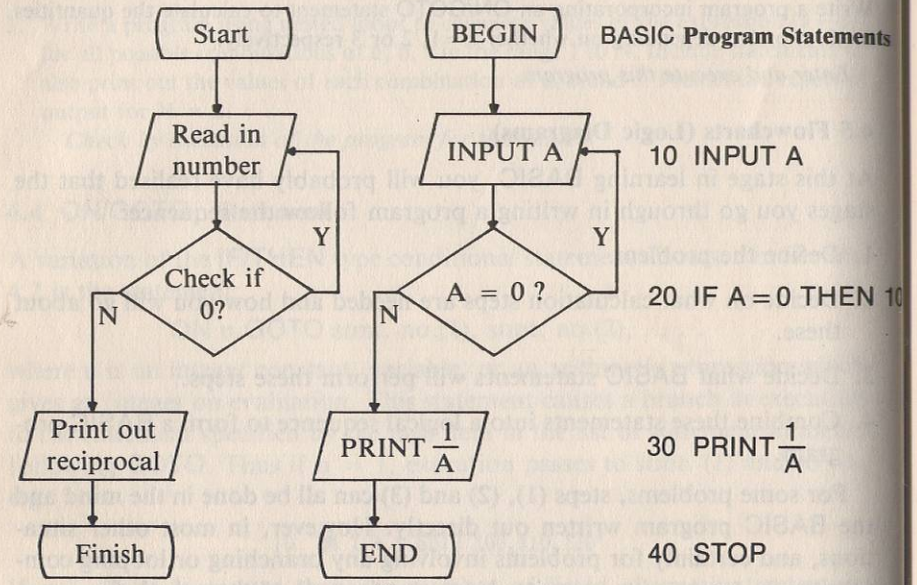
Input/Output



Decision and Branching steps

Within the assignment and input/output symbols one can write any number of instructions of that type. These can range from exact BASIC statements to very non-mathematical verbal descriptions. These symbols are connected by arrows to describe the logial sequence of execution of the steps.

An example of these two extreme types of flow chart and the resulting BASIC program that might be derived from either of them, is for the problem that we considered earlier in this chapter. This involved the input of a number, followed by calculation and printing of its reciprocal if the number was not zero:



(a) (b) (c)

Note how the symbols Y (YES or TRUE) and N (NO or FALSE) are attached to the decision symbol.

You are encouraged to develop the habit of drawing up a flow chart before writing down specific BASIC statements for solving a problem. Not only will you be able to see more clearly branching and looping situations but you will be more efficient and, in the long run, save time in the program debugging stage by minimising errors in logic and the use of incorrect statements.

Ex. 4.8

Draw flow diagrams for the programs described in:

i. Ex. 4.3.3
ii. Ex. 4.5.4
iii. Ex. 4.3.8

Chapter 5

Arrays, Lists and Character Strings

5.1 Arrays and the DIM Statement

There will be many occasions when you will want to perform calculations on lists or tables of data. Different variable names can be given to each item of data but clearly this can become a tedious task and lead to many repetitive statements for any more than a few items. For example, a program to input three numbers and calculate their mean and standard deviation might be of the form:

```
10 SUM = 0
20  INPUT A1
30 SUM = SUM + A1
40  INPUT A2
50 SUM = SUM + A2
60  INPUT A3
70 SUM = SUM + A3
80 MEAN = SUM / 3
90 MSD = ((MEAN - A1) ^ 2 + (MEAN - A2) ^ 2 + (MEAN - A3) ^ 2) / 3
100 SD =  SQR (MSD)
110  PRINT MEAN
120  PRINT SD
130  END
```

Note that because the value of each number must be retained after the calculation of the mean, in order to calculate the standard deviation, then they cannot be read in as a single variable name within a loop.

To extend this program to allow for even 10 numbers would clearly require the repetition of many statements of the type given in stmt. nos. 20 and 30. To get around this difficulty, almost all computer languages including Applesoft BASIC allow for the use of *subscripted variables* or *arrays* which have the general form:

variable name (subscript(s))

Any legitimate variable name (see Section 2.3) can be used and hence arrays may represent integer, real or character string data. Up to 88 different subscripts separated by commas are allowed and each one must be either an integer number, a variable representing an integer or an arithmetic expression which gives an integer on evaluation. A subscript value of zero but no negative integer values are allowed.

The most commonly used arrays involve a single subscript (1-dimensional array) as in a simple list, or two subscripts (2-dimensional array) as in a table.

Some examples of arrays expressions include:

```
A(1)

A(1,2,3)

A$(0,I+1)

A(100,0)
```

The general element in a list of 100 numbers might be referred to as A(I) with I taking any integer value between 1 and 100. For a table of numbers such as:

i \ j	1	2	3
1	b_{11}	b_{12}	b_{13}
2	b_{21}	b_{22}	b_{23}
3	b_{31}	b_{32}	b_{33}

then the general element in such a table could be labelled as B(I, J) where the I refers to the row position and the J the column position.

It is necessary to indicate to the Apple how much space to reserve for each array. This is done using the DIM statement which must precede the first use of an array in a program:

DIM array name 1 (maximum value of subscript(1)), . . .

where the values in brackets are called the *dimensions* of the array, e.g.

```
DIM A(100),B(3,3)
```

Note that in Applesoft BASIC, all arrays are assumed to begin with a subscript of 0 (regardless of whether you actually use that subscript value). Hence in the above DIM statement in fact space for 101 elements has been allowed for array A and 16 (4x4) spaces for array B. Therefore the DIM statement

```
DIM A(99),B(2,2)
```

could have been used to allow for 100 elements of a single subscripted array A, and 9 elements of a doubly subscripted array B. However, it is much easier in writing a DIM statement to simply use the maximum values of each subscript, provided there is sufficient memory space to allow for the extra elements. If no DIM statement is used or an array element is used before the corresponding DIM statement, a maximum value of 10 is assumed for the dimension.

The use of arrays and their subscripts lend themselves to manipulation within program loops and hence with FOR/NEXT statements. For example, the part of the above program involved in calculating just the mean value could be rewritten for N($\leq$100) numbers in terms of a 1-dimensional array as follows:

```
10  DIM A(100)
20 SUM = 0
30  INPUT N
40  FOR I = 1 TO N
50  INPUT A(I)
60 SUM = SUM + A(I)
70  NEXT I
80 MEAN = SUM / N
90  PRINT MEAN
100  END
```

The limitation on N being $\leq$100 only appears in the dimension given for the A array in the DIM stmt. 10 and so the program can be readily extended by simply increasing this dimension by an appropriate amount. The storage of any array obviously takes up memory space and so one should avoid using unrealistically large dimension values when they are not required. Note that it is good programming practice to dimension all arrays at the *beginning* of a program.

Ex. 5.1

1. Decide whether the following array expressions are valid in BASIC.
   i. `A(0)`
   ii. `A$(100,-1)`
   iii. `%PROFIT(2)`
   iv. `BDOG(I+1,J-1)`
2. Extend the above program written in terms of the array elements A, to calculate the standard deviation as well as the mean, and to print out these values along with the input data.

   Enter and execute for the 6 values of 12.60, 12.45, 12.72, 12.61, 12.68 and 12.52 as sample input.
3. Write a program to sort N ($\leq$100) numbers given as input, into ascending order of magnitude.

   Enter and execute for the 6 values input in the sequence 6, −3.2, 0, 2, 10, −8.
4. Write a program which will input two lists of N numbers each. It should then print out any values which are common to the two lists.

   Enter and execute the program for the sample data
   List 1 *1.6, 2, 1, 5, −3.8, 16.2*
   List 2 *0.0, 1.8, 3, 2, −10, 1.6*

5. Write a program to input the elements of the table:

3	6	8	9
2	3	6	2
4	3	2	8

Calculate and print out the sum of each row and each column, and the total sum.

Enter and execute the program for the above values.

Two dimensional arrays are also very useful for the manipulation of *matrices*. These involve elements arranged in rows and columns just like the table shown on p. 42. For example, to add together two matrices A and B of the same order (no. rows x no. columns) of n x m to give a resultant matrix C of the same order, one uses the formula

$$c_{ij} = a_{ij} + b_{ij}$$

where the i,j subscripts refer to the row and column positions of each matrix element. A simple program to input the elements of the two matrices and calculate their sum might take the form

```
10  DIM A(3,3),B(3,3)
20  INPUT N
30  INPUT M
40  FOR I = 1 TO N
50  FOR J = 1 TO M
60  INPUT A(I,J)
70  INPUT B(I,J)
80  PRINT A(I,J) + B(I,J)
90  NEXT J
100  NEXT I
110  END
```

In this program the matrix elements are input in the order of row followed by column i.e. A(1,1). . . .A(1,M), A(2,1). . . .A(2,M), A(N,1). . . .A(N,M). Some versions of BASIC have specific instructions to manipulate matrices e.g. MAT INV(A) will invert the matrix with array elements A. However none of these special matrix functions are available in Applesoft BASIC.

Ex. 5.2

1. *Execute the above program for the matrices*

$$A = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 2 & 3 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 7 & 8 \\ 7 & 5 & 0 \\ 8 & 0 & 6 \end{bmatrix}$$

2. A matrix A of order p x m can be multiplied by a matrix B of order m x q to give a resultant matrix C of order p x q such that the i,j-th element of C is given by the summation formula:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Write a computer program to input the elements of two such A and B matrices, multiply them together and then print out the elements of the resultant matrix C.

Enter the program and execute for the two square 3 x 3 matrices:

$$A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 3 \\ 2 & 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 0 \\ 1 & 0 & 2 \end{bmatrix}$$

Check your output values of c_{ij} by calculating these elements manually using the above formula.

5.3 Input and Output of Lists

So far we have inputted and outputted only one value of a constant or variable at a time using the INPUT and PRINT statements. This can be cumbersome for lists of data and so at this stage it is useful to know that both these input/output (I/O) statements can be extended to include lists as follows:

$$\left.\begin{matrix} \text{INPUT} \\ \text{PRINT} \end{matrix}\right\} \text{X, Y, ...}$$

where X, Y, etc., are variable names, including arrays, numeric constants, or character strings. For example:

```
INPUT A1(1),A(I),B(I+2)

PRINT "THIS IS AN EXAMPLE ",3.8,X(I)
```

The only limit on the number of items in the list is the limit of 255 characters for each statement. This facility can save the use of many INPUT and PRINT statements in a program. For example, in the program (p. 41) for which we inputted three values using three separate INPUT statements, these could more efficiently have been incorporated into just one INPUT statement:

```
INPUT A1,A2,A3
```

Upon execution of this statement, a question mark would appear on the screen and the three values would then be keyed in, separated by commas and in exactly the same order as in the program statement.

If insufficent entries have been keyed and terminated by RETURN, then two question marks ?? will appear on the screen indicating that additional entries are required. If too many entries are used, the message ? EXTRA IGNORED is printed and the program continues execution. Any entry for numeric input which is not real, integer, a comma or a colon will result in the error message ? REENTER and the INPUT instruction will be re-executed. Similar action occurs if just RETURN with no other entry is keyed in response to the question mark. Any spaces in any position are ignored for numeric input while any spaces between the ? and the first character of literal data are ignored for character input.

An INPUT statement can be interrupted using CTRL C RETURN only if it is the first character keyed in response to the question mark. However subsequent use of the CONT command does not recommence program execution and instead leads to the error message ? SYNTAX ERROR.

A related statement to INPUT is

GET variable name.

Upon encountering this command in a program the computer will wait until a *single character* is keyed in. This is not printed on the screen. A typical use of the GET command is a query to continue running a program:

```
10  PRINT "DO YOU WISH TO CONTINUE (Y/N)"
20  GET A$
30  IF A$ = "N" THEN  STOP
40  GOTO 50
```

Finally, a useful programming hint that will save time in entering a program into the computer, is that the word PRINT can be keyed as ? and the computer will translate this automatically during execution or listing of the program as the full PRINT wording.

5.4 Character Strings

There are many situations where one wishes to manipulate sequences of alphanumeric characters directly and not in the way that numeric characters are interpreted as numbers in calculations. Such a sequence of alphanumeric data is called a *character string* and, as briefly noted in our initial discussion of the PRINT statement on p. 12, is denoted by enclosing the characters within quotation marks, e.g.

```
"THIS IS A CHARACTER STRING"
"THE COST IS $1.80"
"12.3/4.8"
```

The difference between a number and a character string is clearly shown in the last example where if one was to attempt to execute the arithmetic statement

```
10 X=10↑2+"12.3/4.8"
```

an error message ? TYPE MISMATCH ERROR would occur because the "12.3/4.8" is now enclosed in quotation marks and hence is interpreted literally and not as a number. This leads to the alternative naming of character strings as *literal data*. This term refers to all the characters in a string but without the quotation marks.

Up to 255 legitimate characters can be used within the quotation marks. There is absolutely no restriction on the allowed combinations of characters

including the use of any reserved words (e.g. "GOTO" is an allowed character string). A character string containing no characters is also valid and is called the *null* string.

Character strings may be assigned to *string variable names* which follow the same rules as numeric variable names except that they are distinguished by the $ symbol after the variable name, e.g.

```
A$
STRING$
```

String variables may also be subscripted arrays, e.g.

```
A$(21)  Y$(I,J)
```

One can assign a character string to a string variable using a normal assignment statement, e.g.

```
10 A$ = "THIS IS A STRING ASSIGNMENT"
```

The default value for a string variable is the null string.

String variables can be manipulated in an IF statement involving the equality and inequality operators as in the following program:

```
10 A$ = "CAT"
20 B$ = "DOG"
30  IF A$ < > B$ THEN A$ = B$
40  PRINT A$
```

Ex. 5.3

1. What do you predict the output to be from execution of this program?

   Check your predictions by entering and executing the program.

2. Write a program to input a list of names and then check for the occurrence of a particular name.

   Enter and execute for appropriate input data.

It is common to incorporate character strings in INPUT and PRINT statements either on their own or in lists with other types of data, e.g.

```
10  INPUT "A = ";A
20  PRINT "TITLE IS ";A$
```

When a character string appears before the variable name in an INPUT statement then that literal data will appear without any prompt requesting entry of the value of that variable.

Note that a semicolon must be used in the INPUT statement to separate the character string from the variable name. If a comma is used then the error message ? SYNTAX ERROR IN stmt. no. will be displayed. Either a semicolon or comma may be used in the PRINT statement. However, the layout of the printed output will vary, depending upon which of these two characters is used, as will be discussed in Section 6.1.

Strings may be joined together (*concatenated*) by using the addition (+) operator, e.g.

```
10  INPUT "ENTER THE NAME OF ANY DAY ";A$
20  PRINT "THE DAY OF THE WEEK IS" + " " + A$
```

Ex. 5.4

1. *Enter and execute this program. What effect does concatenating " " in the middle of stmt. no. 20 have on the output?*

2. Write a program to print out all possible combinations of pairs of colours calculated from an input list of single colours. Be careful to avoid printing the same combination twice e.g. blue-red and red-blue.

   Enter and execute for the colours Blue, Red, Yellow, White and Black.

3. Write a program to input any 3 alphabetic letters and then using the concatenation operator, print out all possible 3 letter words that can be constructed from combinations of these letters.

   Enter and execute for the letters e, b and d.

4. Add statements to your mean and standard deviation program (Ex. 5.1.2) to input and print out a title as well as a description of the printed results.

   Execute the program for a title of EXP. 10 and the 5 values of 5.61, 5.55, 5.48, 5.53, 5.52 and 5.56.

5.5 DATA Statements

All the data used in the programs considered so far have been entered from the keyboad during execution of one or more INPUT statements. This method of data entry is fine if you wish to write a general program for any data and use values you do not wish to specify until execution time. However, this method of data entry does require keying in each number and hence can be quite slow.

If you know the values of at least some of your data at the time of writing your programs e.g., constant data in a calculation, then it is much faster to incorporate the data into the program at this stage rather than at execution time. This can be done using specific assignment statements but it is quicker and more efficient to use the READ and DATA statements:

READ variable name (1), variable name (2) . . . .
DATA list of real, integer or string values

Up to 255 characters can be incorporated in both statements.

Upon execution of the first READ statement in a program, the Apple looks for a list of data contained in a DATA statement and assigns the first element in the first DATA statement to variable name (1). It then assigns the second element in the DATA list to variable name (2) and so on until all the variable names in the READ statement are assigned values, e.g.

```
10  READ A1,A2,A3$
20  DATA  2.3,2.6,SHIP
```


A DATA statement may contain more values than are required to satisfy all the variables in the first READ statement. A pointer is set up by the computer to indicate to itself the position in the list of the last value that has been read. Execution of a second READ statement will cause the computer to assign the first variable name in this READ statement to the next value in the DATA statement *after* that indicated by the pointer.

Use of the statement

RESTORE

at any time will set the pointer back to the beginning of a DATA list.

The elements in a DATA list must agree exactly in type with those in the corresponding READ statements. There are generally no restrictions on the nature of the data except that no quotation marks may appear anywhere in a string. Similarly, colons and commas are not allowed in literal values appearing in a DATA list. Finally, DATA statements can be placed anywhere in a program and hence can be either before or after the related READ statement.

Ex. 5.5

1. In the program

```
10  DIM C(3)
20  READ A,X1,B$
30  READ C(1),C(2),C(3)
40  DATA  1.2,3,GRAPHICS,3.6,5.2,4.8
```

   deduce the values that will be assigned on execution to each of the variables in the READ statements.

   Check your deductions by including a statement to print the values of variables and then enter and execute the program.

2. Modify the program you developed in Ex. 5.4.4 for the calculation of the mean and standard deviation, to include the title and the data given for this exercise in a DATA statement.

3. A number ($\leq$100) of students have completed a multichoice examination consisting of 10 questions each with 5 options. Write a program to read in the correct answers using READ/DATA statements, to input the number of students and then to input their answers. Calculate and print out each student's mark and the most popular option for each question.

   Enter the program and execute for the following test data:

Question No.	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Correct Option	*4*	*3*	*3*	*2*	*4*	*1*	*5*	*3*	*2*	*4*
Student Answer (1)	*4*	*3*	*2*	*3*	*1*	*5*	*5*	*3*	*1*	*1*
(2)	*1*	*3*	*3*	*2*	*4*	*2*	*2*	*1*	*3*	*4*
(3)	*3*	*3*	*3*	*3*	*4*	*2*	*5*	*3*	*1*	*4*
(4)	*4*	*3*	*1*	*2*	*1*	*2*	*5*	*2*	*3*	*4*
(5)	*4*	*3*	*2*	*2*	*5*	*2*	*5*	*3*	*2*	*2*

Chapter 6

Printing Formats and Multi-Statement Lines

6.1 Printing Formats

We have so far encountered the following forms of the PRINT statement:

```
PRINT A
PRINT "THE CHARACTER",X$
PRINT A,X$,B
```

The execution of any PRINT statements which use *commas* to separate items, causes the standard 40 character wide lines on the video screen to be broken up into 3 areas called *tab fields*. The first two are 16 characters wide while the third is only 8 characters wide. To illustrate this point, execute in the immediate mode the following statement:

```
PRINT "THE VALUES ARE ",1.2,",",3.8
```

In this example, it is desirable not to have such large spaces between the numbers in the output. This can be achieved by replacing some or all of the commas in the PRINT statement by *semicolons*. All items separated by semicolons are printed with no spaces between them.

Ex. 6.1

Execute in immediate mode the above PRINT *statement. Examine the effect on the output of using semicolons in place of some or all of the commas.*

You may exert further control over the spacing between the output characters on the screen, by using the TAB function:

TAB (n)

where n is any integer between 0 and 255, or either a variable or arithmetic expression which has the value of an integer number. The columns are labelled for this function from 1 to 255. If TAB(0) is used the cursor will move to column 256. This statement may appear anywhere in a PRINT statement and when encountered during execution, it will cause the cursor to move n positions from the left margin of the text window, e.g.

```
PRINT TAB(5);"A=";3.8;TAB(15);"B=";4.6
```

Ex. 6.2

Execute the above statement in immediate mode and observe the effect of changing the numeric values in both TAB *functions.*

If n is less than the value of the current cursor position, then the cursor is not moved. If n is such as to move the cursor key beyond the rightmost limit of the text window, then the cursor moves down to the next line and spacing continues from that point.

Ex. 6.3

Execute the following PRINT *statements and account for the output format:*

```
PRINT TAB(5);"A=";3.8;TAB(5);"B=";4.6
PRINT TAB(5);"A=";3.8;TAB(45);"B=";4.6
```

There is another function which takes the form

SPC (n)

When used in a PRINT statement, this function introduces n (any integer, variable or arithmetic expression which on evaluation lies in the range 0 to 255) spaces between the item previously printed and where printing of the next item is to begin. Semicolons rather than commas should be used with this function. Note the distinction between the TAB( ) and SPC( ) functions —the former specifies the distance between the left hand margin of the text window and the current printing position whereas the SPC function specifies the current print position relative to the position of the last character printed.

Ex. 6.4

Explain whether you would expect the following two statements to give exactly the same output format:

i. `PRINT TAB(5);"A=";TAB(10);1.2`

ii. `PRINT SPC(4);"A=";SPC(3);1.2`

Check your predictions by executing both statements in immediate mode.

The TAB and SPC functions must be used within a PRINT statement and can only move the cursor to the right. A more general statement which can be used on its own and *not* in a PRINT statement, and which can cause printing to occur on the left or right of the current printing position, is the HTAB statement:

HTAB n

This causes the printing cursor to move *horizontally* to n (any integer number, variable or arithmetic expression which on evaluation lies in the range 0-255) positions from the left of the text window irrespective of the current printing cursor position.

A related statement:

VTAB m

causes the printing cursor to move *vertically* down m (any integer number, variable or arithmetic expression which on evaluation lies in the range 1 to 24) lines from the top of the screen.

Ex. 6.5

Enter and execute the following program and account for the output:

```
10  FOR I = 1 TO 24
20  VTAB I
30  HTAB I
40  PRINT "###"
50  NEXT I
60  END
```

Insert the statement 55 GOTO 10 *and account for the change in output. Remember to key* CTRL C *to stop an endless loop.*

Sometimes you may wish to *completely clear the screen* and place the cursor at the top left hand corner before printing any output. This is achieved using the statement:

HOME

which has no parameters.

Ex. 6.6

Observe the effect of the HOME *statement by entering, listing and executing the following program:*

```
10  HOME
20  FOR I = 1 TO 24
30  PRINT  TAB( I);I
40  NEXT I
50  END
```

You may wish *to change the appearance of the output* arising from the execution, and not the entering or listing of a program. There are three statements which can be used within a program for this purpose:

1. FLASH

   which causes the output to be alternatively shown on the screen in white (green) or black and then reversed to black or white (green). This output format remains in force until one executes the following instruction *within a program*:

2. NORMAL

   which resets the screen output to white (green) or black with no alternation.

3. INVERSE

   which sets the output to appear as black letters on a white (green) background. Again this output format can be reset to normal conditions by executing a NORMAL statement from within a program.

Ex. 6.7

Enter and execute the program

```
10  HOME
30  FLASH
40  PRINT "4 3 2 1 ...BLAST OFF !"
50  NORMAL
60  END
```

Replace the FLASH *statement by* 30 INVERSE.

Note that even though the output format returns to normal after execution of this program, the output from the program still remains on the screen in either the flashing or inverse mode. This can be removed from the screen by executing a HOME command in immediate mode.

Finally it should be noted that the rate of transfer of characters to peripheral devices including the screen display, can be controlled using the statement:

SPEED = n

where n is in the range 0 (slowest) to 255 (fastest). The normal screen display occurs at the fastest speed.

6.2 Multistatement Lines

In Applesoft BASIC, more than one statement can be used in the same line, e.g.

```
120 X = X + 1: PRINT X:Y = Y + 2
```

Each statement is separated by a colon(:). Only the first statement has a statement number and hence this is the only statement which can be directly branched to by GOTO, GOSUB (see Ch. 7) or IF/THEN statements. Until now all statements have been defined as beginning with a statement number, but it should now be noted that they could equally well be incorporated as the second or subsequent statement in a multistatement line. In this situation they do not need a statement number.

The advantages in using more than one statement per line include that it saves memory by not having to store extra statement numbers; it makes for a more compact program, and it leads to faster execution of statements. The disadvantages include the fact that it is harder to read a program with multistatement lines and that if a mistake is made in one statement then the whole line including any correct statements, must be re-entered.

Ex. 6.8

1. Rewrite the mean and standard deviation program that you developed in Ex. 5.5.2 using TAB statements to obtain a neat output.
   Enter and test your program on some suitable test data.

2. The following information relates to the transactions in a personal bank account for a week:

Starting balance	1/9/82	$670.12
Cheque	2/9/82	$70.84
Deposit	2/9/82	$100.00
Cash withdrawal	5/9/82	$50.00

   Write a program to input this data, calculate the new (current) balance and display the results in the format:

Date Starting Balance

Date	Transaction	Credit	Debit	(New Balance)
....	.....	.....	.....	.....
....	.....	.....	.....	.....
....	.....	.....	.....	.....

Current Balance

   Try to make the data input procedure as 'friendly' as possible.

   Enter and execute for the above data.

3. Write a program which will input some specified number (≤ 100) of integer values in the range 1 to 10 and to plot the frequency of their values as a histogram of the form:

```
        Frequency
v
a   X
l   XXXX
u   XX
e   XXXX
```

   Enter and execute for the series of 30 numbers: 1 3 3 5 1 10 6 8 7 7 1 3 3 1 1 10 2 5 5 5 5 4 3 8 9 4 3 2 2 1.

4. Write a program to plot the function $y = x^2$ for $-6 \leq x \leq +6$, so that it fills the screen as much as possible.

   Enter and execute this program.

Chapter 7

Functions and Subroutines

7.1 Library Arithmetic Functions

There are a number of arithmetic tasks which are common to many programs. These include taking a square root, evaluation of the exponential function and of trigonometric functions such as sin x, cos x and tan x. The steps involved in performing these tasks can be quite numerous and involved. For example, the evaluation of e^x involves summation of the series:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots .$$

To avoid having to repeat the statements associated with all these steps each time whenever these tasks are encountered in a program, a series of commonly used functions is stored permanently in the computer memory (hence the term library or built-in functions). Table 1 lists the library functions which are included in Applesoft BASIC.

These functions can be used in a program by simply inserting their name followed by an *argument* enclosed in brackets. This is a numerical value or an arithmetic expression which upon evaluation gives a numerical value, e.g.

```
X=Y1+SQR(Y1)

X=A+SIN(0.5)

B=EXP(C+2^0.5)

PRINT "THE VALUE OF SIN(";A1;") IS ";SIN(A1)
```

Functions may also be called within functions using appropriate combinations of brackets to specify the arguments, e.g.

```
X=Y+LOG(X+SQR(X))
```

The INT function is particularly useful for *rounding off* numbers to either the nearest whole number, e.g.

```
A=INT(B+0.5)
```

or to say N decimal places, e.g.

```
A=INT(B*10^N+0.5)/10^N
```

Note the addition of 0.5 to insure the *nearest* whole number.

Table 1

Applesoft Function Name (argument)	*Purpose and Value Returned to the Program*
SIN(X)	sin X
COS(X)	cos X
TAN(X)	tan X
	(X is specified in radians)
ATN(X)	$\tan^{-1}$ X returns a value in the range $\pm\frac{\pi}{2}$ radians.
INT(X)	largest integer less than or equal to the value of X
SGN(X)	−1 if X<0 0 if X = 0 +1 if X>0
ABS(X)	Absolute value of X i.e. always ≥ 0
SQR(X)	$\sqrt{X}$
EXP(X)	e^X
LOG(X)	$\log_e X$
RND(X)	If X>0, this function returns a different random real number in the range 0 to 1.0. If X = 0, returns the most recent random number again. If X<0, returns a random number in the range 0 to 1.0. If a particular negative X is used first, then subsequent use of a positive X value will generate the same sequence of random numbers (each time).

Ex. 7.1

1. Write statements using library arithmetic functions to evaluate the following algebraic expressions:

$$y = (e^X + e^{-X})/2$$

$$x = a^2 + b^2 + 2ab\cos x$$

$$E = E - \frac{nR}{F}\log_{10}C \quad (\log_{10}x = 2.303\log_e x)$$

2. *Enter the following program into memory:*

```
10  INPUT "N= ?";N
20  FOR I = 1 TO 5
30  PRINT  RND (N)
40  NEXT I
50  GOTO 10
```

Input N = 1, 0, −1 followed by +2 and account for the random numbers printed as output in terms of the explanation given in Table 1 for this function.

3. *Write and execute a program using the* INT *and* RND *library functions to simulate throwing a dice, i.e. generate random integer numbers in the range 1 to 6.*

4. Write a program to round off a given decimal number to N decimal places.

Enter and execute for N = 2 and the following numbers: 12.1234, 0.0012, −3.612, 3.612. Account for any difference in output for these last two values.

Some arithmetic tasks may not be commonly enough encountered to warrant their being incorporated into the computer memory as library functions. However, some of these tasks may still appear several or more times within a given program. If these tasks can be performed in one statement, then rather than repeat this statement each time that task is encountered in a program, one need only define once a special function name together with one or more appropriate arguments.

The BASIC statement for such a user-defined function has the form:

DEF FN name (real variable name) = arithmetic expression

and this is called in a program by writing:

FN name (argument)

For example, a simple program to calculate the volume of a shell of a sphere with inner and outer radii of a and b respectively might be:

```
10  INPUT A,B
20 V = (4 / 3) * 3.14 * B ^ 3 - (4 / 3) * 3.14 * A ^ 3
30  PRINT A,B,V
```

To avoid repeating the $4/3\ \pi r^3$ terms in stmt. 20, one could define a function to perform this task and incorporate it in the program as follows:

```
10  INPUT A,B
20  DEF  FN VOL(Y) = (4 / 3) * 3.14 * Y ^ 3
30 V =  FN VOL(B) -  FN VOL(A)
40  PRINT A,B,V
```

The function is defined in stmt. 20 to have the name VOL and is expressed in terms of a variable Y. This is called a *dummy argument* because when the function is actually called in stmt. 30, the real arguments A or B replace Y in the evaluation of the function.

Only one argument is allowed in a function and the function name obeys the same rules as variable names in that only the first two letters are unique. Thus a function with the name NOSIG would be considered the same as that with the name NOPTS. A function must be defined before it is used in a program. A particular function name can be redefined later in a program to represent some other arithmetic expression.

Ex. 7.2

1. The only inverse trigonometric library function available in Applesoft BASIC is the arctan (ATN). Incorporate the expression:

   $$\arcsin(x) = \arctan(x/\sqrt{1-x^2})$$

   in a program as a user-defined function to calculate the arcsine function in degrees for $x<1$.

   Enter and execute for $x = 0.553$.

2. Write a program incorporating a user-defined function to perform the summation:

   $$(a+\frac{1}{a})^a + (b+\frac{1}{b})^b + (c+\frac{1}{c})^c$$

7.3 Subroutines

These are similar in concept to functions in that they are a collection of one or more statements which may be repeatedly called from any part of a program. The statement which causes the main program to branch to the first statement of a subroutine is:

GOSUB stmt. no. (n)

where stmt. no. (n) is the statement number of the first statement in the subroutine. Any number and type of statement can then be placed after this first statement. If one or more of these statements transfers control outside the subroutine then control must be subsequently returned to the subroutine

before it is ended. The final statement in a subroutine must be:

RETURN

This indicates to the computer that control is to be passed back to the next statement *after* the original calling GOSUB statement, in the main program.

An example of the possible use of a subroutine might be in a program which involves checking at various points whether the variable A is less than 0, and if so, to print out a message to this effect and set A = 0 before continuing the calculation. A possible program might be:

```
      .       .
      .       .
      .       .
100  IF A >  = 0 THEN 130
110  PRINT "A < 0"
120 A = 0
130   .       .
      .       .
      .       .
      .       .

200  IF A >  = 0 THEN 230
210  PRINT "A < 0"
220 A = 0
230   .       .
      .       .
      .       .
```

One could avoid the repetition of these three statements each time a check on A is made by writing them once as a subroutine and calling it up each time a check is to be made on A as follows:

```
      .       .
      .       .
100  GOSUB 500
      .       .
      .       .
200  GOSUB 500
      .       .
      .       .
500  IF A >  = 0 THEN  RETURN
510  PRINT "A < 0"
520 A = 0
530  RETURN
      .       .
      .       .
```

Clearly as the number of times that A is checked increases, then so does the number of statements saved by using a single subroutine increase.

Other subroutines can be called from within a subroutine and up to 25 subroutines can be nested in this way.

Apart from reducing the total number of statements used in a program, the use of subroutines often allows the grouping together of statements which represent a logical step in a program. In this way they can lead to more intelligible programs and for this reason alone, their use whenever convenient is to be encouraged.

Ex. 7.3

1. The logarithm of a number x to any base, say b, can be calculated by the equation:

$$\log_b(x) = \frac{\ln(x)}{\ln(b)} \qquad \text{where } \ln(x) = \log_e(x)$$

Incorporate this expression as a subroutine in a program to calculate $\log_b(x)$ given b and x.

Enter and execute this program. Compute $\log_2(35)$, $\log_4(27)$ and $\log_{25}(15)$.

2. Write a program to calculate the binomial coefficient given by the expression:

$$\frac{n!}{k!(n-k)!}$$

Incorporate a subroutine to calculate a general factorial (see Ex. (4.5.3)) and then call this from the main program to evaluate each of the three factorial terms.

Enter and execute the program for $n = 6$ and $k = 1, 2, 3, 4, 5$ and 6.

3. Incorporate the subroutine developed in the previous exercise (7.3.2) into a further subroutine to calculate the quantity e^z from the series expression:

$$e^z = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \frac{z^4}{4!} + \ldots .$$

Use these nested subroutines in a program to calculate the hyperbolic cos function defined as:

$$\cosh(z) = \frac{e^z + e^{-z}}{2}$$

Enter and execute for $z = 0.5$.

4. Write a subroutine which will provide linear least squares estimates of the coefficients a and b together with their standard deviations for the linear equation:

$$y = bx + a$$

given n pairs of (x,y) values as input data. Use the following expressions for a and b:

$$a = \frac{\Sigma x^2 \Sigma y - \Sigma x \Sigma xy}{(n\Sigma x^2 - (\Sigma x)^2)}$$

$$b = \frac{n\Sigma xy - \Sigma x \Sigma y}{(n\Sigma x^2 - (\Sigma x)^2)}$$

and their standard deviations:

$$\sigma_a^2 = \frac{\Sigma x^2 \Sigma r^2}{(n-2)(n\Sigma x^2 - (\Sigma x)^2)}$$

$$\sigma_b^2 = \frac{n\Sigma r^2}{(n-2)(n\Sigma x^2 - (\Sigma x)^2)}$$

where $\Sigma r^2 = \Sigma y^2 + b^2\Sigma x^2 + na^2 - 2b\Sigma xy - 2a\Sigma y + 2ba\Sigma x$.

Incorporate this subroutine into a main program which prints out the quantities a, b, and σ_a and σ_b.

Enter and execute for the data:

(1.11, 0.93), (2.05, 1.92), (2.99, 2.95), (4.01, 4.01)

An alternative statement to the simple GOSUB stmt. no. is the statement:

ON n GOSUB stmt. no. (1), stmt. no. (2)

This is similar to the ON/GOTO statement discussed in Section 4.4. and causes execution to pass to the subroutine with n-th statement no. in the list following the GOSUB command. This statement allows branching to different subroutines depending on the value of n which can be an integer or an arithmetic expression evaluating as an integer, e.g.

```
210  ON X% GOSUB 500,600,700
```

7.4 Library Character Functions

Applesoft BASIC provides a number of library functions which perform a variety of tasks on character strings including:

1. LEN (string expression): returns the number of characters (0-255) in a string.

2. LEFT$ (string expression, n_1) and RIGHT$ (string expression, n_2) return the first n_1 leftmost, or the last n_2 rightmost characters respectively in the string expression, e.g.

```
PRINT LEFT$("GOOD MORNING",4)      will output GOOD
PRINT RIGHT$("GOOD MORNING",7)     will output MORNING
```

3. MID$ (string expression, n_1) returns the substring starting at the n_1-th character and preceeding through to the last character of the string, e.g.

```
PRINT MID$("GOOD MORNING",6)       will output MORNING
```

A second argument, n_2, can be specified and in this form the function will now return the substring of n_2 characters beginning at the n_1-th character from the start. e.g.

```
PRINT MID$("THE CRICKET MATCH",5,7)
```
will output CRICKET

In all these three functions viz. LEFT$, RIGHT$ and MID$ the n_1 and n_2 terms can be replaced by arithmetic expressions whose evaluation gives an integer number.

4. STR$ (arithmetic expression) will cause the arithmetic expression to be evaluated and its value then converted into a character string, e.g.

```
A$=STR$(3.15/2)
```

will cause the '1.575' to be assigned to the string variable A$.

5. VAL (string expression) performs the opposite function to STR$ and converts the string expression up to the first non-numeric character (i.e. any character other than the digits 0 to 9, space, decimal point or + and − sign) to a real or integer number. The first character must be an allowable digit or otherwise a 0 is returned, e.g.

```
PRINT VAL("999")
```
will output the number 999
```
PRINT VAL("BALL")
```
will output 0

6. CHR$ (arithmetic expression) converts the value of the arithmetic expression (which must be an integer in the range between 0 and 255) into the corresponding ASCII character.

ASCII stands for the *A*merican *S*tandard *C*ode of *I*nformation *I*nterchange which enables communication, for example, between a keyboard where characters are typed and the computer which only accepts numbers. Thus for every keyboard character there is a corresponding code number. A list of 96 ASCII characters and their numeric codes which are accepted by Applesoft BASIC is given in Table 7.2.

The function ASC (string expression) is the opposite to CHR$ and converts an ASCII character into its ASCII numeric code, e.g.:

```
PRINT CHR$(55)
```
will output 7
```
PRINT ASC("A")
```
will output 65

Examples of the use of these two functions in programs include:

1. A program that reads the ASCII code numbers from a data statement and then prints out the corresponding ASCII characters:

```
10  READ Y
20  IF Y = 0 THEN  END
30  PRINT  CHR$ (Y)
40  GOTO 10
50  DATA  65,80,80,76,69,0
```

2. A program that accepts any single character from the keyboard and outputs on the screen the corresponding ASCII code number:

Table 2

ASCII Keyboard Character	*ASCII Decimal Code*				
ctrl @	0	$	36	H	72
ctrl A	1	%	37	I	73
ctrl B	2	&	38	J	74
ctrl C	3	'	39	K	75
ctrl D	4	(	40	L	76
ctrl E	5	)	41	M	77
ctrl F	6	*	42	N	78
ctrl G	7	+	43	O	79
ctrl H or ←	8	,	44	P	80
ctrl I	9	—	45	Q	81
ctrl J	10	.	46	R	82
ctrl K	11	/	47	S	83
ctrl L	12	Ø	48	T	84
ctrl M or RETURN	13	1	49	U	85
ctrl N	14	2	50	V	86
ctrl O	15	3	51	W	87
ctrl P	16	4	52	X	88
ctrl Q	17	5	53	Y	89
ctrl R	18	6	54	Z	90
ctrl S	19	7	55	-	91
ctrl T	20	8	56	-	92
ctrl U or →	21	9	57	shift-M( ])	93
ctrl V	22	:	58	^	94
ctrl W	23	;	59	-	95
ctrl X	24	<	60		
ctrl Y	25	=	61		
ctrl Z	26	>	62		
ESC	27	?	63		
-	28	@	64		
ctrl shift M	29	A	65		
ctrl ^	30	B	66		
-	31	C	67		
space	32	D	68		
!	33	E	69		
"	34	F	70		
#	35	G	71		

N.B.: ASCII decimal codes 28, 31, 91, 92 and 95 have no keyboard equivalent on an Apple II.

```
10  INPUT "ASC CHARACTER ?";Y$
20  PRINT "ASCII CODE IS "; ASC (Y$)
30  GOTO 10
```

Ex. 7.4

1. *Enter and execute the above two programs.*
2. *Write a program that will convert each of the* ASCII *numeric codes from 32-90 into their corresponding* ASCII *characters.*

ASCII code numbers and their conversion into characters are not commonly encountered in simple BASIC programs. However, they can be involved in addressing peripheral devices such as line printers and disk drives. This feature will be briefly discussed at the end of Ch. 9.

Chapter 8

Graphics, Sound and the Games Controllers

Many applications of computers and the examples considered so far in this book, involve the display of results as numeric or alphabetic characters. This is referred to as the *text* mode. Computers however are also being increasingly used for calculations which involve the display of pictures in what is called the *graphics* mode. The Apple II computer has proved to be particularly suitable in this respect. To fully appreciate and use these facilities requires a detailed knowledge of programming of the Apple hardware and often at a much more fundamental (and hence more difficult) level than Applesoft BASIC. These details are quite beyond the intended scope of the present text and for further details the reader is referred, for example, to Ch. 8 and 9 of ref. [2] and Ch. 6 of ref. [5]. Instead, we will introduce only some simple aspects of these graphical procedures and include a brief description of sound and the games paddles.

Included in these procedures, are provision for varying the colour(s) of the display but as many video monitors are still monochrome, then little emphasis will be given to the interesting colour effects that can be obtained using this provision.

8.1 Text Mode

This is the normal mode of output in which the screen is divided up into 24 lines (rows) each of which can have up to 40 characters (columns). Close examination of these characters will show that they are made up of a grid (matrix) of dots of light, 5 dots wide and 7 dots high. Adjacent characters are separated by one dot space. Up to 960 characters can be displayed on the screen in this mode and their positions can be controlled using the PRINT and related format statements discussed in Chapter 6. For example, the heading:

```
!!!!!!!!!!!
! RESULTS !
!!!!!!!!!!!
```

can be displayed in the centre of the screen using the statements:

```
10  HOME
20  VTAB 8
30  PRINT  TAB( 12);"!!!!!!!!!!!"
40  PRINT  TAB( 12);"! RESULTS !"
50  PRINT  TAB( 12);"!!!!!!!!!!!"
60  END
```

Similarly the letter A can be written as:

```
    A
   A A
  A   A
  AAAAAA
 A     A
A       A
```

using the statements:

```
10  HOME
20  VTAB 8
30  PRINT  TAB( 20);"A"
40  FOR I = 1 TO 5
50  IF I = 3 THEN  PRINT  TAB( 17);"AAAAAAA"
60  IF I <  > 3 THEN  PRINT  TAB( 20 - I);"A"; SPC( 2 * I);"A"
70  NEXT I
```

The following exercises will give you practice in displaying various shapes in this mode.

Ex. 8.1

1. Write programs in text mode to draw the following shapes centred in the middle of the screen.

(a) (b) (c) (d) (e)

2. Combine the statements developed for programs (d) and (e) above, into a single program which displays the message HI! on the screen. Add an extra statement which will cause it to flash on and off.

Enter and test these programs on the computer.

The graphical display produced in this mode, although useful for highlighting results and headings, is obviously rather crude and time consuming to devise. In order to obtain better resolution, one can use either of two special Applesoft BASIC graphics modes.

8.2 Low Resolution Graphics Mode

In this mode the screen can be divided up in either of two ways as shown in Fig. 3.

i. *full graphics*: the screen is divided up into 48 rows each of 40 columns. As for arrays, these are labelled from 0-47 and 0-39 respectively.

ii. *split graphics/text*: The graphics (upper) part of the screen consists of a reduced number of 40 rows, each of which is still 40 columns wide (labelled 0-39). The remaining (lower) part of the screen can be used to display 4 lines of text each of which is up to 40 characters wide. Clearly each line of this text corresponds to 2 rows of graphics. This text area is called the *text window*.

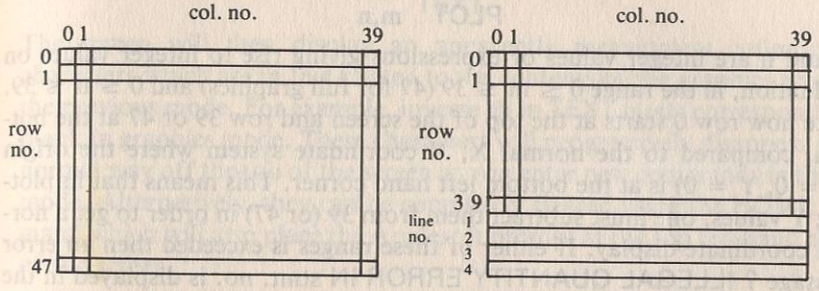


FIGURE 3

This text window displays characters in the usual text manner. However, the graphics part is quite different in that instead of using characters with space around them, solid blocks are used. These blocks are in fact not squares but rectangles approximately twice as wide as they are high. Each of these blocks consists of two coloured sub-blocks stacked on top of each other. Up to sixteen colours can be chosen for each block. On a monochrome video screen, these different colours will appear as different patterns of dots of different intensities.

To enter this low resolution graphics mode from text mode, the following command can be used either in direct or programming mode:

GR

This will switch the Apple from text mode to the split low resolution graphics/text mode. The screen will be cleared and the cursor will be placed at line 1, column 1 of the text window. Any subsequent commands, messages, etc, will appear in this area.

To display a block (point) in the graphics area, one must first *choose a colour* using the command (note the spelling!)

COLOR = n

where n is either an integer in the range 0 to 15 or an arithmetic expression which will evaluate to give such an integer. Each n value corresponds to one of 16 colours:

0 black 1 magenta 2 dark blue 3 purple 4 dark green
5 grey (no. 1) 6 medium blue 7 light blue 8 brown
9 orange 10 grey (no. 2) 11 pink 12 light green
13 yellow 14 aquamarine 15 white

To plot a point in this colour, at the position corresponding to the n-th row (Y axis) and m-th column (X axis), use the command:

PLOT m,n

m and n are integer values or expressions giving rise to integer values on evaluation, in the range $0 \leq m \leq 39$ (47 for full graphics) and $0 \leq n \leq 39$. Note how row 0 starts at the top of the screen and row 39 or 47 at the bottom, compared to the normal X, Y coorindate system where the origin (X = 0, Y = 0) is at the bottom left hand corner. This means that in plotting Y values, one must subtract them from 39 (or 47) in order to get a normal coordinate display. If either of these ranges is exceeded then an error message ? ILLEGAL QUANTITY ERROR IN stmt. no. is displayed in the text window and the program stops.

The colour of the block plotted will be that of the most recent COLOR command. The default value is 0 (black) which means that if a PLOT command is executed before *any* COLOR command then there is no effect on the screen. The following exercise illustrates the use of these low resolution commands in the split graphics/text mode to display the range of colours and screen positions.

Ex. 8.2

Enter and run the following program:

```
10  REM    ** EX.8.2  LOW RES GRAPHICS
20  GR
30  HOME
40  PRINT "COLOUR NO.0 1 2 3 4 5 6 7 8 91011121314"
50  POKE 34,22
60  FOR I = 0 TO 15
70  COLOR= I
80  PLOT 8 + 2 * I,37
90  NEXT I
100  INPUT "COL.NO. ,ROW NO. ,COLOUR?";N,M,I
110  IF M < 0 AND N < 0 GOTO 150
```



```
120  COLOR= I
130  PLOT M,N
140  GOTO 100
150  END
```

Initially choose column and row numbers in the allowed range 0 to 39 and colour numbers in the range 0 to 14. Then choose one or more values outside these ranges. If both the row and column number are set to negative values, then stmt. 110 causes a transfer of control to the end of the program and normal termination. Otherwise an error message will appear and program execution is halted.

In both these situations, the Applesoft prompt will be placed in the text window ready for new commands. If you wish to re-run the program, then enter RUN as normal. If, however, you wish *to return to full text mode* use the command:

TEXT

The screen will then display an apparently meaningless collection of characters which are in fact related to the contents of the graphics screen in the previous mode. For example, inverse @ in TEXT mode corresponds to a blank in graphics mode. These characters will progressively disappear in the normal way off the top of the screen as you enter new commands in the text mode. Alternatively, they can be completely cleared using the HOME command which will also place the Applesoft prompt at the top left hand corner of the screen.

Ex. 8.3

Use the TEXT *and* HOME *commands to re-enter the text mode.*

8.3 PEEK and POKE Statements

The type of instruction encountered in line 50 of the program in Ex. 8.2 is one of two types of instruction which are commonly used in more advanced BASIC programs to directly access or change the contents of memory. These take the form:

i. PEEK (n) function

which reads the value residing in memory location n and assigns it to a variable using a statement of the form:

variable name = PEEK(n)

ii. POKE n,m statement

which causes the value of m to be placed in the memory location n.

In both cases, n can be a number, variable or expression which evaluates to a valid memory location. There are two ways of expressing memory locations, either a positive integer in the range 0 to 65535, or by subtracting 65536 from the positive address number. One reason for having this alter-

native negative form is because the largest positive integer number allowed is 32767 and hence any memory location in the range above this value viz. 32768 to 65535 cannot be expressed as a valid integer number. By subtracting 65536 from an address in this upper range, one obtains a negative integer number which will be valid in the range −32767 to zero. For example, memory location 41234 is expressed as −24301, i.e. 41234-65535. If a memory location is specified as a real value, then the computer will automatically convert it into an integer value.

The parameter m in the POKE statement may be a number, variable or an expression with an integer value between 0 and 255.

In theory, the Apple II computer can have a maximum of 65536 memory locations (labelled 0 to 65535). These are divided up into general areas and each of these is assigned a range of addresses as follows:

Memory Category	*Memory Type*	*Address (Decimal)*
read/write (random access)	RAM	0-49151
input/output (I/O)	RAM	49152-53247
read only	ROM	53248-65535

Specific operations are performed within each of these areas. For example, firmware such as the Applesoft program which interprets the Applesoft statements and the monitor program is located in ROM. Text characters and low resolution graphics symbols are located in RAM at either of the locations 1024-2047 or 2048-3071. These areas are referred to as low resolution graphics *pages* 1 and 2 respectively and are also used for text when in the text mode. Page 2 is not easily accessible but either page can be used for these two modes. The availability of locations in other areas often depends on the size of memory installed in your Apple II computer. For example, in a 32 K byte system, the locations 32768 up to 48152 are unavailable, whereas these can be used in a 48 K byte system.

Some very useful information can be obtained and important processes activated by using PEEK and POKE instructions at appropriate locations. Lists of useful locations are available and the reader is referred to Appendix J of ref. [2], Appendix E of ref. [5] and ref. [6] for further details. Generally one can PEEK at RAM and ROM memory locations but POKE only at RAM locations. However, use of even the PEEK command at some locations can cause certain operations to occur. If the correct location is not specified for either command then unpredictable changes *can* occur to software and disks. Hence, a useful rule to remember is *NEVER USE A* PEEK *OR* POKE *STATEMENT UNLESS YOU ARE QUITE SURE OF THE CONTENTS OF THE LOCATION.*

The *text window* is the area of the screen which is available for the display of characters. In the normal TEXT mode it is 40 columns by 24 lines. Locations 32 to 35 are associated with the *size of the text window*:

Location Address	*Contents*	*Range*
32	Left margin	0-39
33	Width	1-40
34	Top margin	0-23
35	Bottom margin	0-23

PEEK and POKE statements can be used at any of these locations. For the program given above in Ex. 8.2, the number code for the colours is present in line 1 of the text window (labelled line 21 as for full text) during any subsequent plotting of points. This is achieved by resetting the top margin of the text window from the default value of 21 in this split graphics/text mode to 22 using the statement:

```
50 POKE 34,22
```

This results in all subsequent text in this window being restricted to lines 2, 3 and 4 (22, 23 and 24 in full TEXT mode).

Ex. 8.4

Insert the statements:

```
25 PRINT  PEEK (34)
55 PRINT  PEEK (34)
```

into the program given in Ex. 8.2. Run the new program and note how the contents of location 34 have been changed.

A useful memory location is −16384 which is associated with a key being depressed on a *keyboard*. If the value of PEEK (−16384) is greater than 127, then this indicates that a key has been pressed, and the ASCII decimal code for the character associated with that key can be obtained by subtracting 128 from this value. This statement must be followed by POKE −16368, 0 which resets the keyboard strobe and allows the next keyboard character to be read. A common usage of these statements is to cause a resumption of execution of a program after a temporary pause by simply pressing any key on the keyboard as demonstrated in the following program:

Ex. 8.5

Execute the following program:

```
10  REM   ** EX 8.5  KEYBOARD
20  PRINT "PRESS ANY KEY TO CONTINUE..."
30  GOSUB 100
40  PRINT "PRESS $ TO STOP"
50  GOSUB 100
60  IF ( CHR$ (A - 128) = "$") THEN  END
70  PRINT "THAT WAS THE "; CHR$ (A - 128);" KEY ..TRY AGAIN !"
80  GOTO 50
90  REM ** SUBROUTINE TO CHECK AND RESET KEYBOARD
100 A =  PEEK ( - 16384)
110  IF A <  = 127 GOTO 100
120  POKE  - 16368,0
130  RETURN
```

Practice using other characters to stop program execution by changing the $ character in stmt. 60.

The same effect could also have been achieved using a GET *statement (see section 5.3).*

8.4 Further Low Resolution Graphics

An example of the use of the POKE statement relevant to low resolution graphics, involves the locations which set imaginary switches associated with display characteristics. *To eliminate the text window in the split graphics/text mode*, change to full graphics mode and increase the number of rows in the graphics screen to 48, use the statement:

```
POKE -16302,0
```

This clears the text window only for the duration of program execution and in the absence of any PRINT statements within the program. To restore the split graphics/text mode from the full graphics mode use:

```
POKE -16301,0
```

The text window is then returned to the bottom of the screen.

Ex. 8.6

Execute the following program:

```
10  GR
20  POKE  - 16302,0
```

and note how the area normally occupied by the text window has now been replaced by an apparently meaningless collection of characters often in an inverse-type style. These will include the Applesoft prompt] which appears on completion of program execution. To show that in fact the text window is cleared during execution, include the extra delay statements:

```
30  FOR I = 1 TO 2000: NEXT I
```

into the above program and run it again.

To draw a shape in this low resolution graphics mode using solely the PLOT command is clearly laborious in programming and execution time. For example, to draw a simple 'square' in the centre of the graphics screen one might use the statements:

```
10  GR
20  COLOR= 15
30  FOR I = 1 TO 10
40  FOR J = 1 TO 10
50  PLOT 18 + I,18 + J
60  NEXT J
70  NEXT I
```


To plot any longer lines such as the axes of a graph would be even more laborious. To overcome this problem, there are two special instructions specific to this low resolution graphics mode, which can be used to plot horizontal and vertical lines much faster than by using the equivalent PLOT commands. These take the form:

1. VLIN y_1, y_2 AT x_0

which draws a *vertical* line between row nos. y_1 and y_2 in column no. x_0 and:

2. HLIN x_1, x_2 AT y_0

which draws a *horizontal* line in row no. y_0 between column nos. x_1 and x_2. Using these instructions, a program for drawing a 'square' of side length n, might take the form:

```
20  GR
30  HOME
40  INPUT "COLOUR =?";C
50  IF C > 16 GOTO 150
60  COLOR= C
70  INPUT "LENGTH OF SIDE =?";N
80 N1 = 20 - N / 2
90 N2 = 20 + N / 2
100  HLIN N1,N2 AT N1
110  HLIN N1,N2 AT N2
120  VLIN N1,N2 AT N1
130  VLIN N1,N2 AT N2
140  GOTO 40
150  END
```

Ex. 8.7

1. *Execute the above program for a range of colours and lengths. Modify the program using an appropriate loop to display a pattern of the form:*

(Different shadings represent different colours)

consisting of a series of 'squares' of increasing size which fills the entire screen. Arrange for no two squares to be of the same colour.

2. *Write and run programs which draw solid block characters on the screen for the letters A, P, L and E. Combine these into a program which plots the word* APPLE *on the screen.*

3. *A business reported the following monthly sales figures for last year.*

	Jan	Feb	Mar	Apr	May	Jun	Jul
Sales	28	2	37	26	0	13	12

	Aug	Sep	Oct	Nov	Dec
Sales	33	16	34	8	22

Write and execute a program which displays this data as a histogram on the screen. Include the months as a legend in the text window. Arrange for each month to be represented by a different coloured block.

There is one other low resolution graphics function:

SCRN (n, m)

which when incorporated in a statement of the form

variable name = SCRN (n, m)

will assign to that variable name the integer value in the range 0 to 15 corresponding to *the colour of the point at row n and column m*. Since all undrawn points have, by default, a colour value of 0 corresponding to black, this function can be used to decide whether or not a point has been drawn on. This has application for graphics involving moving objects such as in games. A simple example of such a use of this function is the following program which displays a block starting at some random position on the left hand side of a 'square'. It then moves in straight lines within the square and is reflected off the walls.

```
20  GR
30  COLOR= 15
40  HLIN 0,39 AT 0
50  HLIN 0,39 AT 39
60  VLIN 0,39 AT 0
70  VLIN 0,39 AT 39
80 X = 1
90 Y =  INT ( RND (1) * 39)
100 HRIZ = 1
110 VERT = 1
120 X = X + HRIZ
130 Y = Y + VERT
140  IF  SCRN( X,Y) > 0 GOTO 220
150  COLOR= 15
160  PLOT X,Y
170  FOR I = 1 TO 10
180  NEXT I
190  COLOR= 0
200  PLOT X,Y
210  GOTO 120
220  IF (X = 0 OR X = 39) THEN HRIZ =  - HRIZ
230  IF (Y = 0 OR Y = 39) THEN VERT =  - VERT
240  GOTO 120
```

Some explanation of this program is in order particularly as it illustrates some features which are common to many dynamic graphical situations. Stmts. 40-70 draw the outline of the 'square'. The random starting position on the Y axis is generated using the RND function in stmt. 90. Note that this function itself gives only a random value between 0 and 1.0 (see Table 1 p. 56) and hence must be multiplied by 39 to get an appropriate range for the length of the 'square' side. The horizontal and vertical velocities are set in stmts. 100 and 110, and the new X and Y positions calculated in stmts. 120 and 130. If the new position does not coincide with a wall as indicated by SCRN (X, Y) being zero, then the new point is plotted. If however this function is non-zero then a collision has occurred and the horizontal and vertical velocities are adjusted in stmts. 220 and 230 respectively, depending on which type of wall is encountered. Note that in plotting the movement of an object it is always necessary to erase the *previous* position by plotting with the black colour as in stmts. 150 and 160. Stmts. 170 and 180 are an example of a *time delay* procedure. The magnitude of the delay is a function of n: a value of 10 gives a delay of about 14 milliseconds. In this instance, it is used to slow down the motion of the block.

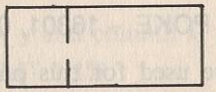
Ex. 8.8

Run the above program. To stop the program, key CTRL C. *Add statements to*

i. stop the program on pressing any key on the keyboard (see Ex. 8.5);

ii. display in the text window a running count of the number of collisions;

iii. include a vertical barrier of the form



and sound the bell on each collision.

8.5 High Resolution Graphics (HGR) Mode

Although useful in a number of applications, the low resolution graphics mode often suffers through the limited resolution obtainable. Thus it is difficult to realistically plot many detailed shapes and mathematical functions involving smooth curves. To overcome this situation there is a further graphics mode in Apple BASIC called high resolution graphics. As for low resolution graphics, this mode operates in either a split graphics/text or full graphics form. The number of columns however is significantly increased to 280 (labelled 0 to 279) while the number of rows is increased to either 160 (labelled 0 to 159) or 192 (labelled 0 to 191) respectively as shown below:

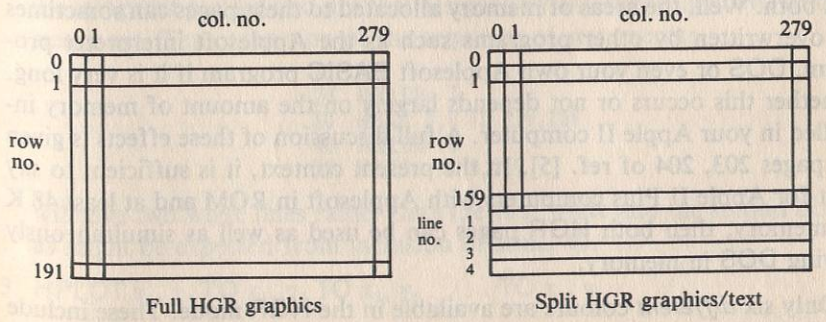


FIGURE 4

The resolution is thus improved by a factor of 7 on the horizontal (X) axis and 4 on the vertical (Y) axis compared to the low resolution graphics mode. One text line is now equivalent to 8 rows and one text column is equivalent to 7 columns. There are a number of different commands used to plot in this mode.

To *enter the* HGR *mode*, use the following command:

```
HGR
```

This will switch the screen output from full text mode to the split HGR graphics/text mode. The screen will be cleared but the cursor will not necessarily be placed in row 1 of the text window and it may be necessary to print several lines before it is visible. The *text window can be removed* and replaced by the remaining 32 rows of graphics using the statement:

```
POKE -16302, 0
```

and *restored* using the statement:

```
POKE -16301, 0
```

These are identical to those used for this purpose in the low resolution graphics mode (p. 72) and behave in a similar manner. To select either graphics mode and either page and without clearing the screen, use the statement:

```
POKE—16304,0.
```

The HGR command accesses an area of memory at locations 8192 to 16383 assigned to high resolution graphics and called HGR page 1. There is also a second area called HGR page 2 in memory locations 16384 to 24575 which can be accessed using the command:

```
HGR2
```

Execution of this command erases the text window and then displays a full HGR screen with *no text window*.

You will no doubt be wondering which of these two pages should be used, if not both. Well, the areas of memory allocated to these pages can sometimes be overwritten by other programs such as the Applesoft interpreter program, DOS or even your own Applesoft BASIC program if it is very long. Whether this occurs or not depends largely on the amount of memory installed in your Apple II computer. A full discussion of these effects is given on pages 203, 204 of ref. [5]. In the present context, it is sufficient to say that for Apple II Plus computers with Applesoft in ROM and at least 48 K of memory, then both HGR pages can be used as well as simultaneously having DOS in memory.

Only six *different* colours are available in the HGR mode. These include black and white which are of two types:

Colour	HGR *Number*
Black 1, 2	0,4
White 1, 2	3,7
Green	1
Violet	2
Orange	5
Blue	6

A colour can be selected using the statement:

```
HCOLOR = n
```

where n must be in the range 0-7.

To plot a point, line, or sequence of lines in HGR *mode* generally using the colour most recently selected, use the HPLOT statement in either of three forms:

1. HPLOT x, y
   plots a single point in row y and column x, e.g.

```
HPLOT 0,159
```

will plot a single HGR point in the bottom left hand corner of the graphics screen.

2. HPLOT x_1, y_1 TO x_2, y_2
   plots a straight line between two points specified by row y_1, column x_1 and row y_2, column x_2, e.g.

```
HPLOT 0,0 TO 279,159
```

will draw a diagonal line from the top left hand corner to the bottom right corner of the screen. An abbreviated form of this statement which can be used to draw a line between the last point plotted and a new point at x_1, y_1 is:

```
HPLOT  TO x1, y1
```

Note that for this abbreviated form of HPLOT, the colour used is that of the last point plotted and not that assigned by any subsequent HCOLOR statement. Thus execution of the sequence of statements:

```
10  HGR
20  HCOLOR= 3
30  HPLOT 0,0 TO 279,159
40  HCOLOR= 0
50  HPLOT  TO 0,159
```

will give two white lines rather than the first white and the second black as might be expected from inclusion of stmt. 40.

3. HPLOT x_1, y_1 TO x_2, y_2 TO x_3, y_3 . . . . TO x_n, y_n

   plots straight lines between points specified by the sequence of the co-

ordinates and in the colour of the most recent HCOLOR statement. For example, the statement:

```
110 HPLOT 0,0 TO 279,0 TO 279,159 TO 0,259 TO 0,0
```

will plot a square around the edge of the graphics screen.

In all of these forms, x and y can be specified as either integer or floating point numbers, variables or arithmetic expressions which evaluate to these quantities provided they remain within the range for x of 0 to 279 and for y of 0 to 159 (191 for full graphics). Floating point values are truncated. If either parameter is outside the allowed range then the error message ?ILLEGAL QUANTITY ERROR will be produced and program execution halted. Attemps to plot y- coordinates between 160 and 191 in the mixed graphics/text mode will not show up on the screen.

Ex. 8.9

In immediate mode, enter the HGR *mode page 1, select a colour and practice using a variety of* HPLOT *commands. Begin with the examples given above. Note whether it is possible to easily observe a single point in this mode compared to a block in the low resolution graphics mode.*

There is no comparable statement in the HGR mode to the SCRN function. Instead the colour and/or the positions of each point plotted can be stored in memory if it is likely that you will need to ascertain whether a given point has been plotted and if so, in what colour.

The increased resolution in the HGR mode enables much more realistic shapes and smoother curves to be displayed than in the low resolution mode. These can be displayed as a series of points either individually or connected by straight lines. As an illustration, consider the trigonometric sine function. To plot one complete cycle of this function that fills the whole screen in the split text/HGR mode, one needs to calculate 280 values of SIN(X) over the range of 0 to 2 π radians (remember the arguments for trigonometric functions in Applesoft are in radians *not* degrees). These values are then scaled to approximately fit into the allowed 160 y (row) positions. A possible program to perform these tasks as well as printing a legend for the horizontal scale and a title in the text window is as follows:

```
20  DIM Y(300)
30  HGR
40  HOME
50  HCOLOR= 3
60  HPLOT 0,0 TO 0,159
70  HPLOT 0,79 TO 279,79
80  FOR J = 1 TO 4
90  HPLOT J * 70 - 1,156 TO J * 70 - 1,159
100  NEXT J
110 XINC = 2 * 3.14159 / 279
```



```
120  FOR I = 0 TO 279
130 Y(I) = 79 - 79 *  SIN (I * XINC)
140  HPLOT I,Y(I)
150  NEXT I
160  VTAB (21)
170  PRINT "0"; TAB( 8);".5 PI"; TAB( 20);"PI"; TAB( 28);"1.5 PI"; TAB( 36);"2 PI"
180  PRINT "SEPARATE POINTS PLOTTED ..WAIT"
190  FOR I = 1 TO 3000: NEXT I
200  FOR I = 0 TO 278
210  HPLOT I,Y(I) TO I + 1,Y(I + 1)
220  NEXT I
230  POKE 34,24
240  PRINT "TO SEE THEM JOINED !"
250  END
```

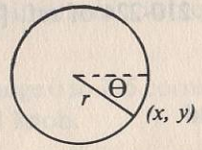
Ex. 8.10

1. *Run the above program for plotting a* SIN *function. Decide for yourself whether separate or joined points are best for representing such a function.*
2. *By changing the value of 2 in stmt. 110 to multiples of 2 viz. 4, 6, ... etc, plot 2, 3, ... cycles for the* SIN *function. Make appropriate changes to the scale in the text window (stmt. 170).*
3. *Prepare similar plots for the* COS *function by changing the* SIN *in stmt. 130 to* COS.
4. *Write a general program for plotting simple mathematical functions of the form $y = f(x)$. Draw in appropriate axes and print a legend in the text window. Execute your program for the following functions. Choose appropriate ranges of x.*

   (i) $y = x^2$
   (ii) $y = 2x^2 - 3$
   (iii) $y = 4x^3$
   (iv) $y = 2x + 0.66$
   (v) $y = 0.5x - \cos x + 2$

5. *The x and y coordinates of a circle can be expressed in terms of the radius r and an angle* Θ *as follows:*

   $$x = r\cos\Theta$$
   $$y = r\sin\Theta$$

   Write and execute a program to plot on page 1 of the HGR *mode, a circle for some radius given as input in the range 1 to 70 and centred in the middle of the screen.*

6. *The value of a commodity showed the following variations in price over a 2 year period:*

	Jan	Feb	Mar	Apr	May	Jun	Jul
year 1	1.93	2.01	2.20	2.40	2.31	2.05	1.86
year 2	2.01	2.05	2.10	2.22	2.40	2.41	2.15

	Aug	*Sep*	*Oct*	*Nov*	*Dec*
year 1	*1.83*	*1.90*	*2.00*	*2.01*	*1.98*
year 2	*2.04*	*1.98*	*1.96*	*1.83*	*1.95*

Execute a program to plot this data on the screen. Distinguish between the two sets of data by plotting one as a continuous line and the other as separate points.

7. *Execute a program which simulates an object moving in a straight path in some random direction from either side of the screen. Use the* RND *function to generate the starting and finishing positions.*
8. *Write and execute a program to plot in* HGR *the character string* HI!

From the above exercises you will have noted that it is not possible to easily plot characters on the screen. Also a large number of HPLOT statements are needed to plot even relatively simple shapes. To overcome these problems and allow for more complex characters and shapes to be plotted, there are facilities in the HGR mode to create *shape tables*. These are effectively instructions to the computer to plot given distances in the four basic directions: up, down, left and right. Once constructed, a shape table can be kept for use in other programs. To use a shape table, it is stored in a separate part of memory to the HGR page and then accessed from the program. There are four special HGR command available to manipulate these shapes:

1. DRAW : draws the shape.
2. XDRAW : erases the shape without removing the background.
3. ROT : rotates the shape by some specified angle.
4. SCALE : scales the size of the shape.

The striking and dynamic visual effects in many computer games are achieved using these commands with shape tables. However, their construction and use is complicated and beyond the scope of this text. The reader is referred to p. 210-224 of ref. [5] and p. 91-100 of ref. [2] for details.

8.6 Sound

The speaker in the Apple II Plus computer produces a single click whenever it is accessed using a statement of the form:

variable name = PEEK (−16336)

Ex. 8.11

1. *Execute the above command in immediate mode for the variable A.*
2. *Enter and run the program:*


```
10  INPUT "N = ?";N
20  FOR I = 1 TO N
30 A =  PEEK ( - 16336)
40  NEXT I
50  GOTO 10
```

and investigate the different sounds that can be achieved with different N values. In particular note the difference between the sounds for N = 1 and N = 2.

To construct more complex sounds, in some cases resembling genuine musical notes, one must vary the frequency of these clicks. This cannot be done using solely Applesoft BASIC statements. Instead machine language routines must be incorporated into the BASIC programs and again this is beyond the scope of the present text. Further information can be obtained, for example, on pages 224-229 of ref. [5].

Finally it should be remembered that the Apple II Plus computer also has a bell which can be rung using the statement:

```
PRINT "": REM  CTRL G
```

8.7 Games Controls (Paddles)

These are peripheral hardware devices which are connected to the Apple computer through a special socket connection on the main board. Each device consists of a knob to control the setting of a variable resistance in the range 0 to 150 k Ω and an on/off button. Up to 4 games controls can be simultaneously operated but normally only two are used at any time. These allow independent input and output to the computer and to this end they are commonly used by two players during a computer game to move playing pieces and to initiate an event such as firing a gun or hitting a ball.

There are two commands that can be used to access a given games control (specified by the integer variable n in the range 0 to 3):

1. PEEK (−16287 − n): this returns a value greater than 127 when the button is pressed. When it is not being pressed it returns a value less than or equal to 127.
2. PDL (n): this returns an integer value in the range 0 to 225 corresponding to the resistance setting of the games control knob.

If two paddles are read consecutively, one must wait about 140 milliseconds between readings.

The following exercises require the use of games cntrol no. 0. Check that this paddle is connected to the computer. If not, carefully follow the instructions given on p.4 of *The Applesoft Tutorial*.

Ex. 8.12

1. *Execute the following program which illustrates the sensing of the games control button:*

```
20  INPUT "TIME DELAY";N
30 A =  PEEK ( - 16287)
40  PRINT A
50  IF A > 127 THEN  PRINT "BUTTON DEPRESSED"
60  FOR I = 1 TO N
70  NEXT I
80  GOTO 30
```

It is necessary to introduce a measureable time delay in order to allow the button to be released before being used again. Execute this program for a range of n values including 10, 100 and 1000. Note the change in the A *value printed out, when the button is pressed.*

2. *Execute the following program which illustrates the use of the games control knob to move a block plotted along the horizontal axes in the low resolution graphics mode.*

```
20  GR
30  HOME
40  VTAB 23
50  PRINT "MOVE KNOB ON PADDLE 0 "
60 NUX = ( PDL (0) / 255) * 39
70  REM  ** DETERMINE IF KNOB HAS BEEN MOVED
80  IF NUX = NOLD GOTO 60
90  COLOR= 0
100  PLOT NOLD,20
110  COLOR= 15
120  PLOT NUX,20
130 NOLD = NUX
140  GOTO 60
```

Note the scaling in stmt. 60 to achieve values in the range of 0 to 39 appropriate to the number of columns in this graphics mode.

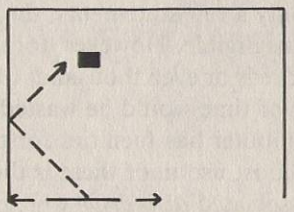
3. *Execute the following program which combines both the games control commands and sound to control the movement of a small triangle plotted in the* HGR *mode.*

```
20  HGR
30 A0 =  PDL (0)
40  PRINT "PRESS BUTTON TO CHANGE DIRECTION"
50 DIR = 1
60 X = 140
70 Y = 75
80  HCOLOR= 3
90  HPLOT X,Y TO X + 2,Y + 2 TO X + 4,Y TO X,Y
95  REM  :: CHECK TO SEE IF BUTTON DEPRESSED
100  IF  PEEK ( - 16287) <  = 127 GOTO 150
110 DIR =  - DIR
120 S =  PEEK ( - 16336)
125  REM ** DELAY TO ALLOW BUTTON TO BE RELEASED
130  FOR I = 1 TO 500
140  NEXT I
145  REM  ** READ KNOB AND SEE IF IT HAS CHANGED
150 A =  PDL (0)
160  IF A = A0 GOTO 100
170 A0 = A
```



```
180  HCOLOR= 0
190  HPLOT X,Y TO X + 2,Y + 2 TO X + 4,Y TO X,Y
200  IF DIR =  - 1 GOTO 230
210 X = A
220  GOTO 80
230  IF A > 150 THEN A = 150
240 Y = A
250  GOTO 80
```

4. *Write and execute a program which simulates a single player hitting a ball in a court. Use the low resolution graphics mode to draw a three sided court of shape:*



Include a block (bat) which can move back and forwards along the lower horizontal axis under the control of the games control knob. Arrange for a ball to be served from the current block position when the games control button is pressed. Assume a diagonal path and allow the ball to bounce off the walls in the normal manner. Click the speaker for each collision. When it returns to the bottom axis, the bat should be moved so as to keep the ball in play. The aim of the game is to keep the ball in play as long as possible. To this end, include a display of the current time, and the best previous time.

Chapter 9

Introduction to the Use of Disks and the Disk Operating System (DOS)

You will have noted already that any program in memory is permanently lost whenever the Apple is switched off or the command NEW is used. For programs consisting of only a few statements, the time required to re-enter them via the keyboard is negligible. However, for many 'real-life' programs which can consist of hundreds or even thousands of statements, an excessive and unnecessary amount of time would be wasted re-entering them via the keyboard. The Apple computer has facilities for the permanent storage of programs and data. The most useful of these is the *floppy disk* (sometimes called diskette) and its associated *disk drive* which is a peripheral hardware device used to read or write information to or from the disk.

9.1 Use of a Disk and Disk Drive

A disk consists of a flexible plastic shape, 5¼ " in diameter, and coated with a brown magnetic material that enables information to be stored on it in a similar fashion to that of an audio recording tape. For protection of this surface and to keep it clean, the disk is permanently sealed within a square black plastic cover which is never opened but allows the disk to spin freely when mounted in the disk drive. In turn, the disk and its plastic cover are commonly stored in a paper envelope. When not in use, the disk should always be kept in this cover and stored vertically.

Although only a small part of the actual disk itself is ever exposed at any one time, any dirt, grease or dust on this exposed part can lead to errors in the transmission of data between the disk and disk drive, or in the data stored on the disk. Hence, this exposed part of the disk should never be touched with bare fingers. Note that the read area is on the *underside* of the disk. The disk should also be kept away from any magnetic or electric fields. A felt tip pen should be used to label the disk in order to prevent any undue pressure on the soft surface.

The disk drive contains a motor which spins the disk within its plastic packet and a read/write head which picks up the encoded information on the disk surface.

To *insert a disk* into an empty disk drive, remove the disk from its paper envelope by handling the label edge between the thumb and forefinger. Open the disk drive door with the other hand by pulling on its bottom edge. Insert the disk horizontally with the label uppermost and the read/write cutout towards the drive:

Gently push the disk into the drive until it encounters some resistance at which point the drive door should be able to be closed completely by gently pushing it down. Keep the disk flat throughout this procedure and do not push too hard as any bending can lead to permanent damage.

To *remove a disk* from a disk drive, *always* check that the disk drive red 'IN USE' light is not on. Then open the disk drive door with one hand and use the other hand to remove the disk, again handling it only in the label region. If the 'IN USE' light is on, wait until it is off before attempting to remove the disk.

9.2 To Load an Applesoft BASIC Program from Disk into Memory

1. Switch on the computer and video monitor with no disk in place as described in Ch. 1.

2. Insert into the disk drive, the disk containing the program that you wish to enter into memory. Key PR#6 RETURN. This is both an Applesoft BASIC and a DOS command which selects the slot in the Apple main computer unit containing the disc drive interface (in this case no. 6) for output and then bootstraps DOS from the disk into memory. The red light indicating IN USE should light up on the disk drive followed by a series of whirring and clacking noises. When these stop and the red light goes out, a message should appear on the screen which relates to the contents of the HELLO program (see Sect. 9.5).

3. Keying CATALOG RETURN should produce on the screen a *menu* of program and data files currently held on that disk. If the number of these files is greater than can be accommodated on the screen, then depression of any key will cause all the remaining files to be displayed. The Applesoft prompt and cursor will only appear when the list is complete. No other commands can be executed until this occurs.

4. Select the name of the program that you wish to load into memory and key the instruction

   LOAD program name RETURN

   The red light will come on again followed by whirring and clacking noises as the program is read into memory from the disk. The screen will clear and be replaced by the Applesoft prompt]. The program is not automatically listed with this LOAD command.

5. When the light goes out and the noises stop, the program should be in memory. It can now be listed and run in the usual way.

If you wish to commence execution immediately, then the LOAD command can be replaced by the RUN command which loads the program and runs it.

If the disk drive continues to whirr and clack endlessly and error messages appear on the screen during any of these steps, there may be a number of explanations for this abnormal behaviour. Sometimes, although the disk drive door has appeared to shut neatly after insertion of the disk, the read head cannot be positioned correctly and/or the disk is not spinning correctly. One possible remedy is to remove the disk from the disk drive, shake it gently in its plastic packet in a vertical plane, reinsert it into the disk drive and repeat the above commands. Other causes for being unable to load a program include that you may have inadvertantly misspelt the program name in the LOAD or RUN command, or that the program is written

in another language such as Integer BASIC which is not available on your particular Apple computer system. There is no immediate remedy for this latter cause. Obviously a damaged or dirty disk will also cause problems. Before rejecting a disk, always try running it on another disk drive if one is available.

Ex. 9.1

Insert the Apple disk labelled DOS 3.3 SYSTEM MASTER *into the disk drive, obtain a menu and execute the program* BRIAN'S THEME.

You will have noticed that each of the files in the menu is preceded by an alphabetic letter. This specifies *the file type* as follows:

A = Applesoft BASIC	B = Binary image (machine language)
I = Integer BASIC	T = Text or data file

Applesoft files can be loaded and run in the usual manner. For details on using the other three file types consult ref. [4].

The number between the file type and file name is the *number of sectors* (length) taken up by the file on the disk.

9.3 To Save a Program in Memory by putting it onto a Disk

1. *Before keying in your program*, insert into the disk drive the initialised disk on which you wish to store your program. Boot DOS using the PR#6 command.

2. Key NEW and then enter your program from the keyboard or from another disk.

3. Key SAVE program name RETURN. This will cause the disk drive IN USE light to come on and whirring and clacking sounds to occur for a short duration. When these cease, the program should be stored on the disk under whatever name you have used in the SAVE statement. If a program under that name already exists on that disk, then the DOS will delete it and replace it with the current program of that name.

4. To check that the program has indeed been stored on disk, key CATALOG RETURN and the program name should appear in a menu along with those of all the other programs and data files on that disk including the initial HELLO program.

Ex. 9.2

Boot DOS *from an initialised disk.*

Type the NEW *command to clear memory of any previous program statements and then enter the short program:*

```
10 PRINT " THIS DOES SHOW"
20 PRINT "THAT PROGRAMS CAN BE STORED ON DISK"
```

Load this program onto an initialised disk using the name TEST *and check that this task has been successfully performed by turning off the power. Turn it on again, boot the disk and then run* TEST.

Obviously a program on one disk can be transferred to another disk by first loading it into memory and then transferring it back to the other disk.

9.4 Renaming, Deleting and Locking a Program on Disk

To *rename* a program currently held on disk:

1. Insert the disk into the disk drive and boot DOS with PR#6.
2. Key RENAME old name, new name, e.g.

```
]RENAME TEST, EXAMPLE
```

One must be careful that there is not another program already on disk with the same new name as DOS will automatically write over this with the latest program.

To *delete* a program currently held on a disk:

1. Insert the disk into the disk drive and boot DOS with PR#6.
2. Key DELETE program name. The program should now be deleted from the disk and the Applesoft prompt] will appear on the screen.

One can check that the program has indeed been changed in name or deleted by typing CATALOG and observing that the program name is changed or no longer in the menu respectively.

To prevent accidental deletion or writing over of a file on disk, use the command:

LOCK name

An asterisk (*) will now appear in the menu by the file with this name. It is suggested that you always lock your HELLO program. To unlock the file for deletion or change, use:

UNLOCK name

Ex. 9.3

Rename the program inserted in Ex. 9.2. Lock it and then attempt to delete it from your disk. Unlock and delete it.

9.5 To Initialise a new (Blank) Disk

Before a disk can be used for storage of programs or data it must be initialised. This involves placing DOS on the new disk and physically organising the layout of disk sectors and tracks.

To initialise the disk:

1. Insert the System Master Diskette into the disk drive and boot DOS using the command PR#6 [RETURN]. The following message (or one very similar) will appear on the screen:

```
DOS VERSION 3.3                    08/25/80
 APPLE II PLUS OR ROMCARD          SYSTEM NAME
(LOADING INTEGER INTO LANGUAGE CARD)
```

   There will then be a wait until the Applesoft prompt] appears on the screen below this message.

   The part of the above message in brackets will only appear if a language card has been added to the basic Apple computer system.

2. Remove the System Master Diskette from the disk drive and insert your new blank disk.
3. Use the greeting program which is loaded in automatically on booting DOS from the System Master Diskette. Alternatively key NEW and then enter a greeting program which might typically take the form:

```
10  HOME
20  PRINT "SLAVE DISK CREATED ON 48 K SYSTEM"
30  PRINT "BY YOUR NAME ON DATE"
40  END
```

   The information which is contained in the PRINT statements in this program will appear on the screen whenever the new disk is subsequently booted. In fact, the information 'DOS VERSION . . . . . etc' which appears when the System Master Diskette is booted, was created in a similar type of greeting program. Any other information that might be useful (such as a disk name) can be included in futher PRINT statements in this greeting program. Other DOS commands can also be included (see the next Section).

4. Execute the greetings program by keying RUN [RETURN] and check from the screen output that all of the required information is in fact printed by the greetings program. Make any necessary changes.
5. Type INIT HELLO [RETURN].
   The new disk will now spin for nearly a minute making many whirring and clacking sounds. These will cease when initialisation is complete and the Applesoft prompt] will appear on the screen.
6. The initialised disk can now be removed from the disk drive and labelled with a felt tip pen before insertion into its paper envelope.
7. To check that the new disk has in fact been correctly initialised, insert the disc and boot by using PR#6 [RETURN]. The information contained in the PRINT statements of the greetings program (now called HELLO) should appear on the screen. If you key CATALOG [RETURN] then only the HELLO program should be listed in the menu.

Ex. 9.4

Obtain a blank disk and initialise it with an appropriate greetings program.

9.6 Use of DOS Commands Within a Program

DOS commands such as PR#*n*, RUN, LOAD, SAVE, etc., can be used in either the direct or programming modes. However, if they are used in the latter context *and* in the presence of DOS, then they must be prefixed by the CTRL D command (ASCII decimal code 4). This is incorporated into a print statement along with the specific command. For example, if you want to obtain a catalog from within a program in the presence of DOS, then an acceptable program statement would be:

```
110  PRINT  CHR$ (4);"CATALOG"
```

An alternative is to assign a string character (traditionally D$) to CTRL D by pressing these keys between quotation marks at the time of keying in the instruction, e.g.

```
105 D$ = "": REM  CTRL-D TYPED BETWEEN QUOTES
110  PRINT D$;"RUN TEST "
```

Note that because the CTRL D character has no printed equivalent, it does not show up in the normal program listing and hence an appropriate REM statement, must always be included to this effect.

DOS commands can be very useful within programs for example, to turn peripheral devices on and off such as printers (see Appendix I), and to initiate execution of other programs. A so-called 'turn key' system in which a program is automatically loaded and execution commenced solely through use of the command PR#6, or on powering-up the computer with the disk in place, can be arranged by incorporating the appropriate RUN statement in the initial HELLO program.

Ex. 9.5

Initialise a disk with the following HELLO *program:*

```
10  HOME
20  PRINT "HELLO PROGRAM EXECUTING"
30  PRINT  CHR$ (4);"RUN TEST"
40  END
```

Key NEW *and load the following program with the name* TEST *on to the disk:*

```
10  PRINT "TEST PROGRAM EXECUTING"
20  PRINT  CHR$ (4);"CATALOG"
30  END
```

With the disk still in the disk drive, switch the computer off and then on again. Note how the program TEST *is now executed automatically without any further commands.*

There are many other DOS commands available for such tasks as creating and handling data files. These and other DOS commands are discussed in detail in ref. [4].

Appendix I

Use of a Printer

There are many occasions when you will want to have a permanent copy of a program listing or of results appearing on the screen. This can be achieved using a printer which is physically connected to the Apple main computer unit via an interface card in slot n. This may be any number between 0 and 7, but for printers is usually 1.

To *access the printer*, the output which is normally to the screen is redirected to the printer using the command:

PR#n

To *return from the printer to the screen*, use the command

PR#0

Both these commands can be used in either direct or programming modes. In the latter case if DOS is present, then they must be prefaced by CTRL D as discussed in the previous Section 9.6.

An example of the use of these two statements within a program would be the situation where one wished to print out on a printer, a character string which is input from the keyboard:

```
10  INPUT Z$
20  PRINT  CHR$ (4);"PR#1"
30  PRINT "THE CHARACTER STRING IS ";Z$
40  PRINT  CHR$ (4);"PR#0"
50  END
```

Ex. I.1

Check that the printer is turned on and ready to receive output from the Apple.

Enter the above program into the Apple. Obtain a listing on the printer by using the commands:

```
]PR#1
]LIST
```

Return the output from the printer to the screen by keying the command:

```
]PR#0
```

and then execute the program in the normal manner using RUN.

Repeat the above listing procedure but omit the PR#0 *command after listing. Execute the program again and note the difference in printer output.*

There are many options available on printers to control the format, including such matters as character size, line spacing, number of characters

per line and underlining. These are controlled by the setting of hardware switches on the printer and/or specific statements within a program. The latter often involve the use of a specific key prefaced by ESC (ASCII decimal code of 27). Thus for example, the printer control statement ESC Ø would be incorporated in a program using the statement:

```
PRINT CHR$(27);CHR$(48)
```

Note that if the printer is set up to print more than 40 characters to a line, then no display will appear on the screen until another PR#0 command is used.

For details of these software and hardware printer controls, refer to the technical manual supplied with your printer.

Appendix II

An Example of Techniques for Debugging Programs

There are many reasons why a program keyed into the computer may not perform satisfactorily. These fall into the three categories:

1. Syntax errors.
2. Run time errors.
3. Logic errors.

The process of removing these and other errors is commonly referred to as *debugging* a program and Applesoft BASIC has a number of facilities available for this purpose.

To illustrate these three types of errors and how one might go about removing them, key the following program into the computer:

```
10  DIM A(2)
20  READ A(1),A(2),A(3)
30 SUM = 0
40  FOR I = 1 TO 2
50 SUM = SUM + A(I)
60  NEXT I
70 PRNNTSUM
80  DATA  1.2,1.3,1.4
90  END
```

This program sums the three numbers specified in the DATA stmt. 80. Upon running this program exactly as given above the following screen output will be observed:


```
]RUN
?BAD SUBSCRIPT ERROR IN 20
```

An examination of stmt. 20 using LIST 20 will not show any obvious error in this statement. However, a consideration of the dimension statement (no. 10) will indicate that only three elements have been assigned to array A viz. A(0), A(1) and A(2) whereas in stmt. 20 an A(3) element is referenced. This same conclusion could have been reached from an explanation of the above error message in Appendix C of ref. [3]. This is an example of a *run time error*.

It can be corrected by either retyping a whole new line:

```
10  DIM A(3)
```

or by using the cursor keys to *change* the 2 to 3 in stmt. 10 as follows:

```
]LIST 10

10  DIM A(2)
]■
```

Key ESC, followed by I key (three times) and then J key (once). This will move the cursor to a position over the 1 in the stmt. no. at the beginning of this statement. The right arrow (retype) key is then pressed 10 times which will move the cursor over the 2 in the dimension of A without removing any characters. Key the new dimension 3 and then the right arrow key once followed by RETURN. If you list stmt. 10 you will see that it has now been modified.

The above procedure illustrates the use of the ESC key with the I, J, K, L keys which move the cursor up, left, right and down respectively anywhere on the screen. Note that to modify a statement it is necessary to move the cursor to the *beginning* of the statement number, before using the right arrow (retype) key. Note also that the cursor must be finally moved right to the end of the line before pressing RETURN as only the characters to the left of the cursor are entered into the computer.

Having removed this error, running the program will now produce the output:

```
]RUN

?SYNTAX ERROR IN 70
```

This is an example of the first type of error and an examination of stmt. 70 will clearly show that PRINT has been misspelt in the original program. Correcting this error and rerunning the program will now give the screen output:

```
]RUN
2.5
]■
```

The program appears to have executed correctly as no error messages have

been printed out. However, an examination of the data shows that the answer should have been 3.9. Thus although the program has no syntax or run time errors, it must have a *logic error*. These are often the most difficult to discover and to correct.

In this example, several methods could be used to discover the logic error(s). Firstly as the progam is short, it is relatively simple to work through each step by hand. This would indicate that in fact only two terms are being summed as expressed by the upper limit in stmt. 40 being 2 and not the 3 as intended. Alternatively this logic error could be discovered by inserting the diagnostic print statement

```
43 PRINT I,SUM
```

within the counting loop. This will print out the value within the loop each time it is executed. Execution of the program would then give the output:

```
]RUN
1             0
2             1.2
2.5
]■
```

from which it is clear that I has only taken on two values rather than three as intended. Stmt. 40 could then be corrected to give an upper limit for I of 3 instead of 2. Stmt. 43 is then deleted.

A third method for determining this logic error involves the use of the TRACE command. If this is used immediately before running as follows:

```
]TRACE

]RUN
```

then the following output will appear:

```
#10 #20 #30 #40 #43 10
#50 #60 #43 2  1.2
#50 #60 #70 2.5
#80 #90
```

This command causes the computer to print out the statement program number of each statement as it is executed as well as the normal output. Again it will be seen that the loop stmts. 50 and 60 are only executed twice instead of the three times as expected. This TRACE command will stay in operation until the NOTRACE command is specifically keyed in. It is particularly useful for tracing the flow of logic in big programs.

Now that all errors have been corrected this program should produce the correct answer of 3.9.

Finally, it should be noted that the cursor control keys (ESC , J, K, L, M) can also be used to *insert and delete* characters in an existing line. For example, if you wished to insert TAB(5) before SUM in the PRINT stmt. 70, then you could precede as follows:


```
]LIST 70
70  PRINT SUM
]■
```

Keying ESC , I (twice), J (once) will bring the cursor over 7 in 70. Retype (10 times—use REPT key) will bring the cursor to the space between PRINT and SUM:

```
70  PRINT■SUM
```

Keying ESC , I (once) will move the cursor one space vertically to the empty line above:

```
    ■
70  PRINT SUM
```

TAB(5); can now be keyed in at this position, using first the space bar and then the appropriate keys for TAB(5); to give

```
        TAB( 5);■
70  PRINT SUM
```

ESC , M will now bring the cursor down to the original line:

```
        TAB( 5);
70  PRINT SUM  ■
```

Keying J (6 times—use the REPT key) will give:

```
        TAB( 5);
70  PRINT ■UM
```

The space bar is then pressed once, followed by the retype (→) key (3 times) which will bring the cursor to the end of SUM and hence the end of the whole statement:

```
        TAB( 5);
70  PRINT SUM■
```

Keying RETURN will enter the whole statement including the inserted characters, into memory as shown by listing this statement:

```
]LIST 70
70  PRINT  TAB( 5);SUM
```

The underlying principle to remember in using these cursor keys for editing purposes is that they only change the position of the cursor and not what is in memory. Subsequent use of any *other* keys will however change the contents of memory when the RETURN key is finally pressed. Thus it is essential to use the retype key to go over all characters from the first digit in the statement number at the beginning of the statement right through to the last character which you wish to include in that line.

Characters can obviously be deleted using a similar procedure by positioning the cursor over the first digit in the statement and then using the retype key up to the first character to be deleted. The space bar is then used to put blank characters in place of all those that you wish to delete, followed by the retype key for the remainder of the statement.

Appendix III

References for Further Information

The following manuals published by the Apple Corporation are very helpful and often contain technical details of hardware as well as software related to Applesoft BASIC.

1. Apple II BASIC Programming Manual.
2. Applesoft II BASIC Programming Reference Manual.
3. Apple II Reference Manual.
4. The DOS Manual (DOS 3.3).

Other useful texts include:

5. 'Apple II Users Guide' by L. Poole with M. McNiff and S. Cook, Osborne/McGraw Hill, 1981.
6. 'What's Where in the Apple—An Atlas to the Apple computer' by W. T. Luebbert, Micro Ink. Inc., 1981.
7. 'Call A.P.P.L.E. in Depth—All About Applesoft', Apple Pugetsound Program Library Exchange, 1981.
8. 'BASIC with Style: Programming Proverbs', P. Nagin and H. F. Ledgard, Hayden Book Company, 1978.
9. 'Beneath Apple DOS', D. Worth and P. Lechner, Quality Software, 1981.
10. 'Apple Machine Language', D. Inman and K. Inman, Reston/Prentice Hall, 1981.
11. 'Apple BASIC—Data File Programming', L. Finkel and J. R. Brown, John Wiley, 1981.

There are a number of periodicals which are either completely devoted to, or contain at least some articles on the Apple computer. Their titles and addresses for subscription details include:

12. Apple Orchard, International Apple Core, P.O. Box 976, Daly City, CA 94017, U.S.A.
13. CALL A.P.P.L.E., 6708 39th Ave., S.W., Seattle, WA 98136, U.S.A.
14. Cider Press, Apple Core SF, 1515 Sloat Blvd., Suite 2, San Francisco, CA 94132, U.S.A.
15. Nibble, P.O. Box 325, Lincoln, MA 01773, U.S.A.

16. Peelings II, P.O. Box 188, Las Cruces, NM 88044, U.S.A.
17. Byte, 70 Main Street, Peterborough, NH 03458, U.S.A.
18. Personal Computing, 50 Essex Street, Rochelle Park, NJ 07662, U.S.A.
19. Popular Computing, P.O. Box 307, Martinsville, NJ 08836, U.S.A.
20. Kilobaud Microcomputing, 80 Pine Street, Peterborough, NH 03458, U.S.A.
21. Softside, Softside Publications, P.O. Box 68, Milford, NH 03055, U.S.A.
22. Micro—The 6502/6809 Journal, MICRO INK, Chelmsford, MA 01824 U.S.A.
23. Windfall, Database Publications Ltd, Europa House, 68 Chester Road, Hazel Grove, Stockport SK7 5NY, England.
24. Creative Computing, P.O. Box 5214, Boulder, CO 80321, U.S.A.

A complete list of these and other Apple books, etc., has been published in Nibble (1982) Vol. 3, No. 1, p. 165.

Appendix IV

Model Answers for Selected Exercises

Ch. 1

1.2

When a line is full the cursor automatically moves down to the next line. 40 characters (including Applesoft prompt]) on each line.

Keying RETURN gives ?SYNTAX ERROR because the symbols that you have been keying do not represent a valid Applesoft BASIC statement or expression.

ESC @ clears the screen.

Ch. 2

2.1

(a) Screen output is as follows:

i. 6.28 ii. 7.28 iii. 3 + 4.28 iv. EX 1.2

(b)
```
PRINT 2
PRINT "APPLESOFT"
PRINT "HELLO"
PRINT 3.148
PRINT "TEST VALUE = ?"
PRINT 2.86 + 6.27
PRINT "-------"
PRINT "1.23456789"
```

Any number of blanks or no blanks are allowed. Keying RETURN a second time gives just the Applesoft Prompt] again and does not lead to the PRINT stmt. being executed again.

2.2.1

1, 16087, 1.23456789E+11, 45780.0, −3.18, −1E+09, Ø, Ø

2.2.2

```
PRINT 2
PRINT −3
PRINT 25853
PRINT 85696
PRINT Ø
PRINT 0.018
PRINT −3.14E-12
```

2.3

	Valid Type	*Invalid*
1A23		Starts with number
%A		Starts with non-alphabetic character
X	Real	
22/3		Starts with number
A12%3		Contains non-alphanumeric character (%)
D.B.		Contains non-alphnumeric character (.)
USA	Real	
ROBERT'S $		Contains non-alphanumeric character (')
NAME %	Integer	
APOLLO—MOONCRAFT$		Contains non-alphanumeric character (−)
ADOG	Real	
ADOG$	String	

2.4.1

Order of execution:

(1) 3 + 2 = 5 (2) 2 ^ 3 = 8 (3) 4 x 8 = 32
(4) 32/16 = 2 (5) 1 + 2 = 3 (6) 3 − 5 = −2 (Answer)

No − correct BASIC expression should be:

(1 + (4*2) ^ 3)/(16 − 3 + 2) = 34.2

2.4.2

	Valid	*Accurate*
(a)	Yes	Yes
(b)	No	No: (A + 2 * B)/C
(c)	Yes	Yes
(d)	Yes	No: 1/(1 + (3 * Y) ^ 0.5)

2.4.3

(a) PRINT (1.01 + 3.06E-12) ^ 0.5 = 1.00498756
(b) PRINT (3 + 10.6)/8.9 = 1.52808989
(c) PRINT 2/(1 + 3 ^ 2) = .2
(d) PRINT (4/3) * 3.14 * R ^ 3
(e) PRINT 1 + A + A ^ 2/2 ^ 2 + A ^ 3/3 ^ 3
(f) PRINT M1 * M2/(M1 + M2)
(g) PRINT 2 * A * B/(C + D ^ 0.38)
(h) PRINT 5/(X + 2) − 2*(X − 8)/(X ^ 2 − 4)

2.5

(1) 1/3 is regarded as an arithmetic expression which is evaluated to give the decimal number of 0.333333333.

(ii) Division by zero.

(iii) 0.1E-38 is less than the minimum allowed real number. Hence it is set to zero.

2.6

Valid: (a), (b), (d), (h), (i).

Invalid: (c) Missing operator between 0.51 and B.
(e) DATE contains reserved word 'AT'.
(f) MONDAY should be enclosed in quotation marks.
(g) Cannot assign a real variable to a string variable.
(j) Cannot have an operator or more than one variable on the left hand side of an assignment statement.

Predict the value of −1, because of rounding off of 2/3-1 = −0.333333333 to an integer value. Execution of X = 2/3-1 will give −0.333333333 because X is now a *real* number.

Ch. 3

3.7

0.333333333

Replace stmt. no. A = 3 by stmt. no. A = 4.

3.8

Message ?DIVISION BY ZERO ERROR IN 15 appears on the screen because you have attempted to calculate 1/0 in stmt. 15.

3.9

2, 125 i.e. two numbers are output because you have overwritten stmt. 10 but not stmt. 20 of your old program.

3.12

```
10  REM   **EX 3.12.1
20  INPUT A
30  INPUT B
40  PRINT 1 / A
50  PRINT 1 / B
60  END

]RUN
?3.6
?6.8
.277777778
.147058823
```

```
10  REM  ** EX 3.12.2
20 X = 1 + 2 + 3 + 4 + 5 + 6
30  PRINT X
40  END

]RUN
21
```

```
10  REM     **EX 3.12.3
20  INPUT A
30  INPUT B
40  PRINT A * B
50  END

]RUN
?10.8
?3.6
38.88
```

```
10  REM    ** EX 3.12.4
20  INPUT DISTNCE
30  INPUT TIME
40  PRINT DISTNCE / TIME
50  END

]RUN
?328
?5
65.6
```

```
10  REM     ** EX 3.12.5
20  INPUT R
30 V = (4 / 3) * 3.14 * R ^ 3
40  PRINT V
50  END

]RUN
?6.8
1316.42198
```

```
10  REM      ** EX 3.12.6
20  INPUT SALARY
30 TAX = (41 / 100) * SALARY
40  PRINT TAX
50  END

]RUN
?25600
10496
```

```
10  REM   **EX 3.12.7.A
20  INPUT METRE
30  PRINT METRE * 100
40  PRINT METRE * 1000
50  END

]RUN
?1.26
126
1260
```

```
10  REM         ** EX 3.12.7.B
20  INPUT POUNDS
30  PRINT POUNDS * 0.45
40  END

]RUN
?3.82
1.719
```



```
10  REM         ** EX 3.12.7.C
20  INPUT F
30  PRINT (5 / 9) * (F - 32)
40  END

]RUN
?98.1
36.7222222
```

```
10  REM         ** EX 3.12.8
20  INPUT PRICE
30  PRINT 10 - PRICE
40  END

]RUN
?6.83
3.17
```

```
10  REM          ** EX 3.12.9
20  INPUT LNGTH
30  PRINT ((LNGTH / 1.25) + 1)
40  REM  ** REMEMBER THE EXTRA END POST !
50  PRINT LNGTH * 5
60  END

]RUN
?460
369
2300
```

```
10  REM  ** EX 3.12.10
20  INPUT WAVELNGTH
30  PRINT (3E8 / WAVELNGTH)
40  END

]RUN
?254E-9
1.18110236E+15
```

```
10  REM   ** EX 3.12.11
20  INPUT NUMBER
30 A% = NUMBER + 0.5
40  PRINT A%
50  END

]RUN
?6.23
6
```

```
10  REM    ** EX 3.12.12
20  INPUT P
30  INPUT V
40  INPUT N
50  PRINT (P * V) / (N * 8.14)
60  END

]RUN
?101E3
?1.2E-3
?0.056
265.882766
```

```
10  REM  **EX 3.12.13
20  INPUT U
30  INPUT A
40  INPUT T
50  PRINT U * T + .5 * A * T ^ 2
60  END

]RUN
?300
?9.8
?250
381250
```

```
10  REM      ** EX 3.12.14
20  INPUT A
30  INPUT N
40  INPUT D
50  PRINT A + (N - 1) * D
60  END

]RUN
?1
?6
?2
11
```

```
10  REM       ** EX 3.12.15
20  INPUT A
30  INPUT N
40  INPUT D
50  PRINT (N / 2) * (2 * A * (N - 1) * D)
60  END

]RUN
?1
?20
?2
760
```

```
10  REM          ** EX 3.12.16
20  INPUT X1
30  INPUT X2
40  INPUT X3
50 MEAN = (X1 + X2 + X3) / 3
60  PRINT MEAN
70  PRINT (((X1 - MEAN) ^ 2 + (X2
      - MEAN) ^ 2 + (X3 - MEAN) ^
     2) / 3) ^ 0.5
80  END

]RUN
?3.65
?3.52
?3.48
3.55
.0725718038
```

Ch. 4

4.3

1. Statements i and ii are valid. Statement iii is invalid because of incorrect construction of the relational expression—the correct form would be:

```
10  IF H < 3 OR H > 3 THEN I = I + 1
```

Statement iv. is similarly invalid and an acceptable form would be:

```
10  IF H > 3 AND X > 3 GOTO 20
```

```
10  REM ** EX.4.3.3
20  INPUT A1
30  INPUT A2
40  INPUT A3
50 MAX = A1
60  IF A2 > MAX THEN MAX = A2
70  IF A3 > MAX THEN MAX = A3
80  PRINT MAX
90  END
```

```
10  REM ** E.4.3.4 EXTRA STMTS.
55 MNN = A1
65  IF A2 < MNN THEN MNN = A2
75  IF A3 < MNN THEN MNN = A3
85  PRINT MNN
```

```
10  REM  ** EX.4.3.5 NUMBER CALC
20 S = 0
30 S2 = 0
40  INPUT A
50  IF A = 0 GOTO 90
60 S = S + A
70 S2 = S2 + A ^ 2
80  GOTO 40
90  PRINT S
100  PRINT S2
110  GOTO 20
```

```
10  REM ** EX.4.3.6
20  INPUT DEPSIT
30 CURRENT = DEPSIT
40  INPUT OUT
50 CURRENT = CURRENT - OUT
60  PRINT CURRENT
70  IF CURRENT < 0.1 * DEPSIT THEN
      PRINT "REMAINDER NOW LESS T
     HAN 10% OF DEPOSIT"
80  IF CURRENT > 0 GOTO 40
90  END
```



```
10  REM  ** EX.4.3.7
20  INPUT QT
30  INPUT ST
40  INPUT IT
50  PRINT IT - ST
60  IF (IT - ST) < QT THEN  PRINT
     "QUALIFIES"
70  IF (IT - ST) >  = QT THEN  PRINT
     "DOES NOT QUALIFY"
80  GOTO 40
```

```
10  REM  ** 4.3.8 SQUARE ROOT
20  INPUT A
30  INPUT B
40  INPUT C
50 Y = B ^ 2 - 4 * A * C
60  IF Y < 0 THEN  STOP
70 X1 = ( - B +  SQR (Y)) / (2 * A)
80 X2 = ( - B -  SQR (Y)) / (2 * A)
90  PRINT X1
100  PRINT X2
110  END
```

4.5

```
10  REM  ** EX.4.5.2 SUM
20  INPUT LOWER
30  INPUT UPPER
40 SUM = 0
50  FOR I = LOWER TO UPPER
60 SUM = SUM + I
70  NEXT I
80  PRINT SUM
```

```
10  REM  ** EX 4.5.4 FACTORIAL
20  INPUT N
30 FACT = 1
40  FOR I = 1 TO N
50 FACT = FACT * I
60  NEXT I
70  PRINT FACT
80  END
```

4.6

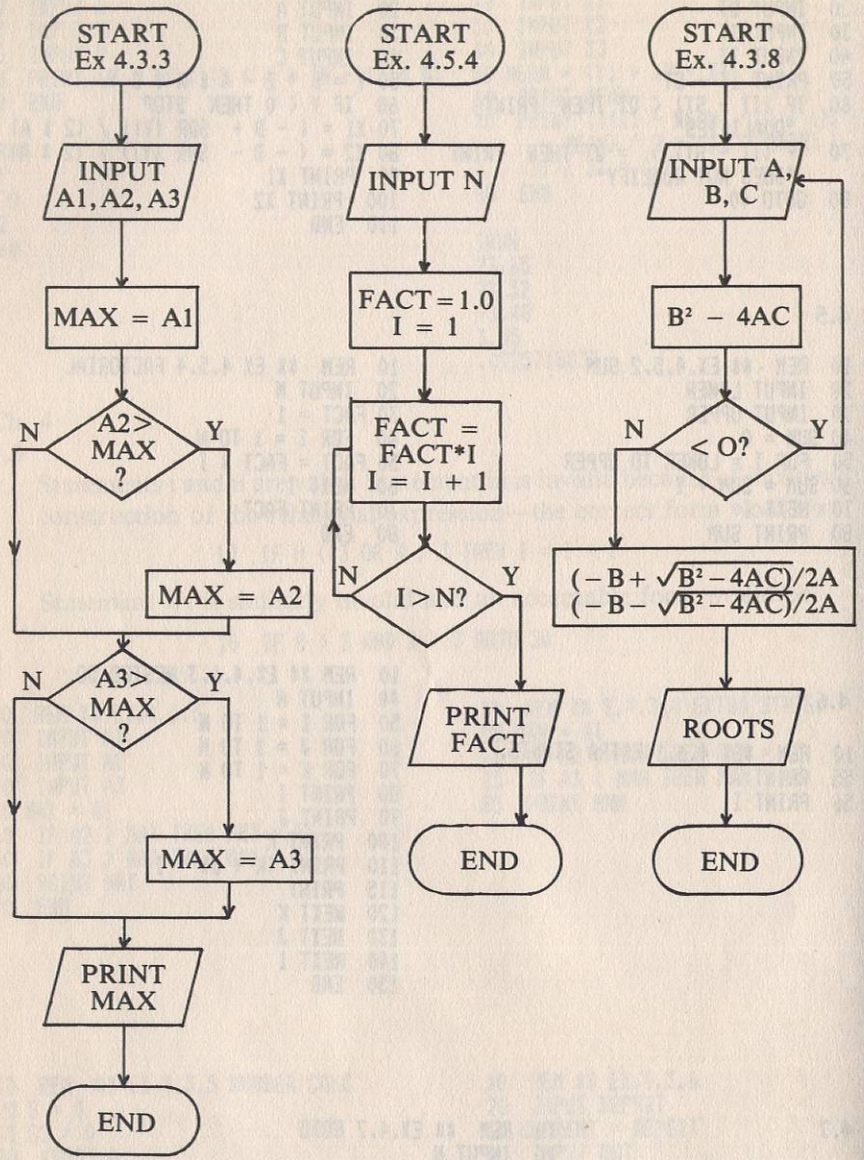
```
10  REM  *EX 4.6.1 EXTRA STMNTS.
55  PRINT J
56  PRINT I
```

```
10  REM ** EX.4.6.3 NESTED DO
40  INPUT N
50  FOR I = 1 TO N
60  FOR J = 1 TO N
70  FOR K = 1 TO N
80  PRINT I
90  PRINT J
100  PRINT K
110  PRINT (K + J) ^ I
115  PRINT
120  NEXT K
130  NEXT J
140  NEXT I
150  END
```

4.7

```
10  REM  ** EX.4.7 GOTO
20  INPUT N
30  ON N GOTO 40,60,80
40  PRINT N + 1
50  GOTO 20
60  PRINT N
70  GOTO 20
80  PRINT N - 1
90  GOTO 20
```

4.8



Ch. 5

5.1

1. Expressions i and iv are valid. Expression ii is invalid because negative subscripts are invalid. Expression iii is invalid because %PROFIT is an invalid variable name.


```
10  REM   ‡‡EX.5.1.2 MEAN AND STD.DEV.
20  DIM A(100)
30 SUM = 0
40  INPUT N
50  FOR I = 1 TO N
60  INPUT A(I)
70 SUM = SUM + A(I)
80  NEXT I
90 MEAN = SUM / N
100  PRINT MEAN
110 SD = 0
120  FOR I = 1 TO N
130 SD = SD + (MEAN - A(I)) ^ 2
140  NEXT I
150  PRINT (SD / N) ^ 0.5
160  FOR I = 1 TO N
170  PRINT A(I)
180  NEXT I
190  END
```

```
10  REM      ‡‡ EX.5.1.4 TWO LISTS
20  DIM L1(100),L2(100)
30  INPUT N
40  FOR I = 1 TO N
50  INPUT L1(I)
60  INPUT L2(I)
70  NEXT I
80  FOR I = 1 TO N
90  FOR L = 1 TO N
100  IF L1(I) = L2(L) THEN  PRINT L1(I)
110  NEXT L
120  NEXT I
130  END
```

```
10  REM  ‡‡ EX 5.1.3  SORT
20  DIM X(100)
30  INPUT N
40  FOR I = 1 TO N
50  INPUT X(I)
60  NEXT I
70  FOR I = 1 TO N - 1
80  IF X(I + 1) > = X(I) GOTO 130
90 A = X(I + 1)
100 X(I + 1) = X(I)
110 X(I) = A
120  GOTO 70
130  NEXT I
140  FOR I = 1 TO N
150  PRINT X(I)
160  NEXT I
170  END
```

```
10  REM ‡‡ EX 5.1.5 TABLE CALC
20  DIM A(3,4)
30 SUM = 0
40  FOR I = 1 TO 3
50 ROW(I) = 0
60  FOR J = 1 TO 4
70  INPUT A(I,J)
80 ROW(I) = ROW(I) + A(I,J)
90  NEXT J
100 SUM = SUM + ROW(I)
110  NEXT I
120  FOR J = 1 TO 4
130 COL(J) = 0
140  FOR I = 1 TO 3
150 COL(J) = COL(J) + A(I,J)
160  NEXT I
170  PRINT COL(J)
180  NEXT J
190  FOR I = 1 TO 3
200  PRINT ROW(I)
210  NEXT I
220  PRINT SUM
230  END
```

5.2

```
10  REM   ‡‡ EX.5.2.2 MAT MULTIPLY
20  DIM A(10,10),B(10,10),C(10,10)
30  INPUT P
40  INPUT M
50  FOR I = 1 TO P
60  FOR J = 1 TO M
70  INPUT A(I,J)
80  NEXT J
90  NEXT I
100  INPUT M
110  INPUT Q
120  FOR I = 1 TO M
130  FOR J = 1 TO Q
140  INPUT B(I,J)
150  NEXT J
160  NEXT I
170  FOR I = 1 TO P
180  FOR J = 1 TO Q
190 C(I,J) = 0
200  FOR K = 1 TO M
210 C(I,J) = C(I,J) + A(I,K) ‡ B(K,J)
220  NEXT K
230  PRINT C(I,J)
240  NEXT J
250  NEXT I
260  END
```

5.3

```
10  REM  ** EX 5.3.2 NAME SEARCH
20  DIM NAME$(20)
30  INPUT "NUMBER OF NAMES IN LIST ";N
40  FOR I = 1 TO N
50  PRINT "NAME ",I
60  INPUT NAME$(I)
70  NEXT I
80  INPUT "SEARCH NAME : ";Z$
90  FOR I = 1 TO N
100  IF Z$ = NAME$(I) THEN  PRINT "NAME
     OCCURS AT POSITION",I
110  NEXT I
120  END
```

5.4

```
10  REM  ** EX.5.4.2  COLOUR COMB.
20  DIM C$(10)
30  INPUT "NO.OF COLOURS :";N
40  FOR I = 1 TO N
50  INPUT "COLOUR:";C$(I)
60  NEXT I
70  PRINT "POSSIBLE COMBINATIONS ARE"
80  FOR I = 1 TO N - 1
90  FOR J = I + 1 TO N
100  PRINT C$(I),C$(J)
110  NEXT J
120  NEXT I
130  END
```

```
10  REM ** EX.5.4.3  LETTER COMB.
20  DIM A$(3)
30  FOR I = 1 TO 3
40  INPUT "LETTER ";A$(I)
50  NEXT I
60  PRINT "POSSIBLE COMBINATIONS ARE"
70 W0$ = ""
80  FOR I = 1 TO 3
90 W1$ = W0$ + A$(I)
100  FOR J = 1 TO 3
110 W2$ = W1$ + A$(J)
120  FOR K = 1 TO 3
130  PRINT W2$ + A$(K)
140  NEXT K
150  NEXT J
160  NEXT I
```

```
10  REM   ** EX.5.4.4 EXTRA STATEMENTS
25  PRINT "EXP.10"
26  PRINT
100  PRINT "MEAN = ";MEAN
150  PRINT "STANDARD DEVIATION =
     ";(SD / N) ^ 0.5
155  PRINT "INPUT DATA"
156  PRINT
```

5.5

```
10  REM   * EX.5.5.3 STUDENT MARK
      S
20  DIM CQ(10),SQZ(10),QSUM(10,5)
     ,Y(10)
30  REM  ** INITIALISE MARK SUMS
     FOR EACH OPTION
40  FOR I = 1 TO 10
50  FOR J = 1 TO 5
60 QSUM(I,J) = 0
70  NEXT J
80  NEXT I
90  REM  ** READ CORRECT ANSWERS
100  FOR I = 1 TO 10
110  READ CQ(I)
120  NEXT I
130  REM  ** INPUT NUMBER OF STUD
     ENTS
140  INPUT N
150  PRINT "STUDENT NO.    MARK"
160  REM  ** INPUT ANSWERS FOR EA
     CH STUDENT
170  FOR I = 1 TO N
180 MARK = 0
190  INPUT SQZ(1),SQZ(2),SQZ(3),S
     QZ(4),SQZ(5),SQZ(6),SQZ(7),S
     QZ(8),SQZ(9),SQZ(10)
200  REM ** CHECK IF ANSWERS ARE
     CORRECT AND STORE ANSWERS
210  FOR J = 1 TO 10
220  IF SQZ(J) = CQ(J) THEN MARK =
     MARK + 1
230 QSUM(J,SQZ(J)) = QSUM(J,SQZ(J
     )) + 1
240  NEXT J
250  REM  ** OUTPUT MARK
260  PRINT I,MARK
270  NEXT I
280  REM  ** SEARCH FOR MOST POPU
     LAR OPTION
290  PRINT "MOST POPUL;AR OPTIONS
      ARE "
300  PRINT "QUESTION","OPTION"
310  FOR I = 1 TO 10
320 NOQ = 0
330  FOR J = 1 TO 5
340  IF QSUM(I,J) <  = NOQ GOTO 3
     70
350 NOQ = QSUM(I,J)
360 Y(I) = J
370  NEXT J
380  PRINT I,Y(I)
390  NEXT I
400  DATA  4,3,3,2,4,1,5,3,2,4
410  END
```



```
10  REM ** EX.5.5.2 DATA EXTENSIONS
60  READ A(I)
95  READ TITLE$
96  PRINT TITLE$
185  DATA   5.61,5.55,5.48,5.53,5.52,5.56,EXP.10
```

Ch.6

6.4

Yes, both give the same output format of 4♭ A = 3♭ 1.2 since SPC(4) specifies 4 spaces *between* first printing position and commencement of printing the character A. This is the same as execution of TAB(5) which specifies that the A should be printed in column 5.

6.5

```
***
 ***
  ***
   ***
    .
     .
      .
```

Insertion of the GOTO 10 statement causes the whole top right hand triangle part of the screen to be filled with * characters. This occurs because HTAB n shifts the printing cursor n spaces from the *previous* printing position (not from the edge of the text window).

6.8

```
10  REM  ** EX.6.8.1 EXTRA STATEMENTS
25  PRINT  TAB( 20);"EXP.10"
100  PRINT  TAB( 15);"MEAN = ";MEAN
150  PRINT  TAB( 3);"STANDARD DEVIATION";(SD / N) ^ 0.5
155  PRINT  TAB( 12);"INPUT DATA"
156  PRINT
170  PRINT  TAB( 15);A(I)
```

```
10  REM    ** EX.6.8.2 BANK STMT.

20  DIM PURPSE$(100),DTE$(100),AM
    OUNT$(100)
30  INPUT "STARTING BALANCE ";SB
40  INPUT "DATE";SBDTE$
50  INPUT "NUMBER OF TRANSACTIONS
    ?";N
60  FOR I = 1 TO N
70  PRINT "TRANSACTION ";I
80  INPUT "PURPOSE ";PURPSE$(I)
90  INPUT "DATE ";DTE$(I)
100  INPUT "AMOUNT ";AMOUNT(I)
110  NEXT I
120  HOME
130  PRINT SBDTE$; TAB( 12);"STAR
    TING BALANCE"; TAB( 33);"$";
    SB: PRINT
140  PRINT "DATE  TRANSACTION"; TAB(
    19);"CREDIT  DEBIT"
150  PRINT "----  -----------"; TAB(
    19);"------  -----": PRINT
160  FOR I = 1 TO N
170 SB = SB + AMOUNT(I)
180 J = 20
190  IF AMOUN(I) >  = 0 GOTO 220
200 J = 26
210 AMOUNT(I) =  - AMOUNT(I)
220  PRINT DTE$(I); TAB( 9);PURPS
    E$(I); TAB( J);"$";AMOUNT(I)
    ; TAB( 33);"$";SB
230  NEXT I
240  PRINT : PRINT  TAB( 12);"CUR
    RENT BALANCE "; TAB( 33);"$"
    ;SB
250  END
```

```
10  REM   ** EX.6.8.4  PLOT OF Y-
    X^2
20  HOME
30  PRINT  TAB( 3);"0"; SPC( 3);"
    4"; SPC( 4);"9"; SPC( 6);"16
    "; SPC( 7);"25"; SPC( 9);"36
    "
40  FOR I =  - 6 TO 6
50  PRINT I; TAB( I ^ 2 + 3);"*"
60  NEXT I
70  END
```

```
10  REM    ** EX.6.8.3 HISTOGRAM

20  DIM SUM(10),X$(100)
30  REM  ** SET COUNTERS TO ZERO
40  FOR I = 1 TO 10
50 SUM(I) = 0
60  NEXT I
70  REM  ** READ Y-AXIS HEADING
80  FOR I = 3 TO 7
90  READ Y$(I)
100  NEXT I
110  REM    ** INPUT NO.OF NUMBER
    S
120  INPUT "NO. OF POINTS =";N
130  PRINT "ENTER EACH NUMBER"
140  REM  ** INPUT EACH OF THESE
    NUMBERS
150  FOR I = 1 TO N
160  INPUT Y%
170 SUM(Y%) = SUM(Y%) + 1
180  NEXT I
190  REM  **PRINT HISTOGRAM
200  PRINT  TAB( 5);"FREQUENCY": PRINT

210  PRINT  TAB( 5);"012345678910
    "
220  FOR I = 1 TO 10
230  REM  ** FOR EACH NUMBER ,EVA
    LUATE
240  REM  ** LENGTH OF LINE OF *'
    S
250  FOR J = 1 TO N
260 X$(J) = " "
270  NEXT J
280  FOR J = 0 TO SUM(I)
290 X$(J) = "*"
300  NEXT J
310  REM   **CONCATENATE ALL X$(J
    )'S INTO ONE WORD
320 WRD$ = X$(0)
330  FOR J = 1 TO 10
340 WRD$ = WRD$ + X$(J)
350  NEXT J
360  PRINT Y$(I); TAB( 3);I; TAB(
    5);WRD$
370  NEXT I
380  DATA  V,A,L,U,E
390  END
```


Ch. 7

7.1

1. $Y = (EXP(X) + EXP(-X))/2$

   $X = A \wedge 2 + B \wedge 2 + 2*A*B*COS(X)$

   $E = E - (N*R/F) * 2.303 * LOG(C)$

```
10  REM  ** EX.7.1.3 DICE
20 IN =  INT (6 *  RND (1) + 1)
35  PRINT "DICE VALUE = ";IN
40  INPUT "TYPE Y TO THROW AGAIN ?";Y$
50  IF Y$ = "Y" GOTO 20
60  END
```

```
10  REM  ** EX.7.1.4 ROUNDING OFF

20  INPUT "N= ";N
30  INPUT "NUMBER = ";B
40  PRINT "ROUNDED NUMBER = "; INT
    (B * 10 ^ N + 0.5) / 10 ^ N
50  GOTO 30
```

7.2

```
10  REM  ** EX.7.2.1 ARCSINE
20  DEF  FN ARS(X) =  ATN (X /  SQR
    (1 - X ^ 2))
30  INPUT "X= ";X
40  PRINT "ARCSINE (";X;") = "; FN
    ARS(X)
50  GOTO 30
```

```
10  REM  EX. 7.2.2 SUMMATION
20  DEF  FN S(X) = (X + 1 / X) ^
    X
30  INPUT A,B,C
40  PRINT  FN S(A) +  FN S(B) +  FN
    S(C)
50  END
```

7.3

```
10  REM  EX. 7.3.1 LOG TO ANY BAS
    E
20  INPUT "BASE =";B
30  INPUT "X =";X
40  GOSUB 70
50  PRINT "LOG(";X;") TO BASE ";B
    ;" = ";LGBX
60  GOTO 20
70 LGBX =  LOG (X) /  LOG (B)
80  RETURN
```

```
10  REM   ** EX 7.3.2 BINOMIAL CO
    EFFICIENT
20  INPUT "N,K =";N,K
30 X = N
40  GOSUB 150
50 X1 = FACT
60 X = K
70  GOSUB 150
80 X2 = FACT
90 X = N - K
100  GOSUB 150
110 RESULT = X1 / (X2 * FACT)
120  PRINT N,K,RESULT
130  GOTO 20
140  REM   ** START OF SUBROUTINE

150 FACT = 1.0
160  IF X = 0 THEN  RETURN
170  FOR I = 1 TO X
180 FACT = FACT * I
190  NEXT I
200  RETURN
210  END
```

```
10  REM  ** EX.7.3.3 COSH
20  INPUT "Z= ";Z
30  GOSUB 100
35 EPLUS = EZ
40 Z =  - Z
60  GOSUB 100
70 EMINUS = EZ
80  PRINT "COSH(";Z;") =";(EPLUS +
    EMINUS) / 2
90  END
100 EZ = 1
110 X = 1
120  GOSUB 180
130 TERM = Z ^ X / FACT
140  IF  ABS (TERM) < 1E - 06 THEN
     RETURN
150 EZ = EZ + TERM
160 X = X + 1
170  GOTO 120
180 FACT = 1.0
190  IF X = 0 THEN  RETURN
200  FOR I = 1 TO X
210 FACT = FACT * I
215  NEXT I
220  RETURN
```

7.4

```
10  REM   ** EX.7.4.2 CONVERT ASC
    II
20  HOME
30  PRINT "ASC II DEC. CODE ASC I
    I KEYBOARD CHARACTER"
40  FOR I = 32 TO 90
50  PRINT  TAB( 6);I; TAB( 25); CHR$
    (I)
60  NEXT I
70  END
```

```
10 REM ** EX.7.3.4 LEAST SQUARES
20  DIM X(100),Y(100)
30  REM   ** INPUT NO.OF POINTS
40  INPUT "NO. OF POINTS =";N
50  REM  ** INPUT EACH X,Y PAIR
60  FOR I = 1 TO N
70  PRINT "X,Y FOR POINT ";I
80  INPUT X(I),Y(I)
90  NEXT I
100  GOSUB 150
110  REM   ** OUTPUT RESULTS
120  PRINT " GRADIENT= ";B;" +- "
     ;SB: PRINT
130  PRINT "INTERCEPT =";A;"+- ";
     SA
140  END
150  REM  ** START OF SUBROUTINE
160  REM  ** SET ALL SUMS TO ZERO

170 X1SUM = 0:X2SUM = 0:Y1SUM = 0
     :Y2SUM = 0:XYSUM = 0
180  REM  ** CALCULATE ALL SUMS
190  FOR I = 1 TO N
200 X1SUM = X1SUM + X(I):X2SUM =
     X2SUM + X(I) ^ 2
210 Y1SUM = Y1SUM + Y(I):Y2SUM =
     Y2SUM + Y(I) ^ 2
220 XYSUM = XYSUM + X(I) * Y(I)
230  NEXT I
240  REM  ** CALCULATE A,B AND DE
     VIATIONS
250 A = (X2SUM * Y1SUM - X1SUM *
     XYSUM) / (N * X2SUM - X1SUM ^
     2)
260 B = (N * XYSUM - X1SUM * Y1SU
     M) / (N * X2SUM - X1SUM ^ 2)

270 R2 = Y2SUM + B ^ 2 * X2SUM +
     N * A ^ 2 - 2 * B * XYSUM -
     2 * A * Y1SUM + 2 * B * A *
     X1SUM
280 SA =  SQR ((X2SUM * R2) / ((N
     - 2) * (N * X2SUM - X1SUM ^
     2)))
290 SB =  SQR ((N * R2) / ((N - 2
     ) * (N * X2SUM - X1SUM ^ 2))
     )
300  RETURN
```


Ch. 8

8.1

```
10  REM  ** EX 8.1.1.A LETTER O
20  HOME
30  VTAB (10)
40  PRINT  TAB( 16);"@@@@"
50  FOR I = 1 TO 6
60  PRINT  TAB( 16);"@  @"
70  NEXT I
80  PRINT  TAB( 16);"@@@@"
90  END
```

```
10  REM   ** EX.8.1.1.C TRIANGLE
20  HOME
30  VTAB (10)
40  PRINT  TAB( 13);"----------"

50  FOR I = 1 TO 5
60  PRINT  TAB( 12 + I);"!"; SPC(
     11 - 2 * I);"!"
70  NEXT I
80  PRINT  TAB( 18);"!"
90  END
```

```
10  REM   * EX.8.1.1.E  LETTER I
20  HOME
30  PRINT  TAB( 17);"IIIII"
40  FOR I = 1 TO 5
50  PRINT  TAB( 19);"I"
60  NEXT I
70  PRINT  TAB( 17);"IIIII"
80  END
```

8.7

```
10  REM ** EX .8.7.1 SQUARES
20  GR
30  HOME
40  FOR N = 2 TO 38 STEP 2
50  COLOR=  INT ((N / 38) * 15)
60 N1 = 20 - N / 2
70 N2 = 20 + N / 2
80  HLIN N1,N2 AT N1
90  HLIN N1,N2 AT N2
100  VLIN N1,N2 AT N1
110  VLIN N1,N2 AT N2
120  NEXT N
130  END
```

```
10  REM  ** EX 8.1.1.B RHOMBUS
20  HOME
30  VTAB (10)
40  PRINT  TAB( 18);"------"
50  FOR I = 1 TO 5
60  PRINT  TAB( 18 - I);"/    /"
70  NEXT I
80  PRINT  TAB( 12);"------"
90  END
```

```
10  REM   * EX.8.1.1.D LETTER H
20  HOME
30  FOR I = 1 TO 7
40  IF I = 4 THEN  PRINT  TAB( 17
     );"HHHH"
50  IF I <  > 4 THEN  PRINT  TAB(
     17);"H  H"
60  NEXT I
70  END
```

```
10  REM  ** EX 8.1.2 FLASHING HI!

20  HOME
30  FLASH
50  VTAB 8
60  HTAB (12)
70  PRINT "H  H    IIIII   !"
80  FOR I = 1 TO 5
90  HTAB (12)
100  IF I = 3 THEN  PRINT "HHHH";
     "      ";"I";"     ";"!"
110  IF I <  > 3 THEN  PRINT "H
     H";"     ";"I";"     ";"!"
120  NEXT I
130  HTAB (12)
140  PRINT "H  H    IIII    ."
145  NORMAL
150  NORMAL
160  END
```

```
10  REM  ** E.8.7.2  APPLE IN LGR

20  GR
30  COLOR= 15
40 X = 2
50 Y = 15
60  REM ** LETTER  "A"
70  FOR I = 0 TO 4
80  PLOT X + I,Y + 9 - 2 * I
90  PLOT X + I,Y + 8 - 2 * I
100  PLOT X + 4 + I,Y + 2 * I
110  PLOT X + 4 + I,Y + 2 * I + 1

120  NEXT I
130  HLIN X + 2,X + 6 AT Y + 7
140 X = X + 5
150  FOR I = 1 TO 2
160 X = X + 6
170  REM  ** LETTER "P"
180  VLIN Y,Y + 9 AT X
190  HLIN X + 1,X + 2 AT Y
200  PLOT X + 3,Y + 1
210  PLOT X + 4,Y + 2
220  PLOT X + 3,Y + 3
230  HLIN X + 1,X + 2 AT Y + 4
240  NEXT I
250 X = X + 6
260  REM   ** LETTER "L"
270  VLIN Y,Y + 9 AT X
280  HLIN X + 1,X + 4 AT Y + 9
290 X = X + 6
300  REM   ** LETTER "E"
310  VLIN Y,Y + 9 AT X
320  HLIN X,X + 4 AT Y
330  HLIN X + 1,X + 2 AT Y + 4
340  HLIN X + 1,X + 4 AT Y + 9
350  END
```

8.8

(i) Add the additional statements:

```
75  PRINT "PRESS ANY KEY TO STOP"

205 A =  PEEK ( - 16384)
206  IF A > 127 THEN  STOP
```

(ii) Replace stmt. 240 by the statements:

```
240 NOCOL = NOCOL + 1
250  HOME
260  PRINT "NO OF COLLISIONS =";N
     OCOL
270  GOTO 120
```

```
10  REM  ** EX 8.7.3  MONTHLY SAL
     ES LGR
20  GR
30  HOME
40  PRINT "JN FB MR AP MY JN JL A
     G SE OT NV DC"
50  DIM S(12)
60  FOR I = 1 TO 12
70  READ S(I)
80  COLOR= I
90  VLIN 39 - S(I),39 AT 2 * I +
     I - 3
100  NEXT I
110  DATA  28,2,37,26,0,13,12,33,
     6,34,8,22
120  END
```


(iii) Include the new statements:

```
75  VLIN 0,15 AT 15
76  VLIN 25,39 AT 15
220  IF (X = 0 OR X = 39 OR X = 1
     5) THEN HRIZ =  - HRIZ
230  IF (Y = 0 OR Y = 39) THEN VE
     RT =  - VERT
270  PRINT ""; REM  **CTRL G
280  GOTO 140
```

8.10

```
10  REM  **EX 8.10.4 Y=F(X) PLOT
20  DEF  FN Y(X) = X ^ 2
30  HGR
40  HCOLOR= 3
45  REM ** PLOT AXES
50  HPLOT 0,79 TO 279,79
60  HPLOT 139,0 TO 139,159
70  INPUT "MAX. X VALUE = ";MAX
75  REM   ** CALCULATE AXES SCALE
     S
80 YCALE = 80 /  FN Y(MAX)
90 XCALE = 280 / (2 * MAX)
100  FOR I = 0 TO 139
105  REM  ** CALCULATE Y VALUES F
     OR GIVEN X VALUES
110 Y1 = YCALE *  FN Y(I / XCALE)

120 Y2 = YCALE *  FN Y( - I / XCA
     LE)
125  REM  ** CHECK THAT Y VALUESW
     ITHIN ALLOWED PLOT SIZE
130  IF Y1 > 80 OR Y1 <  - 80 GOTO
     150
140  HPLOT 140 + I,80 - Y1
150  IF Y2 > 80 OR Y2 <  - 80 GOTO
     170
160  HPLOT 140 - I,80 - Y2
170  NEXT I
175  REM  ** PLOT AND PRINT LEGEN
     D
180  HPLOT 0,153 TO 0,159
190  HPLOT 70,153 TO 70,159
200  HPLOT 210,153 TO 210,159
210  HPLOT 279,153 TO 279,159
220 A =  INT (100 * MAX / 2) / 10
     0
230  HOME
240  VTAB (21)
250  PRINT  - 2 * A; TAB( 10); -
     A; TAB( 21);0; TAB( 30);A; TAB(
     37);2 * A
```

```
10  REM  ** EX 8.10.5 CIRCLE
20  HGR
30  HCOLOR= 3
40 PI = 3.14159
50  INPUT "RADIUS =";R
60  IF R > 70 THEN R = 70
70 DANGLE = PI / 160
80  FOR ANGLE = 0 TO 2 * PI STEP
     DANGLE
90  HPLOT 140 + R *  COS (ANGLE),
     70 - R *  SIN (ANGLE)
100  NEXT ANGLE
110  GOTO 50
```

```
10  REM  ** EX 8.10.6 COMMODITY G
    RAPH
20  DIM S(24)
30  HGR
40  HOME
50  HCOLOR= 3
60  FOR I = 1 TO 24
70  READ S(I)
75  REM  ** DETERMINE MAX AND MIN
    VALUES
80  IF S(I) > MAX THEN MAX = S(I)

90  IF I = 1 THEN MNN = S(I)
100  IF S(I) < MNN THEN MNN = S(I
    )
110  NEXT I
115  REM  ** CALCULATE Y SCALE
120 YCALE = 120 / (MAX - MNN)
130  FOR I = 1 TO 12
140  IF I = 12 GOTO 160
150  HPLOT 7 + 21 * (I - 1),130 -
    YCALE * (S(I) - MNN) TO 7 +
    21 * I,130 - YCALE * (S(I +
    1) - MNN)
160  HPLOT 7 + 21 * (I - 1),130 -
    YCALE * (S(12 + I) - MNN)
170  NEXT I
180  VTAB (23)
190  PRINT "JN FB MR AP MY JN JL
    AG SE OT NV DC"
200  PRINT  TAB( 12);"ANNUAL SALE
    S"
210  DATA  1.93,2.00,2.01,1.98,2.
    01,2.05,1.86,1.83,1.90,2.00,
    2.01,1.98,2.01,2.05,2.10,2.2
    2,2.40,2.41,2.15,2.04,1.98,1
    .96,1.83,1.95
```

```
10  REM  ** E.8.10.7
20  HGR
30 XINC = 1
40  REM  ** RANDOMDETERMINATION O
    F WHICH SIDE TO START
50 XSTART =  INT ( RND (1) + 0.5)
    * 279
60  IF (XSTART = 279) THEN XINC =
    - 1
70  REM  ** CALCULATE RANDOM STAR
    TING AND FINISHING POSITIONS

80 YSTART =  RND (1) * 159
90 YFINISH =  RND (1) * 158
100  REM  ** CALCULATESLOPE OF FL
    IGHT PATH
110 YINC = (YFINISH - YSTART) / 2
    80
120  FOR I = 0 TO 279
130  HCOLOR= 3
140  HPLOT XSTART + I * XINC,YSTA
    RT + I * YINC
150  HCOLOR= 0
160  HPLOT XSTART + I * XINC,YSTA
    RT + I * YINC
170  NEXT I
180  GOTO 30
```

```
10  REM  **EX 8.10.8 HI
20  HGR
30  HCOLOR= 3
40  HPLOT 40,70 TO 40,150
50  HPLOT 40,110 TO 80,110
60  HPLOT 80,70 TO 80,150
70  HPLOT 100,70 TO 140,70
80  HPLOT 100,150 TO 140,150
90  HPLOT 120,70 TO 120,150
100  HPLOT 180,40 TO 180,120
110  HPLOT 180,140
120  END
```

```
10  REM  ** EX 8.12.4 BALL GAME
20  GR
30  HOME
40  COLOR= 15
50 TIME = 0
60 OLDX = 15
70  REM  ** DRAW COURT SIDES
80  HLIN 0,30 AT 0
90  VLIN 0,39 AT 0
100  VLIN 0,39 AT 30
110  REM  ** PLOT BAT AND SET INITIAL BALL CONDITIONS
120  PLOT OLDX,39
130 HRIZ = 1
```



```
140 VERT =  - 1
150 Y = 39
160  REM  ** READ PADDLE TO PLOT NEW BAT POSITION
170 NUX =  INT (( PDL (0) / 255) * 29) + 1
180  IF NUX = OLDX GOTO 250
190  COLOR= 0
200  HLIN OLDX - 1,OLDX + 1 AT 39
210  COLOR= 15
220  HLIN NUX + 1,NUX - 1 AT 39
230 OLDX = NUX
240  REM  ** CHECK BUTTON TO INITIATE SERVE
250  IF (TIME = 0 AND  PEEK ( - 16287) <  = 127) GOTO 170
260  IF TIME = 0 THEN X = NUX
270 X = X + HRIZ
280 Y = Y + VERT
290 TIME = TIME + 1
300  VTAB (23)
310  PRINT "CURRENT TIME = ";TIME;"          "
320  REM ** CHECK FOR A COLLISION
330  IF  SCRN( X,Y) > 0 GOTO 440
340  REM  ** CHECK FOR A BAT MISS
350  IF (Y = 39) GOTO 490
360  COLOR= 15
370  PLOT X,Y
380  FOR I = 1 TO 15
390  NEXT I
400  COLOR= 0
410  PLOT X,Y
420  GOTO 170
430  REM  ** SOUND(TWICE) FOR A COLLISION
440 Z =  PEEK ( - 16336):Z =  PEEK ( - 16336)
450  IF (X = 0 OR X = 30) THEN HRIZ =  - HRIZ
460  IF (Y = 0 OR Y = 39) THEN VERT =  - VERT
470  GOTO 270
480  REM ** RING BELL AND UPDATE SCORE
490  PRINT "": REM   ** CTRL G BELL
500  IF TIME > BESTTIME THEN BESTTIME = TIME
510  HOME
520  VTAB (22)
530  PRINT "BEST TIME SO FAR =";BESTTIME
540  PRINT "TIME FOR LAST RALLEY = ";TIME
550  GOTO 40
560  END
```

INDEX

A text designed to introduce students to the fundamentals of the Applesoft BASIC language as implemented on the Apple II Plus microcomputer. The book is written in a straightforward teach-yourself style, requiring minimal extra instruction, and no previous knowledge of computing or of the Apple machine is assumed. Direct access to a microcomputer is desirable but not absolutely necessary.

The author introduces topics progressively in order of increasing complexity and includes numerous exercises for the student. These exercises are usually two part, the first involving writing out appropriate program statements (which can be done without access to a computer), the second involving keying the program statements into the Apple and testing the program with sample data. Answers are given, with full program listings for selected exercises.

- Introduces students with little or no background in computer programming to the use of Apple II Plus microcomputers.
- The major concepts and statement types of Applesoft BASIC are introduced in such a way that students can learn with the minimum of supervision or instruction.
- The text is reinforced with an excellent selection of exercises and model answers are provided for many of these.

Edward Arnold

ISBN 0 7131 3498 4