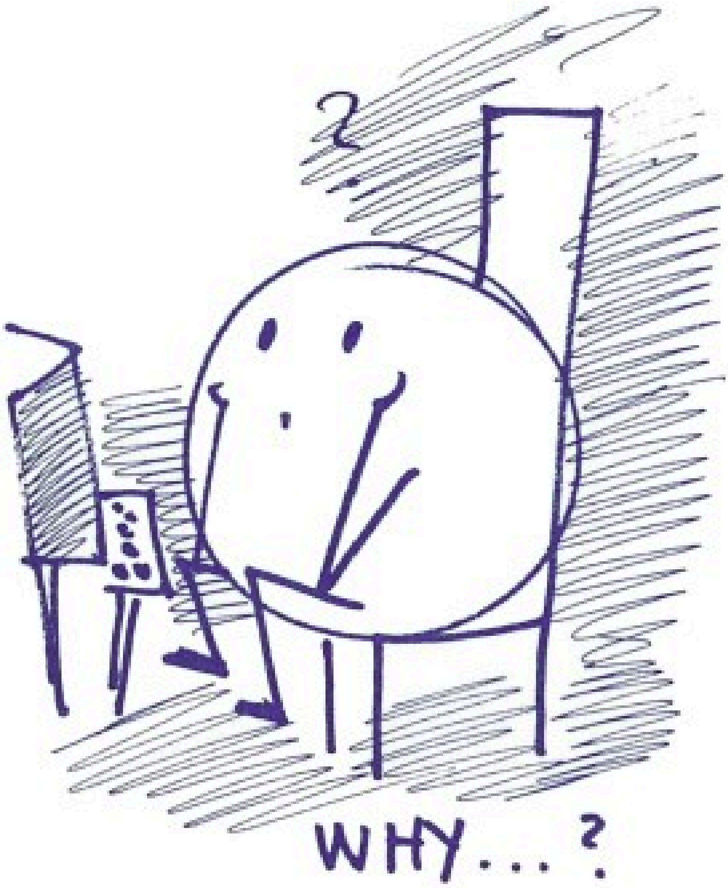
$6.95

THE AMAZING ADVENTURES OF CAPTAIN COMAL™
BOOK 2

COMAL FROM A TO Z



BORGE CHRISTENSEN PRESENTS AN OVERVIEW OF COMMODORE 64 COMAL VERSION 0.14 INCLUDING GRAPHICS AND SPRITES.

TABLE OF CONTENTS

The original manuscript was written on a Commodore computer with WordPro. It was also submitted to Toronto Pet Users Group for possible publication. This edition was edited with Paperclip running on a CBM 8096 and printed on a Starwriter printer.

ISBN 0-928411-00-1

PREFACE

The programming language COMAL (**COM**mon **A**lgorithmic **L**anguage) was designed in 1973 by Benedict Loefstedt and myself in order to make life easier and safer for people who wanted to use computers without being computer people. We combined the simplicity of BASIC with the power of Pascal.

If you take a close look at BASIC you will see that its simplicity stems mainly from its operative environment, and not from the language itself. Using BASIC, a beginner can type in one or two statements and have his small program executed immediately by means of one simple command. Line numbers are used to insert, delete and sequence statements. You do not need a sophisticated Text Editor or an ambitious Operative System Command Language. Input and output take place in a straightforward way at the terminal.

On the other hand there is no doubt that as a programming language, BASIC is hopelessly obsolete. It was never a very good language, and seen from a modern point of view it is a disaster. People who start to learn programming using BASIC may easily be led astray and, after some time, may find themselves fighting with problems that could be solved with almost no effort using programming languages more adequate to guide human thinking.

COMAL includes the gentle operative environment of BASIC and its usual simple statements, such as INPUT, PRINT, READ, etc., but it adds to it a set of statements modelled after Pascal that makes it easy to write well structured programs. Instead of leading people away from the modern effective way of programming, COMAL offers a perfect introduction to this new art.

With C64 COMAL 0.14 it is now possible for any one to become familiar with modern principles of programming. It also includes simple but effective and versatile instructions to control graphics and sprites.

Tonder, Denmark, April 22, 1984
Borge R Christensen

READ THIS FIRST

This manual briefly explains each COMAL keyword as well as assignments, expressions, procedures, parameters, and standard functions. Keywords are presented in alphabetical order. Graphics and Sprites are each presented in their own sections.

Most <items> are defined on location but a few fundamental ones are explained below:

<identifier>	A string of up to 78 characters. The leading character must be a letter, and the following may be letters, digits, or any one of the characters: apostrophe ('), [,], backslash, or left arrow (displayed as underscore on the printer).
<variable name>	An <identifier> to name a real (floating) variable, <identifier># to name an integer variable, or <identifier>$ to name a string variable.
<file name>	A <string expression> that returns a valid disk operating system file name.
<expression>	A <numeric expression> or a <string expression>. A <numeric expression> returns a numeric value (integer or real), and a <string expression> returns a string. Only <numeric expressions> that return values in the range from -32768 to 32767 can be assigned to integer variables. See also EXPRESSIONS.
<numeric constant>	A decimal representation of a number.
<string constant>	A string of characters enclosed in double quotes.
<file#>	A <numeric expression> that returns a value in the range 1-255. The COMAL System uses file #1 and #255 for system use.

<unit#>

A <numeric expression> that returns a value in the range 0-15.

<line number>

An integer in the range 1-9999.

IMPORTANT

In the syntax definitions, items in square brackets [ ] are optional. Items enclosed in braces { } are also optional, but may have several occurences.

It should be stressed that this book is neither a full formal definition nor a textbook. Though it is believed to be complete and correct it presupposes a certain knowledge about programming in general and about Commodore computers in particular. A 470 page handbook that explains and details C64 COMAL and also contains much useful additional information about Commodore computers is:

REFERENCE BOOK

COMAL Handbook by Len Lindsay

Textbooks about COMAL include:

TUTORIAL BOOKS

Beginning COMAL by Borge Christensen
Foundations in Computer Studies with COMAL by John Kelly
Structured Programming with COMAL by Roy Atherton

The newsletter about COMAL is:

NEWSLETTER

COMAL TODAY (Editor: Len Lindsay)

All are available from COMAL Users Group, U.S.A., Limited, 5501 Groveland Ter, Madison, WI 53716.

C64 COMAL 0.14 GRAPHICS

Remember to initialize the graphics system BEFORE you try any of the graphics commands (see SETGRAPHIC). You have 16 different colors available, numbered 0-15. The screen coordinates are 0-319 for the x axis and 0-199 for the y axis. The turtle's home position is in the center of the screen at position 160,99 facing upward (zero degrees heading).

BACK

BACK <distance>

Moves the turtle <distance> screen units backwards. If the pen is down (see PENDOWN), a line is drawn using the current color (see PENCOLOR).

BACKGROUND

BACKGROUND <color>

Sets the background to the color given by the value of <color> (number from 0-15). When in Hi-Res graphics the instruction is not executed, until COMAL meets a CLEAR command (see CLEAR).

BORDER

BORDER <color>

Sets the border to the color given by the value of <color>. See also BACKGROUND.

CLEAR

CLEAR

Clears the graphics screen. Does not affect the sprites.

DRAWTO

DRAWTO <x>,<y>

Draws a line from the present position of the pen to the position (<x>,<y>). The current color is used.

FILL

FILL <x>,<y>

Fills the closed area containing the position (<x>,<y>) with the current color (see PENCOLOR).

The bounds of a closed area is thus defined: a boundary point is one that has a color differet from that of the background or a point on the edge of the present frame (see FRAME).

FORWARD

FORWARD <distance>

Moves the turtle <distance> screen units forward. If the pen is down (see PENDOWN), a line is drawn using the current color (see PENCOLOR).

FRAME

FRAME <xmin>,<xmax>,<ymin>,<ymax>

Defines the frame within which the pen is active. No drawing takes place in points whose coordinates are outside the frame. However the turtle is still displayed outside the frame. The lower left corner of the frame is given by (<xmin>,<ymin>). The upper right corner is (<xmax>,<ymax>). Default frame covers the whole graphics screen: FRAME 0,319,0,199.

FULLSCREEN

FULLSCREEN

Shows the whole of the graphics screen, i.e. no text window is displayed on the upper two lines of the physical screen (unlike SPLITSCREEN).

HIDETURTLE

HIDETURTLE

Makes the turtle invisible. This makes some graphics faster.

HOME

HOME

Places the turtle in the position (160,99) heading vertical upward (zero direction).

LEFT

LEFT <angle>

Turns the turtles head <angle> degrees to the left (counter clockwise).

MOVETO

MOVETO <x>,<y>

Moves the turtle from its present position to the position (<x>,<y>) without drawing a line.

PENCOLOR

PENCOLOR <color>

Sets the color used for drawing, i.e. the color of the pen. This is also the color of the cursor and turtle, and the color in which text is displayed on the text screen. Normally <color> is an integer from 0 to 15.

PENDOWN

PENDOWN

Activates the turtles pen, i.e. the turtle leaves a trace as long as its movements are inside the present frame and the pen's color is different from that of the background (see PENCOLOR).

PENUP

PENUP

Lifts the turtles pen, i.e. it no longer leaves a trace on the screen. However, DRAWTO and PLOT work even if PENUP is set.

PLOT

PLOT <x>,<y>

Displays the position (<x>,<y>) in the current color (see PENCOLOR).

PLOTTEXT

PLOTTEXT <x>,<y>,<text>

Displays in the current color the text given by the string expression <text> on the graphics screen such that the lower left corner of the first character of <text> is placed at the position (<x>,<y>). However, the applied coordinates are set to greatest multiple of 8 less than or equal to the given values. Texts can only be displayed in Hi-Res graphics mode.

RIGHT

RIGHT <angle>

Turns the turtles head <angle> degrees to the right (clockwise).

SETGRAPHIC

SETGRAPHIC <type>

Initializes and makes the graphics screen visible. You have two graphic modes:

```
  High Resolution graphics: <type>=0
  Multicolor graphics:      <type>=1
```

In high resolution graphics you have 320*200 pixels at your disposal. The whole of the graphics screen is split up in 40*25 blocks, each of which holds 8*8 pixels. Each individual block only allows two colors to be applied at a time. One of these colors is the background. The other color is defined as soon as a pixel in the block is set. If on a later occasion a pixel inside a block is set with a different color the whole block changes to the latter one.

In multicolor graphics the resolution in the horizontal direction is only half that of high resolution, i.e. you now have 160*200 pixels at your disposal. Again the screen is divided in 40*25 blocks, but each of the them only holds 8*4 pixels. However each block can hold up to four different colours one of which is the background.

SETHEADING

SETHEADING <direction>

Turns the turtle to point at <direction> degrees clockwise from zero (vertically upward).

SETTEXT

SETTEXT

Hides the graphics screen and shows the text screen. However the graphics instructions still work on the hidden graphics screen. The result of graphics activities can easily be revealed by using the SETGRAPHIC command.

SETXY

SETXY <x>,<y>

Moves the turtle to the position (x,y). If the pen is down (see PENDOWN) a line is drawn.

SHOWTURTLE

SHOWTURTLE

Makes the turtle visible on the graphics screen. When COMAL is started a default SHOWTURTLE is executed, i.e. from start the turtle is shown on the graphics screen (see HIDETURTLE).

SPLITSCREEN

SPLITSCREEN

Displays a window into the text screen on the top two lines of the graphics screen.

TURTLESIZE

TURTLESIZE <size>

Defines the size of the turtle. The value of <size> is an integer from 0 to 10. Default value of <size> is 10.

C64 COMAL 0.14 SPRITES

Eight sprites are available for your use, numbered 0-7 (sprite number 7 is used by the system for the turtle's image). Up to 48 images can be defined. The usual 16 colors (0-15) are available.

DATACOLLISION

DATACOLLISION(<sprite>,<reset>)

This function returns a value of TRUE, if sprite no. <sprite> collides with graphics information (i.e. a non-background sprite pixel is also a non-background graphics pixel). The collision detection is automatically done by the system each time a sprite is drawn. If <reset> has a value of TRUE (1), the system resets the collision flag. If <reset> is FALSE (0), the collision flag is stored by the system for use with the next DATACOLLISION statement.

DEFINE

DEFINE <image#>,<definition>

where <image#> is an integer from 0-47, and <definition> is a string expression that has the 64 characters which defines the image (see your Commodore 64 Users Guide page 68 or the Commodore 64 Programmers Reference Guide page 131 for information about the meaning of the first 63 bytes of a sprite image definition). You can have a pool of 48 images (47 if a turtle is used) and each of these can be used as a model for any one of the 8 (7 if a turtle is used) available sprites. Not all of the 48 images need to be defined, and more than one sprite can use the same image.

HIDESPRITE

HIDESPRITE <sprite>

Sprite no. <sprite> is no longer displayed on the screen.

IDENTIFY

IDENTIFY <sprite>,<image#>

Sprite number <sprite> is given the image defined by <image#>. Imagine you have a cupboard filled with drawings of differet shapes numbered 0-47. Each time the IDENTIFY statement is used, the specified drawing (<image#>) is taken out of the cupboard and its shape is given to sprite <sprite>. The <sprite> must be an integer from 0 to 7 (the turtle is sprite number 7).

PRIORITY

PRIORITY <sprite>,<p>

If <p> is TRUE, the pixels in sprite no. <sprite> will have lower priority than the graphics pixels, i.e. the sprite will appear underneath the graphics. If <p> is FALSE, the sprite will have higher priority than the graphics.

The priority among the sprites is fixed: A sprite with a lower number has a higher priority. Thus sprite no. 0 has a higher priority than sprite no. 1 etc.

SPRITEBACK

SPRITEBACK <color-1>,<color-2>

Defines the two common colors to be used with multicolor sprites, where <color-1> and <color-2> are integers from 0-15.

SPRITECOLLISION

SPRITECOLLISION(<sprite>,<reset>)

A function that returns the value TRUE, if and only if sprite no. <sprite> has collided with another sprite. See DATACOLLISION for explanation of <reset>.

SPRITECOLOR

SPRITECOLOR <sprite>,<color>

Defines the color of sprite no. <sprite> to become <color> (0-15).

SPRITEPOS

SPRITEPOS <sprite>,<x>,<y>

Positions sprite no. <sprite> such that the upper left corner appears at the position (<x>,<y>). The bottom left corner of the screen is (0,0).

SPRITESIZE

SPRITESIZE <sprite>,<xsize>,<ysize>

If <xsize> is TRUE (1), sprite no. <sprite> is expanded to double width, if <ysize> is TRUE, the sprite is expanded to double height. The resolution is not affected by the expansions.

COMAL 0.14 KEYWORDS

ABS

A standard function. ABS(X) returns the absolute value of X.

AND

A Boolean operator that denotes logical conjugation. See also EXPRESSIONS.

APPEND

Specifies that a sequential file is opened in append mode. See also OPEN.

ASSIGNMENTS

The syntax of an assignment is

```
<variable>:=<expression>
```

If the <variable> is of type string, the <expression> must be of the same type. Type conflicts between numerics and strings are normally found and reported as program lines are entered.

The system is, however, very tolerant when numeric types (reals and integers) are concerned. A variable of type real will accept integer values and you may use variables of type integer in real expressions. An integer variable will accept any number in the range from -32768 to 32767. If a real number in that range is assigned to an integer the number is first rounded.

Numeric type incremental and decremental assignments such as:

```
<variable>:=<variable>+<expression>  and
<variable>:=<variable>-<expression>
```

may respectively be written in shorthand form:

```
<variable>:+<expression>  and
<variable>:-<expression>
```

If the keyword LET is typed in before an assignment it is ignored by the system. If the sign of equality (=)is entered instead of the sign of assignment (:=), the system automatically converts "=" into ":=".

EXAMPLES:

```
VOLUME:=LENGTH*WIDTH*HIGHT/3
COUNTER:+INCREMENT
ADDRESS$:=NAME$+"@"+STREET$+"@"+CITY$+"*"
MAX#:=32128
```

ATN

A standard function. ATN(X) returns the arctangent in randians of X.

AUTO

Makes the COMAL system generate line numbers automatically as a program is entered. Its syntax is:

```
AUTO [<line number>] [,<increment>]
```

where <increment> is a positive integer.

COMMAND	GENERATES LINE NUMBERS:
AUTO	0010, 0020, 0030, 0040, etc.
AUTO 110	0110, 0120, 0130, 0140, etc.
AUTO ,2	0010, 0012, 0014, 0016, etc.
AUTO 110,2	0110, 0112, 0114, 0116, etc.

If a valid line number is added to the word AUTO, the generated sequence of numbers will start with the number thus indicated.

If a positive integer preceded by a comma is added, the system will use this integer as an increment in line numbers.

AUTO mode is switched off by pressing the RETURN key twice in succession.

BASIC

Makes the computer switch back to the built-in BASIC interpreter. The syntax of the command is

```
BASIC
```

To return to COMAL the interpreter must be reloaded.

Note: the C64 reset function sometimes fails when the BASIC command is used. To be sure that the system is truly reset to BASIC mode press <STOP>+<RESTORE> once or twice.

CASE STRUCTURE

The CASE structure controls multiway branching. The syntax of the case structure and its individual statements is given below:

```
  CASE <case selector> [OF]
 {WHEN <choice list>
    <statement list>}
 [OTHERWISE
    <statement list>]
  ENDCASE
```

The <case selector> is an <expression>. The <choice list> is a list of <expressions>. The expressions on the <choice list> following a WHEN statement must be of the same type (real, integer, or string) as the <case selector>.

If the value of the <case selector> is equal to the value of one of the expressions on a <choice list> the corresponding <statement list> is executed.

As soon as a <statement list> has been executed, the COMAL interpreter transfers control to the statement following the ENDCASE statement, or stops if no more statements follow. If the value of the <case selector> does not match any of the expressions on the choice lists the <statement list> following OTHERWISE is executed, but if no OTHERWISE statement is present, an error message is emitted and execution of the program is stopped.

On the listing of a program statements in a <statement list> are indented automatically relative to the control statements:

```
CASE GUESS OF
WHEN 1,2,3,4,5
  COLOUR$:="RED"
  FACTOR:=1.5
WHEN 6,7,8
  COLOUR$:="YELLOW"
  FACTOR:=3
WHEN 9
  COLOUR$:="BLUE"
  FACTOR:=10
ENDCASE
```

If the <case selector> GUESS is equal to 1, 2, 3, 4, or 5, the first case is executed. If GUESS is equal to 6, 7, or 8, the second case is executed, and if GUESS is equal to 9 the last of the cases is executed.

```
CASE MONTH$ OF
WHEN "JAN","MAR","MAY","JUL","AUG","OCT","DEC"
  PRINT "THE MONTH HAS 31 DAYS."
WHEN "APR","JUN","SEP","NOV"
  PRINT "THE MONTH HAS 30 DAYS."
WHEN "FEB"
  IF YEAR MOD 4=0 THEN
    PRINT "THE MONTH HAS 29 DAYS"
  ELSE
    PRINT "THE MONTH HAS 28 DAYS"
  ENDIF
OTHERWISE
  PRINT "OLD MAN TURNS OVER IN HIS GRAVE."
ENDCASE
```

CAT

Displays the contents of the diskettes. Its syntax is

```
CAT [<drive no.>]
```

The command

```
CAT
```

causes the system to display catalogs of all diskettes mounted in disk drive unit 8. The command

```
CAT 0
```

shows the catalog of the diskette in drive 0, unit 8.

CHAIN

Loads a program stored on disk and runs it. Its syntax is

```
CHAIN <file name> [,<unit no.>]
```

If <unit no.> is not specified, disk unit number 8 is used. Programs already in main storage will be deleted when the CHAIN statement is invoked. Only programs stored by means of the SAVE command can be retrieved via CHAIN.

```
CHAIN "UPDATE"
```

loads the program named "UPDATE" from drive 0, unit 8, and runs it. See also SAVE and LOAD.

CHR$

A standard function. CHR$(X) returns the character whose ASCII value is X.

CLOSE

Closes data files. Its syntax is

```
CLOSE [FILE] [<file number>]
```

The statement (or command)

```
CLOSE
```

closes all files that have been opened. The following statement (or command) closes file number 3 only.

```
CLOSE 3
```

The keyword FILE is added automatically by the interpreter if not entered by the user. See also OPEN, INPUT, PRINT, READ, WRITE.

CLOSED

If the keyword CLOSED terminates the procedure heading, all variables in the procedure will be local. Normally this is only the case with the parameters.

```
  PROC WINDOW(X,Y) CLOSED
    SCREEN(X,1)
    FOR I:=1 TO Y-X+1 DO ERASE'LINE(I)
    SCREEN(X,1)
  ENDPROC WINDOW
  //
  PROC SCREEN(L,C) CLOSED
    X:=984+L*40
    POKE 209,X MOD 256 //LINE LOW BYTE
    POKE 210,X DIV 256 //LINE HIGH BYTE
    POKE 211,C-1 //COLUMN
  ENDPROC SCREEN
  //
  PROC ERASE'LINE(L) CLOSED
    SCREEN(L,1)
    FOR I:=1 TO 40 DO PRINT " ",
  ENDPROC ERASE'LINE
```

The variables X, Y, L, C, and I are all local, X, Y, L, AND C because they are parameters and I because the procedures are closed. Thus the X used in SCREEN and the X used in WINDOW are different objects. The same goes for I in WINDOW and ERASE'LINE. See also PROCEDURES AND PARAMETERS and FUNC.

CON

Restarts a program which has been stopped.

```
  CON
```

Due to the internal linking of structures in a COMAL program, the CON command cannot be used after deletion or insertion of statements or introduction of new variables. See also STOP.

COS

A standard function. COS(X) returns the cosine of X (X in radians).

DATA

A DATA statement is used to hold numeric or string constants that may be retrieved in a READ statement. Its syntax is:

```
DATA <value> {,<value>}
```

where <value> is a <numeric constant> or a <string constant>. See also EOD, READ, and RESTORE.

```
REPEAT
  READ NAME$,TEL
  FOUND:=(THISNAME$=NAME$)
UNTIL FOUND OR EOD
DATA "COLLINS",23,"JACOBS",34,"HUDSON",45
DATA "KILROY",14,"ATHERTON",10,"BRAMER",15
```

DEL

Removes one or more lines from a program in main storage:

```
DEL [<line number> [-[<line number>]]]  or
DEL -<line number>
```

COMMAND	RESULTS
DEL 100	Removes line 100 from program
DEL 100-200	Removes lines between 100 and 200 inclusive
DEL -300	Removes all lines up to and including 300
DEL 300-	Removes all lines numbered 300 or greater

Important note. A line cannot be removed by just giving its line number. The DEL command should not be confused with the DELETE command which is used to remove files from the disk.

DELETE

Removes files from a disk. Its syntax is

```
DELETE <file name>
```

The <file name> must include the drive number. Thus the command

```
DELETE "0:MYPROG"
```

deletes the file "MYPROG" stored on the diskette in drive 0, unit 8.

DIM

Declares strings and arrays of numerics and strings. Its syntax is

```
  DIM <declaration> {,<declaration>}
```

A <declaration> could be a <numeric declaration> as in

```
  DIM TABLE(-1:100)
```

or a <string declaration> as in

```
  DIM NAME$(0:20) OF 30
```

Since the DIM statement is very versatile and powerful, it is not all that simple to give a detailed description of its syntax. Instead we shall look at some examples. The statement

```
  DIM TABLE(-1:100),MARKS(1000:1500,8:10)
```

declares an array of real numbers, named TABLE, with indices ranging from -1 to 100, and a two dimensional numeric array, named MARKS, with indices ranging from 1000 to 1500 and 8 to 10. You may use any <numeric expression> for lower bound and upper bound, as long as the value returned for the lower one is smaller than or equal to the value returned for the upper one. Non-integer values are truncated. If no lower bound is given the interpreter uses 1 in its place. Thus the statement

```
  DIM JOBCODE(100)
```

declares an array of numerics with indices ranging from 1 to 100 and is totally equivalent to

```
  DIM JOBCODE(1:100)
```

The statement

```
  DIM NAME$ OF 30, ANSW$ OF 3
```

declares two single string variables such that the first one may hold up to 30 characters and the second one up to 3 characters. Single string variables must be declared. The following statement

```
  DIM PUPIL$(30:100,8:10) OF 30
```

declares an array of strings with indices ranging from 30 to 100 and 8 to 10 where each component may hold up to 30 characters.

An array may have any number of dimensions.

DIV

An operator that denotes integer division. See also EXPRESSIONS.

DO

used with FOR and WHILE statements. See FOR and WHILE.

EDIT

Displays a list of the program presently in workspace, but without the structured indentation invoked by the LIST command. The syntax is:

```
  EDIT [<line number> [-[<line number>]]]  or
  EDIT -<line number>
```

The EDIT command is used when editing to avoid including indent spaces on continued screen lines caused by the automatic indentation of lines that are wrapped around. See also LIST.

ELIF

Used with the IF statement. See IF.

ELSE

Used with the IF statement. See IF.

END

Stops execution of a program. See also STOP.

ENDCASE

Terminates the last block in a CASE multiway branching structure. See CASE.

ENDFOR

Terminates the block controlled by a FOR statement. See FOR.

ENDFUNC

Terminates the definition of a user defined function. See FUNC.

ENDIF

Terminates the last block of statements in an IF branching. See IF.

ENDPROC

Terminates the definition of a procedure. See PROCEDURES AND PARAMETERS.

ENDWHILE

Terminates the block of statements controlled by a WHILE statement. See WHILE.

ENTER

Enters a program stored on disk or tape into workspace:

```
ENTER <file name> [,<unit no.>]
```

Default value of <unit no.> is 8. The command

```
ENTER "0:MYPROG.L",9
```

is used to enter the program "MYPROG" found on drive number 0, unit number 9, whereas the command

```
ENTER "YOURPROG",1
```

retrieves the program "YOURPROG" found on the cassette in unit number 1 (datasette).

Only programs stored by means of the LIST command may be retrieved with the ENTER command.

Important note. Program lines that are taken in by the ENTER command are merged into an existing program in the same way as lines typed from the keyboard. See also LOAD, LIST and SAVE.

EOD

A standard Boolean function. EOD returns a value of TRUE (numeric 1) if the last element in a data queue has been read, otherwise a value of FALSE (numeric 0) is returned. See also READ.

EOF

A standard Boolean function. EOF(X) returns a value of TRUE (numeric 1) if the end-of-file in the sequential file X has been reached, otherwise a value of FALSE (numeric 0) is returned. See also READ and INPUT.

```
OPEN 2,"PERSONS",READ
WHILE NOT EOF(2) DO
  READ FILE 2: NAME$,ADR$,CITY$
  PRINT NAME$
  PRINT ADR$
  PRINT CITY$
ENDWHILE
CLOSE
```

ESC

The function ESC returns a value of TRUE (numeric 1) if the STOP key is depressed, otherwise it returns a value of FALSE (numeric 0). The ESC function is not active unless a TRAP ESC- statement is in effect. See also TRAP.

EXEC

Indicates a procedure call. The syntax of a procedure call is

```
[EXEC] <identifier>(<actual parameters list>)
```

The normal way of calling a procedure is by simply stating the name of the procedure followed by a parameter list, if any. But for sake of compatibility with earlier versions of COMAL the dummy keyword EXEC may still be used. Normally the EXEC is suppressed on the listing of the program, but by using the SETEXEC command (see SETEXEC) you can force the interpreter to display it.

The following statements

```
PRINTOUT(NAME$,ADDRESS$)
EXEC PRINTOUT(NAME$,ADDRESS$)
```

are equivalent. They are both calling the procedure PRINTOUT passing the parameters NAME$ and ADDRESS$. See PROCEDURES AND PARAMETERS.

EXP

A standard function. EXP(X) returns the value of e (nat. log. base) to the power of X (thus being the inverse of nat. log.)

EXPRESSIONS

A <numeric expression> can contain constants, variables, and numeric functions, used with parentheses and the following operators according to the usual rules of mathematics:

+	monadic +	+A
-	monadic -	-A
↑	power	A↑B
*	multiplication	A*B
/	division	A/B
DIV	integer division (see below)	A DIV B
MOD	remainder from division (see below)	A MOD B
+	addition	A+B
-	subtraction	A-B

If A and B are integers then A MOD B is the so called principal remainder from division of A by B, i.e. the smallest non-negative integer R such that

```
A = B*Q + R
```

and A DIV B is the quotient Q.

Numeric values may be compared by means of the following relational operators:

<	means "less than"
<=	means "less than or equal to"
=	means "equal to"
>=	means "greater than or equal to"
>	means "greater than"
<>	means "not equal to"

Numeric expressions may be used as Boolean expressions. A numeric value equal to zero is interpreted as FALSE, whereas any value other than zero is interpreted as TRUE. A logical operation returns a numeric 1 for TRUE and 0 for FALSE.

The following Boolean operators are available:

NOT logical negation. NOT A returns a value of FALSE, i.e. numeric 0, if A has a value of TRUE, i.e. a numeric value different from zero, but a value of TRUE (numeric 1) if A has a value of FALSE (is equal to zero).

AND logical conjunction. A AND B returns a value of TRUE if A and B are both TRUE, otherwise a value of FALSE is returned.

OR logical disjunction. A OR B returns a value of FALSE if A and B are both FALSE, otherwise a value of TRUE is returned.

A <string expression> may consist of string constants, string variables, string array elements, or string functions concatenated by means of the + sign. String expressions may be compared (lexicographical order) by means of the operators:

<	means "comes before"
<=	means "comes before or is equal to"
=	means "is equal to"
>=	means "comes after or is equal to"
>	means "comes after"
<>	means "is not equal to"

Note that strings with relational operators make up expressions that return numerical values; 1 for TRUE and 0 for FALSE.

IN is used for string pattern matching. The expression A$ IN B$ returns a value of zero (i.e. FALSE) if A$ is not found as a substring of B$, but if A$ is found as a substring of B$ the expression returns the position of the first matching character.

If NAME$ has a value of "LOTTIE CHRISTENSEN" then the expression

```
"TIE" IN NAME$
```

returns a value of 4 (TRUE).

The priority of the above operators is:

```
                                  (power)
   *  /  DIV  MOD
   +  -
   <  <=  =  >=  >  <>  IN
   NOT
   AND
   OR
```

FALSE

To improve the readability of programs, two constants, TRUE and FALSE, are predefined. TRUE is equal to 1, and FALSE is equal to 0.

FILE

See OPEN, CLOSE, INPUT, PRINT, READ, and WRITE.

FOR STRUCTURE

The syntax of the FOR loop structure is:

```
  FOR <for range> [<step>] DO
    <statement list>
  ENDFOR [<control variable>]
```

where <for range> is

```
  <control variable>:=<initial value> TO <final
value>
```

and <step> is

```
  STEP <step value>
```

The <control variable> is a <numeric variable>, and <initial value>, <final value>, and <step value> are <numeric expressions>.

The <control variable> following the keyword ENDFOR has been bracketed to indicate that it is supplied automatically by the interpreter if not entered by the programmer. To ensure compatibility with earlier versions of COMAL the keyword NEXT is accepted on entry as well as ENDFOR. In a listing the keyword ENDFOR is displayed.

```
FOR X:=1 TO 5 DO
  SUM:=SUM+X
  PRINT SUM;
ENDFOR X
```

First the control variable X is set to 1 and the two statements in the range of the loop are executed. Then X is set to 2, and the statements are executed again. This goes on as long as X is not greater than the final value 5. When X assumes a value of 6 execution of the loop is stopped and the interpreter starts on the statement following the ENDFOR statement. Note that X has a value of 6, i.e. <final value>+1, when the loop terminates. Also note that this value is not actually used in the loop.

```
FOR N:=1 TO 10 STEP 2 DO
  SUM:=SUM+N
  PRINT SUM
ENDFOR N
```

In this example N assumes the values 1, 3, 5, 7, 9, and 11, since a step value of 2 is prescribed. Note that the control variable N has an unused value of 11 when execution of the loop terminates.

```
FOR P:=10 TO 1 STEP -1 DO
  PRINT TEXT$(1:P)
ENDFOR P
```

The statement in the loop is executed for P equal to 10, 9, 8, ..., 1. The termination value of P is 0 and not used in the loop.

A short FOR loop is available. Its syntax is

```
FOR <for range> [<step>] DO <statement>
```

No ENDFOR statement is allowed in this case. The one-line FOR statement may also be used as a command.

```
FOR P:=10 TO 1 STEP -1 DO PRINT TEXT$(1:P)
```

This loop is functionally equivalent to the previous one only this time the short form is used.

```
FOR T:=1 TO 750 DO NULL
```

This loop waits till COMAL has counted from 1 to 750.

FUNCTIONS

The FUNC statement is used as the first statement - or head - of any user defined function. The syntax is

```
FUNC <function identifier> <head appendix>
  <function body>
ENDFUNC [<function identifier>]
```

The <function identifier> is a <variable identifier> and the <head appendix> is specified as:

```
[(<formal parameter list>)] [CLOSED]
```

The <function body> is made up of COMAL statements.

A function value must be returned in a RETURN statement (see RETURN), and at least one such statement must be present in the <function body>.

The <function identifier> following ENDFUNC is supplied automatically by the system during the prepass if not entered by the programmer.

Note: if you are not very familiar with multi-line functions and parameters, it might be advisable that you read the section about PROCEDURES AND PARAMETERS before continuing the present one.

```
PRINT DISTANCE(10,-4)
```

The statement above calls the function below:

```
  FUNC DISTANCE(X,Y)
    IF X<=Y THEN
      RETURN Y-X
    ELSE
      RETURN X-Y
    ENDIF
  ENDFUNC DISTANCE
```

The values of the actual parameters 10 and -4 are assigned ("passed") to the formal parameters X and Y, respectively, and the value 14 is returned. The PRINT statement displays 14.

```
  FUNC POS(A$,B$)
    RETURN A$ IN B$
  ENDFUNC POS
```

This function represents nothing but a renaming of the IN operator. In some cases such a renaming could contribute to a better documentation.

```
  FUNC GCD#(X#,Y#)
    IF (X# MOD Y#)=0 THEN
      RETURN Y#
    ELSE
      RETURN GCD#(Y#,X# MOD Y#)
    ENDIF
  ENDFUNC GCD#
```

This function returns the GCD (Greatest Common Divisor) of two integers. Note that the function itself is of type integer, and that it calls itself recursively.

```
  FUNC VALUE(A$) CLOSED
    LN:=LEN(A$)
    ONES:=ORD(A$(LN))-ORD("0")
    IF LN=1 THEN
      RETURN ONES
    ELSE
      RETURN ONES+VALUE(A$(1:LN-1))*10
    ENDIF
  ENDFUNC VALUE
```

This function also calls itself recursively from the expression in the last RETURN statement.

```
FUNC HASH(A$,HASHER) CLOSED
  LN:=LEN(A$); T:=0
  FOR I:=1 TO LN DO T:+ORD(A$(I))
  RETURN T MOD HASHER
ENDFUNC HASH

FUNC MEAN(N,REF A()) CLOSED
  SUM:=0
  FOR I:=1 TO N DO SUM:+A(I)
  RETURN SUM/N
ENDFUNC
```

This function uses an array A passed as a parameter by reference. See also PROCEDURES AND PARAMETERS and CLOSED.

GOTO

The syntax of a GOTO statement is:

```
GOTO <label>
```

where <label> is an <identifier>. The GOTO statement transfers control to a <label statement> thus defined:

```
<label>:

IF FATALERROR THEN
  PRINT "FATAL ERROR. CANNOT CONTINUE."
  GOTO HALT
ENDIF
...
HALT:
STOP
```

Using a GOTO statement you can jump out of any structure, but not out of a procedure. If you try to jump into a structure the result is unpredictable. Jumping into a procedure may cause a system break down.

IF STRUCTURE

The IF statement is the head of the IF structure that controls conditional branching. The syntax of the IF structure and the statements that go with it is shown in the following diagram:

```
  IF <logical expression> [THEN]
    <statement list>
 {ELIF <logical expression> [THEN]
    <statement list>}
 [ELSE
    <statement list>]
  ENDIF
```

where <logical expression> is the same as <numerical expression>. The keyword THEN is supplied automatically by the system if not entered by the user. The lines in a <statement list> are automatically indented by the interpreter on the program listing.

In COMAL you also have a short form of the IF statement. Its syntax is:

```
  IF <logical expression> THEN <statement>
```

Note that no ENDIF is allowed in this case. On the other hand the keyword THEN must be entered.

```
  IF I<=J THEN
    W:=A(I); A(I):=A(J); A(J):=W
    I:=I+1; J:=J-1
  ENDIF
```

If the expression I<=J evaluates to TRUE (numeric 1) the statement list between IF and ENDIF is executed. If, however, it returns FALSE (numeric 0) the statement list is skipped and control is transferred to the statement following ENDIF.

```
  IF TRY<3 THEN
    PRINT "NO, TRY AGAIN"
  ELSE
    PRINT "NO, THE ANSWER IS ",RESULT
    PRINT "TYPE THAT!"
  ENDIF
```

If the expression TRY<3 evaluates to TRUE, the statement between IF and ELSE is executed, but if it returns the value FALSE, the statements between ELSE and ENDIF is executed. In both cases control is then transferred to the statement following ENDIF.

```
D:=B*B-4*A*C
IF D>0 THEN
  PRINT "TWO REAL ROOTS:"
  PRINT "X1 = ",(-B+SQR(D))/2/A
  PRINT "X2 = ",(-B-SQR(D))/2/A
ELIF D=0 THEN
  PRINT "ONE REAL ROOT:"
  PRINT "X = ",-B/2/A
ELSE
  PRINT "DISCRIMINANT NEGATIVE"
  PRINT "NO REAL ROOTS."
ENDIF
```

If the expression D>0 returns the value TRUE the first three-statement list is executed, and the rest is skipped. If, however, it is evaluated to FALSE, the interpreter evaluates the expression D=0 following ELIF. It that appears to be TRUE, the second statement list is executed. If the second expression also has a value of FALSE, execution finally falls through to the last statement list, i.e. the one following the ELSE statement. Note that never more than one statement list is executed. This means that if two expressions may become TRUE, only the statement list following the first of them is executed.

```
IF OBS<10 THEN
  FREQUENCY(1):+1
ELIF OBS<20 THEN
  FREQUENCY(2):+1
ELIF OBS<30 THEN
  FREQUENCY(3):+1
ELIF OBS<40 THEN
  FREQUENCY(4):+1
ELSE
  FREQUENCY(5):+1
ENDIF
```

In this example it is utilized that one <statement list> at most is executed. If it is TRUE than OBS<10 all the rest of the Boolean expressions are also TRUE, but only FREQUENCY(1) is increased by 1. If on the other hand it is TRUE that 10<=OBS and OBS<20 only the second assignment is executed. It is easy to see how this could be used in statistics.

```
  IF CHAR$ IN SET'OF'LETTERS$ THEN
    IF CHAR$ IN SET'OF'VOWELS$ THEN
      VOWELS:+1 //ANOTHER VOWEL
    ELSE
      CONSONANTS:+1 //ANOTHER CONSONANT
    ENDIF
  ELIF CHAR$=" " THEN
    WORDS:+1 //ANOTHER WORD
  ELIF CHAR$ IN SET'OF'DIGITS$ THEN
    DIGITS:+1 //ANOTHER DIGIT
  ELSE
    SPECIALS:+1 //ANOTHER SPECIAL
  ENDIF
```

```
  IF JOB=3 THEN PRINTOUT
```

is functionally equivalent to

```
  IF JOB=3 THEN
    PRINTOUT
  ENDIF
```

In both cases the procedure PRINTOUT is called if JOB has a value of 3.

IN

A Boolean operator used for string matching. See also EXPRESSIONS.

INPUT

Used to fetch data from keyboard. Its syntax is

```
  INPUT [<prompt>:] <input list> [<print end>]
```

where <prompt> is a <string expression>, <input list> is a list of variable identifiers, and <print end> is a semicolon (;).

```
  INPUT MAXNUMBER
```

When this statement is executed, the system displays the sign "?" and waits for the user to enter a number and press the RETURN key. The number typed in is assigned as a value to MAXNUMBER.

```
  INPUT "ENTER NAME: ": NAME$
```

When this statement is executed the system displays the user defined prompt

```
  ENTER NAME:
```

and pauses to let the user type in a string to be assigned as a value to the variable NAME$.

```
  INPUT NAME$,AGE
```

When this statement is executed the system displays its standard prompt "?" and pauses. The user is expected to type in a string and press the RETURN key. The string is then assigned to NAME$ and the system submits another "?" on the same line and pauses to let the user type in a number.

```
  INPUT A,B,C
```

This statement will ask the user to enter three numbers. The following options may be chosen: You can enter three numbers like

```
  5 80 34
```

and then press RETURN. The variable A is then set to 5, B to 80, and C to 34. You can also enter the three numbers in the following manner:

```
  5,80,34
```

and then press RETURN. Finally you may obtain the same result by entering 5 and press RETURN; then 80 and press RETURN; finally 34 and press RETURN. In the first two cases only one "?" is displayed, in the last case three "?" are submitted.

```
INPUT "FROM: ":FIRST$;
INPUT "  TO: ":LAST$
```

The semicolon terminating the first statement prevents the carriage return and linefeed after the first string has been typed in. The result of a program-user dialog might look like this:

```
FROM: 12.DEC.80  TO: 23.DEC.80
```

The RETURN key was pressed after each entry.

Note that a string variable in an <input list> will pick up all characters entered from the keyboard including commas and quotemarks. Therefore you can not have more than one string variable in the list, and it must always be the last one (unless the user hits the RETURN key after each string requested).

INPUT FILE

Used to retrieve data from a file that was created using PRINT FILE. It will also allow characters to be read directly off the screen. The syntax of an INPUT FILE statement is:

```
INPUT FILE <file#>[,<rec#>]:<input list>[<end>]
```

where <input list> is a list of variable identifiers, <rec#> is a <numeric expression> and <end> is comma (,) or semicolon (;).

```
OPEN FILE 3,"MYDATA",READ
REPEAT
  INPUT FILE 3: LINE$
  PRINT LINE$
UNTIL EOF(3)
CLOSE
```

This program above reads and displays the contents of the sequential file "MYDATA". The following program reads the screen line by line and prints a hard copy of its contents:

```
VIDEO:=3
OPEN FILE VIDEO,"",UNIT 3,READ
SELECT "LP:"
FOR ROW:=1 TO 25 DO
  INPUT FILE VIDEO: TEXT$
  PRINT TEXT$
ENDFOR ROW
CLOSE VIDEO
SELECT "DS:"
```

IDENTIFIERS

Used to name variables, labels, functions, and procedures. An identifier may contain as many as 78 characters, all significant. The first character must be a letter, the following may be letters, digits, or any one of the characters: apostrope ('), [,], backslash, or left arrow.

Here are some valid identifiers:

```
MAXNUMBER, HOUSENO, NUMBER'OF'VOWELS, N1, N2,
N3, CREATE'RECORD, GET'DIGIT
```

INT

A standard function. INT(X) returns the integer part of X, i.e. the greatest integer less than or equal to X.

KEY$

A standard function. It returns the first ASCII character in the input buffer. If no key has been depressed, a CHR$(0) is returned.

```
PROC GET'CHAR(REF T$)
  T$:=CHR$(0)
  WHILE T$=CHR$(0) DO T$:=KEY$
ENDPROC GET'CHAR
```

LABELS

Used as a jump address for a GOTO statement. The syntax of a label statement is

```
<identifier>:
```

Note that GOTO <line number> is not allowed.

```
IF BREAK THEN GOTO HALT
...
HALT:
STOP
```

If BREAK assumes a value of TRUE (value not equal to 0) control is transferred to the label statement. See also GOTO.

LEN

A standard function. LEN(X$) returns the current length (number of characters) of the string value of X$.

LINEFEED

The command

```
LINEFEED+
```

makes the system emit a linefeed after each carriage return, when output is to the printer. The command

```
LINEFEED-
```

disables this facility, i.e. no linefeed is sent out after a carriage return. Default mode is LINEFEED-.

LIST

Displays a program or a part of a program residing in workspace. The syntax is:

```
LIST [<line number>[-[<line number>]]]  or
LIST -<line number>
```

where <name> is the name of a function or a procedure.

COMMAND	RESULT
LIST	List the whole program
LIST 100	List line numbered 100
LIST 100-200	List all lines between 100 and 200 inclusive
LIST -300	List all lines up to and including 300
LIST 300-	List all lines numbered 300 or greater

The LIST command may also be used to store programs on disks or tapes. The command

```
LIST "MYPROG.L"
```

stores a program now in main storage on disk as a program file with the name of "MYPROG.L". The program is stored as source code, and may therefore later be merged with another program in main storage (see ENTER). Since the LIST command handles source code directly, this version is also permitted:

```
LIST 100-200 "YOURPROG.L"
```

In this case line 100-200 are stored in a file named "YOURPROG.L". In order to easily distinguish files LISTed to disk from those SAVEd to disk, it is suggested to end the file name with .L.

If another device than disk unit no. 8 is used, <unit no.> must be added to the command.

A program that has been stored by the LIST command has type SEQ and may be opened as any other sequential file and read by an INPUT FILE statement. See also PRINT FILE, ENTER, and EDIT.

LOAD

Retrieves programs from disk or tape. Its syntax is

```
LOAD <file name> [,<unit no.>]
```

The command

```
LOAD "MAINPROG"
```

will load the program "MAINPROG" into workspace. If you want to retrieve the program from a device other than disk unit no. 8, a unit no. must be specified:

```
LOAD "YOURPROG",1
```

will load the program "YOURPROG" from cassette into workspace. See also CHAIN, SAVE, LIST, and ENTER.

LOG

A standard function. LOG(X) returns the natural logarithm of X.

MOD

An operator that returns the remainder from integer division. See also EXPRESSIONS.

NEW

Clears the whole workspace of program and data. Its syntax is

```
NEW
```

NEXT

Automatically converted into ENDFOR by the interpreter. See also ENDFOR, FOR.

NOT

A Boolean operator that denotes negation. See also EXPRESSIONS.

NULL

The NULL statement does nothing. Its syntax is

```
NULL
```

It might seem a bit strange or even luxurious to have a "no-op" statement like that to perform the "empty action", but it can be inserted in some special cases to satisfy the syntax of COMAL. The example below shows how:

```
FOR I:=1 TO 750 DO NULL //WAIT
```

OF

Ends the CASE header statement and is part of the declaration of string variables or string arrays. See also CASE and DIM.

OPEN

Opens and assigns reference numbers to files. Its syntax is

```
OPEN FILE <file#>,<filename>[,<dev>][,<type>]
```

<file#> is a <numeric expression> that must return a value from 1-255 (but the COMAL System reserves files 1 and 255 for system use), <dev> is

```
UNIT <unit#> [,<secondary addr>]
```

where <secondary addr> is a <numeric expression> that must return a value from 0-15. Finally <type> is READ for sequential reading, WRITE for sequential writing, APPEND for continued sequential writing, or RANDOM <record length> for reading to or writing from a direct access file (random file), where <record length> is a <numeric expression> that must return a positive value.

```
  OPEN FILE 3,"MARKS",READ
```

assigns the file "MARKS" as file number 3. The keyword READ indicates that a sequential file is referred to, and that data may be retrieved from it, starting from the beginning of the file.

```
  OPEN FILE 4,"@0:MARKS",WRITE
```

The file "MARKS" is assigned file number 4. The keyword WRITE indicates that a sequential file is referred to, and that data may be stored in it, starting from the beginning of the file. The "@0:" token indicates that if the file exists already then it may be overwritten. The same effect may be obtained by using these statements:

```
  DELETE "0:MARKS"
  OPEN FILE 4,"MARKS",WRITE
```

The keyword APPEND indicates that a sequential file is referred to, and that data may be stored in it, starting from the end of the existing file, thus appending more data to it.

```
  OPEN FILE 6,"MARKS",APPEND
```

The file "MARKS" is assigned file number 6.

```
  OPEN FILE 3,"CLIENTS",RANDOM 250
```

With this statement the direct access file "CLIENTS" is signed on for both reading and writing. The constant 250 following the keyword RANDOM indicates that each record can be up to

250 bytes long. See also CLOSE, INPUT, PRINT, READ, WRITE.

OR

A Boolean operator that denotes disjunction. See EXPRESSIONS.

ORD

A standard function. ORD(X$) returns the ASCII value of the first character held by X$.

OTHERWISE

Used in the CASE structure to indicate a default case. See CASE.

PASS

Passes strings to the disk drive. The strings are interpreted as commands by the disk operating system (see your disk manual for disk commands). Its syntax is

```
  PASS <string expression>
```

```
  PASS "NO:CONNIE'S DISK,01"     passes a format
command to the disk
```

PEEK

A standard function. PEEK(X) returns the contents (0-255) of a memory location X (X in the range 0-65535) in decimal representation.

POKE

Assigns values to specified locations in memory. Its syntax is:

```
  POKE <location>,<contents>
```

where <location> is a <numeric expression> that must return a value from 0-65535, and <contents> is a <numeric expression> that must return a value from 0-255 (one byte).

```
  POKE 650,128            makes C64 keys repeat
```

PRINT

Outputs data to the screen or the printer. Its syntax is

```
  PRINT [<output list>] [<end>]
```

where <output list> is

```
  <print element> {<separator> <print element>}
```

The <print element> is an <expression> or the TAB function, and <separator> is either a comma (,) or a semicolon (;). If a semicolon is used an extra space is output between one <print element> and the next; if a comma is used no extra spaces are output unless otherwise stated in a ZONE statement (see ZONE). The <end> is the same as <separator>.

```
  PRINT "THIS IS THE ",3,". TIME"
```

outputs

```
  THIS IS THE 3. TIME
```

The same output results from

```
  PRINT "THIS IS THE";3,". TIME"
```

The next statement:

```
  PRINT "PUPIL ",NO," NAME IS ",NAME$(NO)
```

may output the following

```
  PUPIL 5 NAME IS ROY MANNING
```

The same output may be produced by

```
  PRINT "PUPIL";
  PRINT NO," NAME IS";
  PRINT NAME$(NO)
```

Note the use of semicolon as <end> in this case. If comma is used you get

```
  PRINT "PUPIL ",
  PRINT NO," NAME IS ",
  PRINT NAME$(NO)
```

PRINT FILE

Stores data on disk or tape. Its syntax is

```
 PRINT FILE <file#>[,<rec#>]:<print list>[<end>]
```

<print list> and <end> are as specified for PRINT, <rec#> is a <numeric expression>. A file that has been created using PRINT FILE is of type SEQ and data from it may be retrieved by means of INPUT FILE.

```
OPEN FILE 4,"PERSONS",UNIT 1, WRITE
FOR NO:=1 TO MAXNO DO
  PRINT FILE 4: NAME$(NO)
  PRINT FILE 4: ADDR$(NO)
  PRINT FILE 4: PAYCD(NO)
ENDFOR NO
CLOSE
```

The program stores data sequentially on a cassette in the file signed on as number 4. The data thus stored may be retrieved by means of the following:

```
OPEN FILE 6,"PERSONS",UNIT 1, READ
FOR J:=1 TO MAX DO
  INPUT FILE 6: NAME$(J)
  INPUT FILE 6: ADDR$(J)
  INPUT FILE 6: PAYCD(J)
ENDFOR J
CLOSE
```

Normally PRINT FILE and INPUT FILE are only used for sequential data files on cassette. See also READ, WRITE, and OPEN.

PRINT USING

Formats output of numbers. The syntax is

```
PRINT USING <format info>: <using list>[<end>]
```

where <format info> is a <string expression> and <end> is as specified for PRINT. The <using list> is

```
<numeric expression> {,<numeric expression>}
```

The <format info> can contain texts and format fields. A format field is a string that serves as a model for the printout af numeric values. The hash mark (#) reserves a digit place, the dot (.) specifies the location of the decimal point, if any, and a minus sign can be introduced to be displayed if the value of the number is negative.

```
  PRINT USING "  ###    ####.##": A,B
```

If A equals 23.6 and B equals 234.567 the following output is produced:

```
   24    234.57
```

If A is equal to 1234 and B has a value of 546 the following output is produced:

```
  ***    546.00
```

with the three *'s indicating that there is an overflow in the format.

```
  PRINT USING "THE ROOT IS: -##.##": -B/2/A
```

If B is equal to 15.748 and A is equal to 7.2 the statement produces the following output:

```
  THE ROOT IS:  -1.09
```

If B equals 234.67 and A is equal -23.3 the statement produces this output:

```
  THE ROOT IS:  5.04
```

PROCEDURES AND PARAMETERS.

The PROC statement is used as the first statement - or head - of any user defined procedure. The syntax of a procedure is

```
  PROC <procedure identifier> <head appendix>
    <procedure body>
  ENDPROC [<procedure identifier>]
```

The <head appendix> is specified as

```
  [(<formal parameter list>)] [CLOSED]
```

The <procedure identifier> is an <identifier>, the <procedure body> is made up of COMAL statements. The <procedure identifier> following

ENDPROC is supplied automatically by the system during prepass if not entered by the programmer.

The <formal parameter list> is specified as

```
  <formal parameter> {,<formal parameter>}
```

where a <formal parameter> could be either

```
  [REF] <variable identifier>  or
  REF <variable identifier>({,})
```

If the keyword REF is used before a parameter it is passed by reference, otherwise it is passed by value. Arrays of any type can only be passed by reference.

Example: A procedure that starts with this statement

```
  PROC TRY(I,J)
```

called with:

```
  TRY(FIRST,LAST)
```

In this case the identifiers I and J in the procedure head are formal parameters, and a value is assigned to each of them when the procedure is called. The identifiers FIRST and LAST referred to in the calling statement are actual parameters and must be defined whenever the statement comes to be executed. During the procedure call, I is assigned the value of FIRST (the value of FIRST is "passed" to I), and J is assigned the value of LAST. Since actual values are passed, I and J are called value parameters.

But there is more to it. I and J will be treated as local variables to the procedure TRY, and that means that they will not be known to the "world" outside the procedure, and therefore they will not be confused with variables I and J, if any, in other parts of the program. Also when the procedure is finished any trace of local variables is removed.

Actual parameters to be passed by value may be constants, variables, or expressions, as long as they are ready to "deliver a value" on request, i.e. whenever a call is invoked. The procedure TRY might be called by statements like

```
  TRY(1,9)  or  TRY(P-1,P+L-1)

  PROC BACKWARDS(W$)
    LN:=LEN(W$); B$:=""
    FOR I:=LN TO 1 STEP -1 DO B$:+W$(I)
  ENDPROC BACKWARDS
```

The above procedure is called from these mainlines:

```
  DIM B$ OF 30
  INPUT "ENTER WORD (MAX. 30 CHAR.): ": B$
  BACKWARDS(B$)
  PRINT B$
```

The value of B$ is passed to W$ during the call. Note that W$ is not declared explicitly. When a string variable is used as a formal parameter it is automatically given the length necessary to hold the actual string value passed to it. When the procedure is finished the part of memory occupied by W$ is set free.

A procedure is headed

```
  PROC WRITERECORD(R,N$,REF M())
```

and is called by

```
  WRITERECORD(STUDENTNO,NAME$,MARKS)
```

In this example R and N$ are formal value parameters, and during the call they are assigned the values of STUDENTNO and NAME$, respectively. The

```
  REF M()
```

denotes a formal parameter M that is called by reference. The () following M indicates that M must refer to a one dimensional array. If the call is to be valid, MARKS must be the name of a one dimensional array. With a reference parameter no assignment take place during the call, but the formal parameter in question is simply used by the procedure as a "nickname" for the actual parameter. So in this case MARKS will actually "suffer" from anything WRITERECORD does to M. The following metaphor might help you to remember what a reference parameter is: A boy named JEREMY is called JIM at home - i.e. locally. If JIM is overfed by his mother the world will see JEREMY grow fat. The procedure WRITERECORD might also be headed

```
PROC WRITERECORD(R,REF N$,REF M())
```

The only difference from the former heading is that N$ is now a parameter to be called by reference. N$ will only refer to NAME$ and no assignment takes place. This of course speeds up the process and saves storage.

A procedure with this heading is given

```
PROC PRINTOUT(REF TABLE(,))
```

The (,) following the name TABLE indicate that TABLE must refer to a two dimensional numerical array. Thus (,,) would indicate reference to a three dimensional array, and so forth.

```
PROC BACKWARDS(REF W$) CLOSED
  LN:=LEN(W$)
  DIM B$ OF LN
  FOR I:=LN TO 1 STEP -1 DO B$:+W$(I)
  W$:=B$
ENDPROC BACKWARDS
//
DIM B$ OF 30
INPUT "WORD (MAX. 30 CHAR.): ": B$
BACKWARDS(B$)
PRINT B$
```

The string B$ declared in the procedure has nothing to do with the string B$ declared in the mainline program, since the procedure is closed. In fact W$ is taking over the part of "outer B$". See also FUNC and CLOSED.

RANDOM

Indicates that a file is opened for random access. See OPEN.

READ

Retrieves data from a data queue set up in DATA statements. Its syntax is

```
READ <variable name> {,<variable name>}
```

As data elements are read a data pointer is moved to point to the next element. When the last element in the queue has been read a built-in Boolean function EOD (End-Of-Data) returns a value of TRUE (see STANDARD FUNCTIONS).

The data pointer may be reset to the beginning of a queue by means of the RESTORE statement (See RESTORE).

```
READ NAME$,TEL
...
DATA "JOHN NELSON",34
```

After the READ statement has been executed, NAME$ is assigned the value "JOHN NELSON" and TEL is set to 34. Note that a string constant must be read by a string variable, and a numeric constant must be read by a numeric variable. The types of the variables in the READ statement must be in accordance with the types of the constants in the queue. See also DATA.

```
NO:=1
REPEAT
  READ NAME$(NO),TEL(NO)
  NO:+1
  PRINT NAME$(NO);
  PRINT "HAS TEL.NO.";TEL(NO)
UNTIL EOD
...
DATA "MAX ANDERSSON",34,"PETER CRAWFORD",45
DATA "ANNI BERSTEIN",12,"LIZA MATZON",56
```

READ FILE

Retrieves data from sequential and random access files stored by using the WRITE FILE statement (see WRITE FILE). Its syntax is

```
 READ FILE <file#> [,<rec#>]: <variable list>
```

where <file#> and <rec#> are both <numeric expression>.

Note that a variable on the <variable list> may refer to an array, and in that case a whole array of data can be retrieved in a single execution of a READ FILE statement

```
  DIM NAME$(100) OF 30
  READ FILE 2: NAME$
```

Values for the whole array NAME$ is retrieved from the sequential file signed on as file number 2.

```
  READ FILE 4,RECNO: NAME$,OWNER$,DEST$,CARGO'NO
```

The statement reads from record no. RECNO in the file opened as no. 4. See also OPEN, WRITE, PRINT, INPUT, and CLOSE.

REF

Marks formal parameters to be called by reference. See PROCEDURES AND PARAMETERS and FUNC.

REM

Initiates comments. The interpreter converts it into the symbol "//". A comment may be placed on a line of its own (like a REM statement in BASIC) or at the end of any other statement, and is initiated with the symbol "//".

```
  IF CH$ IN VOWELS$ THEN //IS IT A VOWEL?
    COUNT'VOWELS:+1
  ELSE //MUST BE A CONSONANT
    COUNT'CONSONANTS:+1
  ENDIF //LETTER
```

RENUM

Used to change or adjust line numbers. Its syntax is

```
RENUM [<line number>] [,<increment>]
```

COMMAND	RESULTS IN LINE NUMBERS
RENUM	10, 20, 30, 40, etc.
RENUM 100	100, 110, 120, 130, etc.
RENUM 150,5	150, 155, 160, 165, etc.
RENUM ,2	10, 12, 14, 16, etc.

REPEAT STRUCTURE

The syntax of the REPEAT loop and the REPEAT and UNTIL statements is given in this diagram

```
REPEAT
  <statement list>
UNTIL <numeric expression>
```

The program section given by <statement list> is executed repetitively until the <numeric expression> returns a value of TRUE (i.e. numeric non-zero).

```
REPEAT
  READ NAME$,TEL
  FOUND:=(THISNAME$=NAME$)
UNTIL FOUND OR EOD
```

RESTORE

Resets the data pointer to the first element in a data queue. Its syntax is

```
RESTORE
```

See also DATA and READ.

RETURN

Returns a value from a function, or returns from a procedure before the ENDPROC statement is reached. Its syntax is

```
RETURN [<numeric expression>]
```

Two examples follow:

```
FUNC MAX(X,Y)
  IF X<=Y THEN
    RETURN Y
  ELSE
    RETURN X
  ENDIF
ENDFUNC MAX

FUNC GCD(A,B)
  IF (A MOD B)=0 THEN
    RETURN B
  ELSE
    RETURN GCD(B,A MOD B)
  ENDIF
ENDFUNC GCD
```

Note that the function GCD is calling itself recursively. See also FUNC and PROCEDURES AND PARAMETERS

RND

A standard function. RND(X,Y), X and Y integers and X less than Y, returns a random integer in the range from X to Y. RND(Y) returns a random real number in the range from 0 to 1. If Y is zero or negative, a new sequence of random numbers is seeded and used, but if Y is positive, the next random number from the previously created sequence is used.

RUN

Invokes a prepass of the program in workspace (unless the program has already been prepassed and no changes have been made in it) and then starts execution of it. See also CHAIN. Its syntax is

```
RUN
```

SAVE

Used to store programs on diskette or tape. Its syntax is

```
SAVE <file name> [,<unit no.>]
```

Programs stored by using SAVE may be retrieved by LOAD or CHAIN.

```
  SAVE "AUNTIE"
```

stores the program presently in workspace on a diskette in unit no. 8.

```
  SAVE "UNCLE",1
```

stores the program presently in workspace on a tape in unit no. 1. See also LOAD, CHAIN, LIST, and ENTER.

SELECT OUTPUT

Directs printout to the screen or the printer. Its syntax is

```
  SELECT [OUTPUT] <device>
```

where <device> is "LP:" (Line Printer) or "DS:" (Data Screen). The default output device is the screen.

```
  PRINT "I AM HERE."
  PRINT "WHERE ARE YOU?"
  SELECT "LP:"
  PRINT "I AM HERE BESIDE YOU."
  SELECT "DS:"
  PRINT "THANKS, PRINTER."
```

The two first texts are displayed on the screen, the third one is sent out on the printer, and the fourth one appears on the screen.

SETEXEC

Chooses whether the interpreter will list the keyword EXEC when listing a program (see EXEC). Its syntax is

```
  SETEXEC <sign>
```

where <sign> is + or -.

```
  SETEXEC+  makes COMAL list the keyword EXEC
  SETEXEC-  causes EXEC to be supressed
```

The default mode is SETEXEC-. If you are in SETEXEC+ mode the keyword EXEC is inserted automatically by the system (you never need to type in EXEC). On the other hand you always are allowed to type in the EXEC. The interpreter will simply ignore it while in SETEXEC- mode.

Note: the reason for having this command is one of compatibility. In earlier version of COMAL the EXEC was compulsory, and some people might still like to have it. See also EXEC.

SETMSG

Suppresses the error messages. Its syntax is

```
SETMSG <sign>
```

where <sign> is + or -. Default mode is SETMSG+.

SETMSG+	Enables the error messages
SETMSG-	Disables the error messages

Error messages are held in a file on the diskette to save main storage. This means that you will have to wait about 3 seconds to get a message on the screen. To a trained programmer this could be annoying. Therefore the option to switch the messages off is given with SETMSG. If in SETMSG- mode a prompt like

```
ERROR 12
```

is displayed with the cursor placed on the estimated location of the error.

SGN

A standard function. SGN(X) returns the sign of X: -1 if X is positive, 0 if X is equal to zero, and 1 if X is positive.

SIN

A standard function. SIN(X) returns the sine of X (X in radians).

SIZE

Prints the size of free memory in bytes. Its syntax is

```
SIZE
```

SQR

A standard function. SQR(X) returns the square root of X (X non-negative).

STANDARD FUNCTIONS.

ABS(X)	returns the absolute value of X.
ATN(X)	returns the arctangent in radians of X.
CHR$(X)	returns the character whose ASCII value is X.
COS(X)	returns the cosine of X (X in radians).
EOD	returns a value of TRUE (numeric 1) if the last element in the data queue has been read, otherwise a value of FALSE (numeric 0) is returned.
EOF(X)	returns a value of TRUE (numeric 1) if the end-of-file mark in a sequential file opened as file number X has been encountered, otherwise a value of FALSE (numeric 0) is returned.
ESC	returns a value of TRUE (numeric 1) if the STOP key is depressed, otherwise it returns a value of FALSE (numeric 0).
EXP(X)	returns the value of e (nat. log. base) to the power of X (thus being the inverse of nat. log.)
KEY$	returns the first ASCII character in the keyboard buffer. If no key has been depressed, a CHR$(0) is returned.
INT(X)	returns the integer part of X, i.e. the greatest integer less than or equal to X.
LEN(X$)	returns the current length, i.e. number of characters, of the string value of X$.
LOG(X)	returns the natural logarithm of X, X positive.
ORD(X$)	returns the ASCII value of the first character held by X$.
PEEK X	returns the contents of memory location X (X in the range 0-65768) in decimal representation.
RND(X,Y)	returns a random integer in the range from X to Y, X and Y integers and X less than Y.
RND(X)	returns a random real in the range from 0 to 1. If X is negative the same sequence is always generated, otherwise a random start is implied.

SGN(X) returns the sign of X: -1 if X is positive, 0 if X is equal to zero, and 1 if X is positive.
SIN(X) returns the sine of X (X in radians).
SQR(X) returns the square root of X (X non-negative).
TAN(X) returns the tangent of X (X in radians).

STATUS

Makes the system display the disk operative system status and switches off the error indicator.

STEP

Indicates an optional counter variable increment in a FOR statement. See FOR.

STOP

Stops program execution. Its syntax is

```
STOP
```

STRING HANDLING, SUBSTRINGS.

A string variable must always be declared. For example

```
DIM NAME$ OF 30
```

declares a string variable NAME$ that may hold up to 30 characters. If a string array is declared, the maximum length of the components must also be specified. For example

```
DIM ADDRESS$(100,3) OF 20
```

declares a two dimensional string array, where each component may hold up to 20 characters.

Formal parameters of type string have no predeclared length. Thus in

```
PROC PACK(N$)
```

the parameter N$ is automatically given the length necessary to hold the string value passed to it.

A substring is specified by giving the position of the first and last character in it. If for example NAME$ has the value: "RICHARD PAWSON", then

```
  NAME$(9:14)
```

returns the string "PAWSON".

If the string SPACES$ is declared (DIM) to a length of 60 characters, the assignment

```
  SPACE$(1:60):=""
```

fills SPACE$ with spaces (CHR$(32)).

In the string NAME$, the expression NAME$(5) is equal to NAME$(5:5), i.e. if the substring is only one character long, you only have to give the position of that character.

Also note that substring assignment is allowed. If the following statements are executed

```
  DIM ADDRESS$ OF 80
  ADDRESS$(1:80):=""
  ADDRESS$(21:40):=HOUSE$
```

the current value of HOUSE$ is stored in ADDRESS$ on positions 21-40. If the value of HOUSE$ has a length of more than 20 characters surplus characters are lost.

If a substring of an array component is to be pointed out, the component is first indicated and after that the substring. If TEL$(23) has a value of

```
  "HARRY HENDERSON     3456"
```

then the string expression

```
  TEL$(23)(21:24)
```

returns the value "3456".

SYS

Invokes a machine code subroutine call (JSR). Its syntax is

```
  SYS <memory location>
```

where <memory location> is a <numeric expression> that must return a value in the range 0-65535.

TAB

In a PRINT statement the TAB function may be used to set the next print position. The argument of the TAB function must be positive and not greater than 32767. If a value greater than 80 (line length) results it is first divided by 80, and the remainder is used. Non-integer values are truncated before use. If the TAB function evaluates to a position prior to the current one, the tabulation is effected on the next line.

```
  PRINT "  MATHEMATICS:",TAB(20),2
```

produces this printout

```
  MATHEMATICS:      2
```

with "2" printed in column 20.

```
  PRINT "  MATHEMATICS:",TAB(5),2
```

produces this printout

```
  MATHEMATICS:
    2
```

The example demonstrates that if the TAB function returns a position prior to the current one, the next line is used. See also PRINT.

TAN

A standard function. TAN(X) returns the tangent of X (X in radians).

THEN

Ends an IF and ELIF statement. See ELIF, IF.

TO

Separates <initial value> from <final value> in a FOR statement. See FOR.

TRAP

Enables or disables the functioning of the STOP key. Its syntax is

```
  TRAP ESC <sign>
```

where <sign> is one of the characters + or -. Default mode is TRAP ESC+.

```
  TRAP ESC-   Disables the STOP key
  TRAP ESC+   Enables the STOP key
```

After the statement or command

```
  TRAP ESC-
```

has been encountered by the interpreter, depressing the STOP key will have no effect on program execution, but the function ESC (see ESC) returns the value TRUE (numeric 1). The command or statement

```
  TRAP ESC+
```

brings the STOP key back to normal mode of operation.

TRUE

A predefined constant with the numeric value 1. See also FALSE.

UNIT

Used in OPEN FILE statements when a certain external device must be indicated. Default unit is always disk unit no. 8. See OPEN.

UNTIL

Terminates the block of statements in a REPEAT-UNTIL loop. See REPEAT.

USING

Formats output of numerical values. See PRINT USING.

WHEN

Initiates a block of statements in the CASE structure. See CASE.

WHILE STRUCTURE

The syntax of the WHILE loop and the statements that control it is

```
  WHILE <numeric expression> [DO]
    <statement list>
  ENDWHILE
```

The block of statements in the <statement list> is executed repetitively as long as - i.e. while - the expression following the WHILE keyword is evaluated to TRUE. When the expression evaluates to FALSE, control is transferred to the statement following the ENDWHILE statement.

```
  TAKEIN("NAME")
  WHILE NOT OK DO
    ERROR("NAME")
    TAKEIN("NAME")
  ENDWHILE
```

If the <statement list> contains only one statement a short form of the WHILE loop may be used. Its syntax is

```
  WHILE <numerical expression> DO <statement>
```

In this case no ENDWHILE statement is needed - nor allowed - to terminate the loop.

```
  WHILE X<A(I) DO I:+1
```

is functionally equivalent to

```
  WHILE X<A(I) DO
    I:+1
  ENDWHILE
```

WRITE FILE

Stores data in a sequential or random access file. Its syntax is

```
  WRITE FILE <file#> [,<rec#>]: <variable list>
```

where <file#> is a <numeric expression> that must return an integer in the range 2-254 (the COMAL System uses numbers 1 and 255), and <rec#>

is a <numeric expression> that must return a positive integer.

Data stored using the WRITE FILE statement may be retrieved with the READ FILE statement but not with the INPUT FILE statement.

```
  WRITE FILE 2: NAME$,ADDRESS$,PAYCODE
```

writes sequentially the values of the variables on the list to file number 2.

```
  WRITE FILE 4,NUM: NAME$,ADDR$,DEPTNO
```

writes the values of the variables on the list to file number 4, in the record given by the value of NUM. Note: WRITE FILE and READ FILE cannot be used with files stored on cassette.

ZONE

Defines the width of the print zones. The value of ZONE may be set with this statement

```
  ZONE <zone width>
```

where <zone width> is a non-negative <numerical expression>. Default value of ZONE is zero.

```
  ZONE 10
  PRINT 1,2,3
  PRINT "----5----0----5----0----5"
```

produces the following output:

```
1         2         3
----5----0----5----0----5
```

```
  ZONE 20
  PRINT "PRICE PER POUND:",PRICE
```

If PRICE has the value 1.5 this results:

```
PRICE PER POUND:    1.5
```

```
  PRINT ZONE
```

displays the present value of ZONE.

WHERE TO FIND MORE INFORMATION

This book was a mini reference book on C64 COMAL version 0.14. For a much more complete and detailed reference you should get the COMAL HANDBOOK, a 470 page reference book on COMAL. A comparable reference book, COMMODORE 64 GRAPHICS WITH COMAL, is expected to be available late 1984.

Reference books are great, but to actually learn COMAL, a beginner should use a tutorial textbook. Several good ones are available for COMAL, including: BEGINNING COMAL, FOUNDATIONS IN COMPUTER STUDIES WITH COMAL, and STRUCTURED PROGRAMMING WITH COMAL. All COMAL books mentioned here are available from the COMAL Users Group, U.S.A., Limited.

Your best source of continuing information on COMAL is the COMAL TODAY newsletter. It is packed with articles, programming tips, news, reviews, and program listings. Contributors include Borge Christensen (author of this book and BEGINNING COMAL), Len Lindsay (author of COMAL HANDBOOK), Colin Thompson (well known columnist), UniComal (authors of C64 COMAL), and many others. The reviews of other COMAL books that follow are condensed from various issues of COMAL TODAY.

REVIEW: FOUNDATIONS IN COMPUTER STUDIES WITH COMAL

This textbook is a good value, 313 pages of solid information. It seems to have been written for APPLE COMAL but since it is COMAL, we really could not tell the difference. There are over 100 sample programs. The page of notes from the COMAL Users Group explains the reason for any differences in the programs. The book starts out as a hands on tutorial, teaching you how to write and save simple COMAL programs. From there, the book goes into the theoretical aspects of structured programming with immediate applications in COMAL. It was Clay that first noted that the book was teaching theory, I thought it was just being interesting. The book final digressed to a facinating study of the variety and history of the computer. Clay felt the highlight of these chapters was the 1946 picture of a 30 ton Computer. What I found even more facinating was the books treatment of multidimensional arrays, random and direct access files, and recursive routines. The author concludes with brief chapters regarding the applications of computers and the social and ethical implications of computer dataprocessing.

Overall I found this book to be most enjoyable and informative not only with specific COMAL applications but also in establishing a good foundation.

Reviewed by David Skinner & Clay Ratliff. Originally from Clark County Commodore Computer COMAL Club Newsletter.

REVIEW: COMAL HANDBOOK

This book is more a manual or reference book than a textbook. It is, however, essential for anyone who wishes to learn COMAL on Commodore computers. The main part of the book is in the form of a reference manual covering all the COMAL keywords. For each keyword there is an explanation, its syntax is given, and there are examples and sample programs. Whether or not each keyword matches the standard is stated, as are the versions in which each keyword is available. There are many appendices, some of which are very useful. There are special sections for the COMAL structures, string handling, and useful procedures and functions.

For someone who already knows C64 BASIC and its operating environment, this book should be extremely helpful to them in learning COMAL. Some users may however require additional assistance in getting to grips with the more complex aspects of COMAL such as procedures and functions with parameters. The book is not particularly suitable on its own for someone learning COMAL as a first language. It is however, an essential reference book for all who use COMAL on Commodore computers.

Reviewed by Diarmuid McCarthy. Originally from Riomhiris na Schol, published by the Computer Education Society of Ireland, Colaiste an Spioraid Naoimh, Bishopstown, Cork, Ireland ($30 per year).

REVIEW#2: COMAL HANDBOOK

This book contains the most complete description of the COMAL language to be found anywhere. And I do mean COMPLETE! The Keywords in the book are in alphabetical order, each on a separate page. This allows ample room for an in-depth discussion of how, when, and where it should be used, including which versions support which keywords. The standard syntax is listed first, with default values and possible ranges for each value in a clear and easy to understand format. Next comes one or more examples of how the keyword looks in a working program, with both user inputs and computer responses shown. Finally, cross references show where to find other examples, procedures at the end of the book which contain this keyword, and a list of related keywords.

Together with this book, you have the most efficient programming language working for you. It's as if the authors of COMAL are standing behind you. The answer to your question has already been answered. Complete is the only word to describe the COMAL HANDBOOK. When you need to know something about COMAL, this is the place to look.

REVIEW: BEGINNING COMAL

As a leading educator in Denmark, Borge Christensen has successfully written a hands-on COMAL tutorial aimed at the beginning computer user. Assuming you have had no previous computer experience, this book will teach you to program in COMAL.

A wonderfully direct technique is used to reveal the power and beauty of COMAL. Chapter 1 begins with this program line: 25 PRINT "HI, THERE." By dissecting this simple line of code, Mr. Christensen introduces line numbers, statements, keywords, and string constants. While still on page 1, the student is already presented with a hands-on example to run. By building on these short simple concepts with a complete series of examples and exercises, a student is led from "print your name" through variables, conditionals, iteratives, and into file structures. As each new concept appears, a clear example of its usage is given, along with exercises to show why it works.

While another book might be preferred as a reference, BEGINNING COMAL should be your text of choice for teaching COMAL.

REVIEW#2: BEGINNING COMAL

This is a programmed instruction course in the classical sense. There blanks for you to write in your own answers throughout the book. There are structure diagrams galore, to represent each principle being considered. This is one book that you cannot use without the computer. While the COMAL Users Group did include an errata sheet with the book, it was really more an instruction sheet for using the accompanying disk; there were very few errors of any kind. The book begins with 'the computer writes a message' and before it ends we have covered data management, accounting, statistics, etc. in a somewhat superficial way. Be assured I do not mean that in a negative way, the text remains light and flows gently from one subject to the next. If I had to recommend a best first book for beginning COMAL, I would have to recommend, BEGINNING COMAL by Borge Christensen.

Review originally from Clark County Commodore Computer COMAL Club Newsletter.

ISBN 0-928411-00-1