Now for the Commodore 64

THE COMAL HANDBOOK

SECOND EDITION

LEN LINDSAY

THE COMAL HANDBOOK
Second Edition

(Now for the Commodore 64®)

LEN LINDSAY

Reston Publishing Company
A Prentice-Hall Company
Reston, Virginia

Library of Congress Cataloging in Publication Data

Lindsay, Len.
COMAL handbook.

Includes index.
1. Commodore 64 (Computer)—Programming.
2. COMAL (Computer program language) I. Title.
QA76.8.C64L56 1984 001.64'24 84-3381
ISBN 0-8359-0784-8

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

This book was written using WORD PRO 4, a word processing system, running on a CBM 8032 computer with a CBM 4040 dual disk drive and a C-Itoh Starwriter printer. The entire text was transmitted electronically via telephone with a CAT modem and the McTerm terminal system to the typesetting system in Reston, Virginia. The final typesetting was done on a TYXSET 1000 system using a Mergenthaler Omnitech/2100. The galleys and page makeup were also done on the TYXSET 1000 system using a Canon LBP-10 Laser Printer for page proofs. Program listings, examples, and syntax descriptions were produced, camera ready on the Starwriter printer.

COMAL-80 is a high level language developed by Borge Christensen in Denmark. It was defined in May of 1982. CBM COMAL-80 is a series of specific implementations of COMAL-80 written by Mogens Kjaer, Lars Laursen, Jens Erik Jensen, and Helge Lassen of UniComal in Denmark.

I dedicate this book
to my beautiful six year old daughter

RHIANON

and to all other future programmers

CONTENTS

PREFACE

When I first saw the announcement of a new language named COMAL, soon to be released by Commodore for their PET®/CBM® computers, I had a feeling that it was the language I had been waiting for. I was not satisfied with BASIC, even though eventually you could get it to do what you wanted. I had tried Pascal but did not like its rigidity and operating environment. FORTH was too much like ASSEMBLER to suit my needs, and PILOT did not have the power I was looking for. I immediately began searching for a copy of COMAL to run on my CBM 8032 and for information about the language. Fortunately, I had a long line of good connections, and I managed to get a copy of COMAL before it was released, becoming one of the first in the USA to be running COMAL. Since information about it was extremely scarce, I decided to compile the much needed information myself. This handbook is the result of over two years of testing and working with COMAL. In May, 1983, the manuscript was taken to Denmark, and reviewed in detail with UniComal, the creators of CBM COMAL-80: Mogens Kjaer, Lars Laursen, Jens Erik Jensen, and Helge Lassen.

I gladly quit using BASIC as soon as I had COMAL. The switch to COMAL was a welcome relief. I only wish I had learned COMAL before I acquired so many bad habits from BASIC. Writing programs with COMAL was a pleasure. The language worked logically, matching the way I put my programs together. COMAL made file handling so easy that I am now using disk files for many more things. In one weekend I wrote an entire order processing system, to handle orders placed with the COMAL USERS GROUP. That speaks for how easy it is to program with COMAL.

This handbook has gone through many revisions, and now is in a form that will be most beneficial for everyone, from beginners to the more advanced. It should help you to learn COMAL, and then become a reference later as your

programs increase in size. Even if you don't yet have a computer, after reading this handbook, you probably won't want one unless it runs COMAL.

The help and support of many people made this handbook possible. I wish to thank them all, for even the little things were very important to me. I would especially like to thank (in alphabetical order): Edgar Anderson, Jim Butterfield, John Collins, Dan Duckart, Tom Foth, Steve Kortendick, John Main, Pam Pierce, RESTON Publishing, and Jim Strasma. And no, I did not forget them; I want to thank Borge Christensen, founder of COMAL, and Mogens Kjaer, Lars Laursen, Jens Erik Jensen, and Helge Lassen, the creators of CBM COMAL-80, for their enthusiastic help, support, and cooperation. And most importantly I wish to acknowledge that both my wife, Maria, and daughter, Rhianon, were very understanding with my addiction to working on making this handbook as perfect as I could. At least they always knew where I was.

Len Lindsay

FOREWORD

by Borge Christensen

"I'm tired of BASIC but scared of PASCAL."
— Anonymous Danish student

Programs for computers are of increasing importance. More and more are being written every day. These programs are meant to be used for something, and are going to *do something* which influences people's welfare in such broad areas as economy, health, and culture. The computer extends our *linguistic potential*; man has always been able to use language to trigger off movements and changes in his environment. But it is new that we are now able to leave the traces of our language in tools which, although far away from the place where they were written, can then translate the message into action. The consequence may be new knowledge, new potentials, new wealth, but also disasters that may live long in our history. That is why *good programming languages* are of vital importance.

The programming language COMAL (COMmon Algorithmic Language) 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 did this by combining 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 all that a set of statements modeled after Pascal that makes it easy — in fact almost unavoidable — to write *well structured programs*. Instead of leading people away from the modern effective way of professional programming, COMAL offers a perfect introduction to this new art.

It is a fact not to be overlooked that programming languages are not only used to control computing machinery, but also for *communication of ideas*. Programs are of course code for computers, but they are also texts meant to be read by other people. This aspect of programming is of growing importance. A program is most likely to be evaluated, adapted, and maintained by persons other than those who wrote it. Most BASIC programs are very hard to read and understand because of their lack of structure and proper naming facilities. The word *encoding* is usually the proper one to apply to BASIC source programs. On the other hand, people who start to look at COMAL programs very often report the feeling that they "have seen this before, because it is so easy to read." I am confident that you will get the same feeling by looking at the many examples in this handbook which it is my pleasure and privilege to foreword.

COMAL-80 is easy to learn and powerful to use. Those who want to go on to even more professional high level languages, such as Pascal or Ada, find it natural to do so. Those who do not have such ambitions simply get their jobs done in a neat and clean way. COMAL-80 should be the first language for those who have never tried before, and the next language for anybody else. After reading this handbook you will know how to use it and why you should try to.

INTRODUCTION

WHAT IS COMAL?

COMAL-80 is a high-level language that originated in Denmark. It is *not* a dialect of the BASIC language, although it has borrowed the best aspects from it. Program execution is line number independent, but line numbers are provided, as in BASIC, to allow easy editing of programs. This handbook is written specifically for CBM COMAL-80, which follows the COMAL KERNAL. It should also be applicable to all other *standard* COMAL-80 implementations, such as RC COMAL-80 version 2.00, except for the enhancements and machine dependent features. See Appendix H for the complete COMAL definition, referred to as the COMAL KERNAL. The KERNAL was adopted in May 1982 and reaffirmed in December 1983.

CBM COMAL-80, COMAL for short, is a run-time compiler that retains the friendly environment of Commodore BASIC® while it adds the power of structured programming, styled after PASCAL. Thus COMAL combines the best of BASIC with the best of PASCAL. Plus, the versions adapted for the Commodore 64® computer include the best of LOGO: turtle graphics. See the book *Commodore 64 Graphics with Comal* for details on these.

Many COMAL commands and statements may be executed mediately as they are entered. Or, precede them with a line number and they will be incorporated into a program. COMAL checks each line as it is entered for correct syntax, and won't accept a line until its syntax is correct. It also provides excellent program entry error messages and indicates the location of the error with the cursor. The error message is printed below the line in error. As soon as the line is corrected, the previously overwritten line is replaced on the screen (the error message is nondestructive). You can immediately RUN a program with the RUN command,

as in BASIC. COMAL will first scan the entire program for structure errors and convert all branching statements to the actual addresses. This prepass takes less than a second, even for large programs. If no program structure errors are found, the program is executed (this is a run-time compiler).

Editing a program is similar to editing a BASIC program, complete with full screen editing. One difference is that to delete a program line BASIC uses a line number with no statement following it, while COMAL uses the more advanced DEL command, allowing any block of lines to be deleted at one time. LOADING and SAVING programs is just as with BASIC, with the added capability of merging sections of programs. Both sequential and direct access files are available, and are compatible with similar files created by BASIC.

There are several different versions of CBM COMAL, each written by UniComal:

Version 0.xx: Introductory version 0.11 is the original COMAL that would run on any 32K CBM® or PET® with 4.0 ROMs. It was replaced by version 0.12 due to changes in the COMAL KERNAL definition. That version was updated by version 0.14 to fix several minor flaws and incorporate a LOGO compatible turtle graphics system in the Commodore 64 version. If you only have obsolete version 0.11 or 0.12 you should get it upgraded to version 0.14, available from Reston Publishing Company or COMAL Users Group, U.S.A., Ltd. Version 0.14 for the Commodore 64 is a special adaptation that includes keywords to control sprites, high res graphics, and a LOGO compatible turtle graphics system, plus the addition of the keywords KEY$ and PRINT USING.

Version 1.xx: Version 1.02 was released for the Commodore 8096 as a full COMAL-80 implementation (except for string functions), plus many enhancements. It has been replaced by version 2.00. If you have this obsolete version, you should get it upgraded to version 2.00, available from COMAL Users Group, U.S.A., Ltd.

Version 2.xx: Version 2.00 replaced version 1.02. It includes the user defined string functions lacking in version 1.02 as well as a more consistent file naming system. External procedures and a superb Error Handler make it an excellent language for serious programming. It requires either 96K RAM (CBM 8096), the 64K COMAL ROM board, COMBI board (COMAL in ROM plus 128K RAM), or Commodore's Commodore 64 COMAL cartridge. The version for the Commodore 64 allows packages to control its special graphics, sprite, and sound features. These special features are detailed in the book *Commodore 64 Graphics with Comal.*

How to get COMAL for your Commodore computer

1. Disk loaded version 0.14 is available from Reston Publishing Company or COMAL Users Group, U.S.A., Ltd. Madison WI.
2. Disk loaded version 2.00 is available from COMAL Users Group, U.S.A., LTD. Madison, WI.
3. COMAL C64 Cartridge is available from Commodore Business Machines at each of their international sales centers.

This handbook is written to apply to the current versions 0.14 and 2.00. Sample programs and procedures were tested with version 0.12 and preliminary releases of versions 0.14 and 2.00.

HOW TO USE THIS HANDBOOK

This handbook is structured primarily to be easy to use as a reference. No book can be everything to everyone, but the many sample programs should also provide a mini-tutorial in programming with COMAL. Plus the example procedures and functions listed in Appendix D should give you a good start as you begin writing your own programs. Finally, the complete COMAL KERNAL is reproduced in Appendix H for those who wish to refer to it. It is used as the model or standard for all COMAL implementations.

Thus the sample programs are always followed by a sample run, showing a typical program execution. If you have COMAL on a Commodore computer, type in the sample programs and run them. By actually *doing*, you will understand COMAL much faster than by just *reading*.

TELL ME, I FORGET
SHOW ME, I REMEMBER
INVOLVE ME, I UNDERSTAND

Ancient Chinese Proverb

NOTE: See Appendix O for directions on how to FORMAT a newdisk (called HEADER in CBM BASIC 4.0).

The COMAL KEYWORDS are presented in alphabetical order, for easy reference (to see them grouped by category refer to Appendix Q). Each KEYWORD has its own page (or more if needed), making it easy to flip through the pages to find a specific KEYWORD. Each KEYWORD is presented in the same manner. Once you understand how the information is presented, you can consistently find exactly what you need on any KEYWORD page.

(2) CATEGORY: XXXXX
(3) KERNAL: [XX] VERS 0.14 [X] VERS 2.00 [X]

(4) EXPLANATION XXXXXXX XXXXXX XXXXX XXXXX XXXXX
X XXXX X XXXX XXXX X X XXXXXX XXXXX XXXXX XXX X X
XX XXXX XXXX XXXX XXXX.

(5) NOTES

XXXXX XXXXX XXXXX XXXXX XXXXX XXXXXXX XX XX XXXXXX
XXXX XXX XXX XXXXX XXX XXX XXX XXXXXXX XXXXX

(6) SYNTAX

XXXXXXX ⟨XXXXXX⟩ XXXXXXX ⟨XXXX⟩
(7) ⟨XXXXXX⟩ XXXXXXXX XXX XX XXXXXX XXXX XXXXXX XXX
⟨XXXX⟩ XXXX XXXXXXX XXXX XXXX XXXXX XXXXXXX

(8) EXAMPLES

XXXXXXX XXXXX XXXXXXXX XXXXX XXXXX XXXXXXX
XXX XXXXXX XXXXX XXXX XXXXXXX

(9) SAMPLE PROGRAM / EXERCISE

XXXX XXXXXX XXXXXX XXXX XXXXX XXXXX XXXXX XX X
XXXX XXXX XXX XXXX XXX XXXXX XXX XXXX XXXXX
XXX XXXXX XXXX XXXX XXX X XXX
XXX XXX XXX
XXXX XXXXXXX X XXXXXX
X XXXXXXX XXX XXX

(10) TYPICAL PROGRAM RUN

XXXXXX XXXX XXXX XXXXXXX
XXX XXXX XXXXX XXXXXXXX XXX XXXXXXXX
XXXXX XXX XXXXX XXXXXX X X XXXXXXX XXXXX

(11) ADDITIONAL SAMPLES SEE: XXXXXXX, XXXXXXX, XXXXXXXX
(12) USED IN PROCEDURES: XXXXX, XXXXXXXX, XXXXXXXXXX
(13) SEE ALSO: XXXXX, XXXXXXXX, XXXXXXXXX, XXXXXXX

The typical KEYWORD page is presented in the following manner (refer to the preceding sample KEYWORD page):

(1) KEYWORD — the KEYWORD being presented on the page is identified at the top corners of the page.

(2) CATEGORY — This tells you how the KEYWORD can be used.

(3) VERSIONS — This indicates whether the keyword is part of the COMAL KERNAL or not, and in which versions it is implemented. The following notation is used:

COMAL Kernal		*Versions*	
YES	part of KERNAL	-	not implemented
NO	an enhancement	+	partially implemented
		*	fully implemented

(a) COMAL KERNAL:
This identifies whether the KEYWORD is part of the COMAL KERNAL as printed in Appendix H or not. If it is part of the COMAL KERNAL, it should be similar in any other standard COMAL implementation.

(b) Version 0.14:
This is the introductory version of COMAL written by UNICOMAL. It is included on the COMAL HANDBOOK DISK available from Reston Publishing Company or a User Group disk from COMAL Users Group, U.S.A., Ltd. Version 0.11 (the original release) and version 0.12 are now obsolete, due to changes in the COMAL definition and other updates. Version 0.14 is disk loaded and provides the entire introductory COMAL system (missing some of the COMAL KERNAL and without the enhancements added in version 2.00) with about 6K free user memory on the PET/CBM and 10K on a Commodore 64 computer. The Commodore 64 adaptation includes special procedures and functions to control its special graphic, and sprite features. These special procedures are defined in detail in the book *COMMODORE 64 GRAPHICS WITH COMAL.*

(c) Version 2.00:
This version replaces version 1.02, adding string functions, packages, external procedures, protected input, and error handling. It is available in the following forms, all written by UNICOMAL and completely compatible:

(1) COMAL ROM board.

(2) COMBI board (COMAL in ROM plus 128K RAM and a real-time clock).

(3) Cartridge for the Commodore 64 including additional procedures to control its graphic, sprite, and sound features. These are defined in detail in the book *COMMODORE 64 GRAPHICS WITH COMAL*.

(4) A disk loaded version for the 8096 available from the COMAL Users Group, U.S.A., Ltd.

(4) EXPLANATION — This section gives a brief summary and explanation of the use of the KEYWORD.

(5) NOTES — Specific notes on using the KEYWORD. Important points may be repeated from the explanation section.

(6) SYNTAX — In order to program in COMAL, you must understand each KEYWORD's SYNTAX and how to use it properly. SYNTAX is described in this handbook by a modified form of Backus-Naur notation, similar to the method Borge Christensen used to write the COMAL KERNAL DEFINITION (see APPENDIX H). The method includes the following conventions:

(a) Items in UPPER CASE or lower case not enclosed in angle brackets ⟨ ⟩ must be typed as shown, unshifted (version 2.00 allows you to shift alphabetic characters if you wish).

(b) Items in lower case and enclosed in angle brackets ⟨⟩are supplied by the user. The angle brackets ⟨ ⟩ are not typed.

(c) Items enclosed in square brackets [ ] are optional. If used, do not type the square brackets [ ].

(d) Items enclosed in braces { } are optional, and may have several occurrences. If used, do not type the braces { }

(e) All punctuation should be typed as shown, including parentheses ( ).

(7) Immediately following the SYNTAX is a further explanation of applicable items enclosed in angle brackets ⟨ ⟩.

(8) EXAMPLES — Specific examples of the KEYWORD illustrate its use.

(9) SAMPLE PROGRAM/SAMPLE EXERCISE — This sample shows the KEYWORD as it is actually used. The SAMPLE PROGRAM is a com-

plete program, suitable for entering on your Commodore COMAL system (many of these sample programs are available on the COMAL HANDBOOK MASTER DISK from Reston Publishing). The programs are meant to demonstrate the KEYWORD on that page. Enter the command NEW before starting a new program. The program is printed as you should type it in. In version 0.14 note that an upper case letter means enter that letter *unshifted* (version 2.00 allows shifted letters as an option, not required). Most listings do not include line numbers, since the line numbers do not apply to program execution. Simply enter the command AUTO, and the system will supply line numbers for you. You then can type in the lines as shown. COMAL will convert the "=" to ":=" for an assignment for you. The same is true for optional KEYWORDS (such as DO, THEN, FILE, OUTPUT). The COMAL system will supply them for you. The sample programs are shown with the correct indentation to emphasize the structures. When typing in the program, you don't need to include any leading spaces. Comments about specific program lines (listed to the right of the line in lower case) also should not be typed in. A standard listing convention is used to show special keys in the listing:

LISTING	*KEY TO TYPE*
[CLR]	CLR (clear screen)
[HOME]	HOME (home cursor)
[RVS]	RVS (reverse on)
[OFF]	OFF (reverse off)
[UP]	(cursor up)
[DOWN]	(cursor down)
[RIGHT]	(cursor right)
[LEFT]	(cursor left)

To enter a sample program, remember to do four things:

(a) type NEW, and hit the RETURN key

(b) type AUTO, and hit the RETURN key

(c) type the program lines, each followed by the RETURN key

(d) after the last line is typed in, hit the RETURN key an extra time to terminate AUTO mode (version 0.14) or hit the STOP key (version 2.00).

(10) TYPICAL PROGRAM RUN — This shows a typical execution of the sample program. ITEMS typed in by the user are *underlined* to differentiate them from things printed by the program. Comments about program execution are enclosed in parentheses ( ). This allows an understanding of how the sample program works without actually having to RUN it yourself.

(11) ADDITIONAL SAMPLES SEE — To find more sample programs that include the KEYWORD presented on this page, refer to the KEYWORDS listed here.

(12) USED IN PROCEDURES — The KEYWORD on this page is used in the sample procedures listed here. The procedure listings can be found in Appendix D. A few functions are also included.

(13) SEE ALSO — Other related COMAL KEYWORDS are listed here. Reading about them may give you further insight into the KEYWORD on this page.

EXPLANATION OF SOME COMMON ⟨items⟩ USED IN THIS HANDBOOK

Some common items enclosed in ⟨ ⟩ will frequently be found in the SYNTAX sections of this handbook. These include:

⟨variable name⟩ or ⟨identifier⟩

⟨numeric expression⟩

⟨string expression⟩

⟨statements⟩

⟨filename⟩

⟨filenum⟩

⟨variable name⟩ or ⟨identifier⟩

An ⟨identifier⟩ or ⟨variable name⟩ may be up to 78 characters long. Each character is significant. These characters may be unshifted alphabetic (version 2.00 also allows shifted alphabetic), numeric, apostrophe ('), backslash (\), left square bracket ([), right square bracket (]), and left arrow (←) (changed to underline in listing to an ASCII printer). The first character must be alphabetic. The ⟨ identifier⟩ can be used in several ways. The chart below shows some ways it may be used [replace the (...) with valid array index information]:

ITEM	*REPRESENTED BY*	*EXAMPLE*
KEYWORD	⟨identifier⟩	ENDWHILE
KEYWORD	⟨identifier⟩$	KEY$
real variable name	⟨identifier⟩	COUNT
string variable name	⟨identifier⟩$	NAME$

integer variable name	⟨identifier⟩#	SCORE#
label	⟨identifier⟩:	MONTH:
function name	⟨identifier⟩	EVEN
integer function name	⟨identifier⟩#	GDC#
string function name *	⟨identifier⟩$	UPPER$
procedure name	⟨identifier⟩	SORT
array element	⟨identifier⟩(...)	SCORE(X)
string array element	⟨identifier⟩$(...)	PLAYER$(Y)
integer array element	⟨identifier⟩#(...)	TRACK#(X,Y)

* version 2.00 only

A specific ⟨identifier⟩ may be used in only *one* of the above ways within any one program, other than LOCALly as a parameter of a procedure or function or within a CLOSED procedure or function. For example, if you have a procedure called "TEST" you may not also have a variable named *TEST, TEST#*, or *TEST$*. In addition some ⟨identifiers⟩are reserved for use as a COMAL KEYWORD, and thus may not be used for any other item name. Version 2.00, however, allows you to use COMMAND names (such as SCAN and SIZE) as ⟨identifiers⟩ in a program, as long as the COMMAND is not also a statement keyword. This is so that all ⟨identifiers⟩ in a COMAL program may be independent of the COMMAND language. Thus if you have a version 0.14 procedure named SCAN, it will be allowed in version 2.00, even though SCAN has been added as a COMMAND. However, to execute the procedure from direct mode, it then is required that the word EXEC be used to avoid any ambiguity.

IMPORTANT NOTE: All ⟨identifiers⟩ take up the same amount of program memory, no matter how many characters are used in its name. The complete name is only stored in the name table. Thus using long variable and procedure names does *not* take up any more program memory. The name table may be a bit bigger, but that is a very minor loss. Also, the use of long variable names does *not* slow the program down.

⟨numeric expression⟩

A numeric expression is anything that will evaluate to a numeric equivalent, including the following or combinations of the following:

CATEGORY		*EXAMPLE*
real constant		146.4
integer constant		25
hexadecimal constant	*	$FF
binary constant	*	%10011001

real variable	COUNT
integer variable	WINDOWS#
real array element	SCORE(4)
integer array element	WALLS#(X)
system function	ABS(X)
user defined real function	EVEN(X)
user defined integer function	GCD#(X,Y)
math operation	5 + 34
IN operation	A$IN VALID$
system constant	FALSE
boolean expression	NOT A=B

* Version 2.00 only — see Appendix M for more information.

Examples of combinations are:

```
MONEY - 2.5
SIN(ABS(SCORES DIV 2)) * 5
10 - RESPONSE(CHOICE$  IN VALID$ )
```

⟨string expression⟩

A string expression is anything that will evaluate to a string including the following:

CATEGORY	*EXAMPLE*
string constant	"YOU ARE A WINNER" must be enclosed in quotes
string variable	NAME$
string array element	PLAYER$(2)
substring	ITEM$(3:5)
string array element substring	ARRAY$(3)(6:8)
string operation	DRIVE$+NAME$
string function	CHR$(X)
user defined string function	* SAMPLE$(X,Y)

* Version 2.00 only.

⟨statements⟩

This means that other COMAL statements may go here.

⟨filename⟩

A complete discussion of filenames and their many uses in CBM COMAL is covered in Appendix N. Please refer to that appendix for this important information.

⟨filenum⟩

COMAL supports the same file system as Commodore BASIC. This means that when a file is opened it is given a number that is referenced in statements accessing that file. File numbers can range from 1–255. However, the COMAL system reserves two file numbers for its own use. All versions use file number 255 for system printer access. Version 2.00 additionally uses file number 254 for system disk access, while version 0.14 uses file number 1. Thus a COMAL program can safely use file numbers 2–253 with all versions.

CBM 8000 SERIES SPECIAL EDITING FUNCTIONS

COMAL fully supports the special editing functions available on the CBM 8032 and CBM 8096. These functions include:

CHR$(7) : ring the bell in the computer
CHR$(14) : enter lower case mode
CHR$(15) : set cursor position as upper left corner of window
CHR$(21) : delete the cursor line
CHR$(22) : erase cursor line from cursor position to line's end
CHR$(25) : scroll screen up
CHR$(142) : enter graphic mode (UPPER case and graphics)
CHR$(143) : set cursor position as bottom right corner of window
CHR$(149) : insert a line at current cursor line
CHR$(150) : erase cursor line up to cursor position
CHR$(153) : scroll screen down

Also, the colon ":" will halt any scrolling (useful to pause the display during a program list) and the "9" will resume (the "9" may be hit while the ":" is still depressed). To insert a blank line at the cursor, position press these four keys simultaneously: ESC RVS SHIFT (left side key) "K". To delete the line at the cursor position press these three keys simultaneously: ESC RVS "K".

IMPORTANT NOTE ABOUT THE COMAL NAME TABLE

COMAL creates a "name table" of all ⟨identifiers⟩ used in a COMAL program. Once an ⟨identifier⟩ is placed into the name table it is never removed. Thus, while writing a program, you may change your variable names several times. All of the names you use are retained in the name table, even though some names are never used in your final program. The name table is stored with the program when you SAVE it. However, it is not stored with the program if you LIST it to tape

or disk. Thus to "clean up" your name table, simply LIST your final program to tape or disk, issue a NEW command to clear the program and name table, and then ENTER the program. When the program is ENTERed, the name table is recreated and will only include the identifiers currently used. Since the name table takes up memory space, you should notice that the size of your workspace is increased after you clean up the name table in this manner. In version 2.00 the number of variable names allowed per program is virtually unlimited, while in version 0.14 only 255 names are allowed, sufficient for even long programs.

A benefit of having all variable, array, procedure, and function names in a name table is that program execution is fast, and long names can be used without any loss of program memory space. The program size will be the same regardless of how long the variable names are. Only the name table size will vary. Since the name table uses some of the memory space, only a minimal loss of memory will result from long names. Also, using long identifiers does not slow down the program.

SPECIAL NOTE — NEXT CONVERTS TO ENDFOR

All versions of CBM COMAL accept the keyword NEXT as the terminator to the FOR loop, since it currently is specified in the COMAL KERNAL. However, they convert it to ENDFOR, to be more compatible with the other loop structure terminators. When listed it will be shown as ENDFOR. Please note that the KERNAL may soon be updated to specify ENDFOR rather than NEXT, but until that time, other COMAL implementations all use NEXT as the FOR loop terminator, while CBM COMAL uses ENDFOR. In version 2.00 you can issue the following POKE and COMAL will list NEXT instead of ENDFOR: POKE $24B,PEEK($24b) BITOR %00000100. See the keyword POKE for more details on this setup, as well as a program to automate it.

GETTING STARTED — YOUR FIRST TIME WITH COMAL

If you are just starting to learn COMAL, it may be advisable to obtain a COMAL tutorial that takes you step by step through the many COMAL features. Some of the available books are listed in Appendix I. It also is recommended to get at least one disk of example programs from some source such as the COMAL Users Group.

This book can also be used as a minitutorial. Each sample program listed in the following pages is a complete, ready-to-run program. Nothing extra is needed for most of the programs. The exception is the programs requiring a disk file access.

To get the feel for COMAL, simply type in some of the programs, SAVE them on tape or disk for future use, and RUN them. Then modify them, change some of the lines, add a new line, delete a line, see what happens. This "playing" with the programs is an important step in learning.

COMAL uses PRINT statements to give the computer the ability to relay information to you. This information can be shown on your screen or on a printer.

To see how this works, type in the following line (don't use the SHIFT key, and remember to hit the RETURN key after every line you type):

```
PRINT "HELLO THERE"
```

The computer printed HELLO THERE right below your line, following your instructions. The line was executed in direct mode. It was not included as part of a program. To specify that a line is to be part of a program, simply include a line number from 1–9999 in front of it. For example:

```
10 PRINT "HELLO THERE"
```

When you enter this line, the computer does NOT print HELLO THERE in response. It appears to do nothing. But, in actuality, it has taken the line you entered and stored it for future use as part of a program. To see that it remembered the line, tell it to LIST the current program. The command to do this is LIST (remember to hit the RETURN key after the command):

```
LIST
0010 PRINT "HELLO THERE"
```

Notice that the things you type in are underlined here. The computer's response is not. To make the computer follow your instructions stored in the program, you must give it the RUN command:

```
RUN
HELLO THERE
```

It also is easy to have the computer ask you questions. This is accomplished in several ways, the easiest being the INPUT statement. Try the following short program:

```
10 INPUT "HOW OLD ARE YOU? ": AGE
20 PRINT AGE
```

In this program, the computer asks you a question, and stores your answer into the variable called AGE. Then, you can print the value of AGE any time after that in the program. Try the program:

```
RUN
HOW OLD ARE YOU? 15
15
```

To store your newly created program on disk, put a formatted disk (see Appendix O for the disk format command) into drive 0 and enter the following command:

```
SAVE "0:MYPROGRAM"          normal command - stored in binary
      or
LIST "0:MYPROGRAM.L"        stored in ASCII - use to transfer
                            a program from one version to another
```

Your program is now stored on the disk in drive 0 by the name MYPROGRAM (or MYPROGRAM.L for the ASCII version). For more information on saving programs, see the keyword pages SAVE and LIST. To get your program back from disk, use the following command:

```
LOAD "0:MYPROGRAM"          recall a program stored via SAVE
      or
NEW                         erase the current program
ENTER "0:MYPROGRAM.L"       recall a program stored via LIST
```

To erase the current program from the computer's memory use the command NEW. To have the computer provide automatic line numbers use the command AUTO.

Now you are ready to try some programs that are a few lines longer. Remember, don't use the SHIFT key when typing any letters. They are listed in both UPPER and lower case letters only to emphasize program readability. Also, COMAL will only allow you to type up to 80 characters in any one program line. If you have only a 40-column screen, it will allow you to continue a program line onto the next screen line (two 40-character lines provide you with one 80-character program line). Try the programs listed on the following keyword pages:

- CASE — this program shows you how to use a REPEAT loop, and a multiple choice structure.
- DATA — this program illustrates how information can be stored in the program that it can "automatically" read. If you are using version 0.14 don't include the line RESTORE NAMES.
- DIV — this program shows you how the computer can divide and tells you both the answer and the remainder.
- ELIF — this program has several comparisons that you may find interesting.

- ENDFOR — this program shows how you can use several loops at one time.
- ENDIF — this is a small number guessing game.
- ENDWHILE — this is a small addition drill.
- EXEC — this program shows you how to use modules, often called procedures.
- IN — this is a nice little program to show how COMAL can search through words looking for a specific letter.

HAPPY READING

The rest of the book can be read in any order, no need to read it sequentially. Improper use of COMAL may lead to errors. Since there are innumerable types of errors, many are not detailed in this book. See Appendix J for a few hints at trouble-shooting program errors.

The SETEXEC command gives you the choice of whether you want the word EXEC listed in program listings or not. The system default is to NOT list EXEC. The SETMSG command allows you to turn off the error messages in version 0.14, which may be useful since the messages are on disk, which takes extra time to retrieve. The system default is messages on. Other setup choices are covered under keyword POKE, including UPPER/lower case, quote mode, and control codes. STOP key disable/enable is covered on the ESC keyword page.

NOTES

(1) Version 2.00 does not support SETMSG since its messages come from memory rather than disk.

(2) See keywords POKE and ESC for more information on COMAL system setup.

(3) These commands cannot be part of a program. They may only be used as direct commands.

SYNTAX

```
SETEXEC<type>

SETMSG<type>

  <type> is either + or -
    + means on
    - means off
```

EXAMPLES

```
SETEXEC-
SETMSG-
```

SEE ALSO: POKE, TRAP

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows comments to be included in a program listing. Standard COMAL uses // to represent the beginning of a remark. In addition, CBM COMAL will convert an exclamation point (!) to //. Anything on a line after the // will be ignored and execution continued on the next line. COMAL places one space before the //. If you wish to leave more space than that before your remark, place the extra spaces after the // where they won't be affected by the COMAL interpreter. All remarks are retained when a program is SAVEd, LISTed, or DISPLAYed to disk or tape. They are non-executable and take up memory but don't slow down program execution. They are useful to relate the reasoning behind a program section or to identify sections of a program.

NOTES

(1) Remarks do NOT slow down a program.
(2) An exclamation point (!) is converted to // by the system.
(3) Version 2.00 allows an empty line.

SYNTAX

```
//[<anything>]

 <anything> is optional and may be any printable characters
```

EXAMPLES

```
//
// CLEAR THE SCREEN
// GET THE ANSWER
```

SAMPLE PROGRAM

```
// PRINT AND/OR TABLE
//
DIM type$(FALSE:TRUE) OF 5 // ARRAY FOR WORDS TRUE / FALSE
type$(FALSE):="FALSE";type$(TRUE):="TRUE" // ASSIGN VALUES
ZONE 6 //                                    SET UP THE ZONE
PRINT "AND CHART / OR CHART" //          TITLE OF THE CHART
PRINT "----------------------"
FOR a:=FALSE TO TRUE DO //DO TWO POSSIBILITIES, TRUE/FALSE
  FOR b:=FALSE TO TRUE DO //DO 2 POSSIBILITIES, TRUE/FALSE
    PRINT "A=";type$(a),"B=";type$(b), //VALUE OF A & OF B
    //            NEXT WE PRINT THE VALUES OF THE RESULTS
    PRINT "A AND B=";type$(a AND b),"A OR B=";type$(a OR b)
  ENDFOR b //                                  DO THE NEXT B
ENDFOR a //                                    DO THE NEXT A
ZONE 0 //                          RESET ZONE BACK TO DEFAULT

RUN
AND CHART / OR CHART
----------------------
A= FALSE    B= FALSE     A AND B= FALSE     A OR B= FALSE
A= FALSE    B= TRUE      A AND B= FALSE     A OR B= TRUE
A= TRUE     B= FALSE     A AND B= FALSE     A OR B= TRUE
A= TRUE     B= TRUE      A AND B= TRUE      A OR B= TRUE
```

SEE ALSO: LABEL

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the absolute value of the specified ⟨numeric expression⟩. The absolute value is the value of the numeric expression without a preceding '+' or '-' sign. A negative number is converted to its positive equivalent. A positive number will appear unchanged, since it is already positive. 0 will also be unchanged.

SYNTAX

```
ABS(<numeric expression>)
```

EXAMPLES

```
ABS(-3.2)
ABS(15-num)
ABS(INT(reply))
ABS(high - score)
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "NUMBER (0 TO STOP):" ; num
  IF ABS(num)<>num THEN PRINT "THAT IS NEGATIVE"
  absnum:=ABS(num)
  PRINT "ABSOLUTE VALUE OF";num;"IS";absnum
UNTIL num=0
PRINT "ALL DONE"

RUN
NUMBER (0 TO STOP):58.97
ABSOLUTE VALUE OF 58.97 IS 58.97
NUMBER (0 TO STOP):-98
THAT IS NEGATIVE
ABSOLUTE VALUE OF -98 IS 98
NUMBER (0 TO STOP):-546.3
THAT IS NEGATIVE
ABSOLUTE VALUE OF -546.3 IS 546.3
NUMBER (0 TO STOP):0.1542
ABSOLUTE VALUE OF .1542 IS .1542
NUMBER (0 TO STOP):0
ABSOLUTE VALUE OF 0 IS 0
ALL DONE
```

USED IN FUNCTION: ROUND
SEE ALSO: INT, ORD, SGN, VAL

CATEGORY: OPERATOR
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Logical math operator evaluates to TRUE (a value of 1) only if the ⟨condition⟩ on its left *and* the ⟨condition⟩ on its right are TRUE (values not equal to 0). Otherwise it evaluates to FALSE (a value of 0). The second sample program listed below produces a chart showing each of the four possible combinations.

Version 2.00 has also included an extended operator: AND THEN. If the first expression is FALSE, then the second expression is not evaluated since the result of the operation will be FALSE regardless of whether or not the second expression is FALSE. This prevents evaluation errors in the second ⟨conditon⟩ by allowing the first ⟨condition⟩ to be a test for valid values. For example, you may use an integer variable as a pointer into an array set up via DIM table$(1:max),index(1:max). In version 0.14 you might code:

```
WHILE ptr#>0 AND symbol$<>table$(ptr#) DO ptr#:=index(ptr#)
```

This code could result in an error when the pointer variable is 0, because both conditions are evaluated, and 0 may not be a valid string array subscript. Therefore, the following code could be used to prevent this problem:

```
found:=FALSE
WHILE ptr#>0 AND NOT found DO
  found:=(symbol$=table$(ptr#))
  IF NOT found THEN ptr#:=index(ptr#)
ENDWHILE
```

Version 2.00 allows the condition susceptible to evaluation error to be the second condition, and only evaluated upon successful evaluation of the first:

```
WHILE ptr#>0 AND THEN symbol$<>table$(ptr#) DO ptr#:=index(ptr#)
```

In addition, AND THEN may execute faster than the simple AND.

NOTES

(1) AND is not a bitwise operator as it is in Commodore BASIC. See BITAND for the bitwise operation.
(2) AND THEN is not supported by version 0.14.

SYNTAX

```
<condition> AND <condition>
   or
<condition> AND THEN <condition>

 <condition> is a <numeric expression>
```

EXAMPLES

```
choice$>="A" AND choice$<="Z"
num1=8 AND num2=0
WHILE num>0 AND THEN name$(num:num)=" " DO num:-1
```

SAMPLE PROGRAM

```
DIM word1$ OF 20, word2$ OF 20, guess$ OF 1
word1$:="TWELVE";word2$:="COMAL"     use any words you wish
REPEAT
  PRINT "WHAT LETTER APPEARS IN BOTH"
  PRINT word1$;"AND";word2$,
  REPEAT
    INPUT ": " : guess$
  UNTIL guess$>""                  don't allow null answer
UNTIL (guess$ IN word1$) AND (guess$ IN word2$)
PRINT "YES,";guess$;"IS IN BOTH";word1$;"AND";word2$

RUN
WHAT LETTER APPEARS IN BOTH
TWELVE AND COMAL: T
WHAT LETTER APPEARS IN BOTH
TWELVE AND COMAL: W
WHAT LETTER APPEARS IN BOTH
TWELVE AND COMAL: L
YES, L IS IN BOTH TWELVE AND COMAL
```

SAMPLE PROGRAM

```
DIM type$(0:1) OF 5
type$(0):="FALSE";type$(1):="TRUE"
ZONE 6
PRINT "AND CHART"
PRINT "---------"
FOR a:=FALSE TO TRUE DO
  FOR b:=FALSE TO TRUE DO
    PRINT "A=";type$(a),"B=";type$(b),
    PRINT "A AND B=";type$(a AND b)
  ENDFOR b
ENDFOR a
ZONE 0                          reset ZONE back to default

RUN
AND CHART
---------
A= FALSE     B= FALSE     A AND B= FALSE
A= FALSE     B= TRUE      A AND B= FALSE
A= TRUE      B= FALSE     A AND B= FALSE
A= TRUE      B= TRUE      A AND B= TRUE
```

ADDITIONAL SAMPLES SEE: //, DIM (numeric arrays), EXTERNAL
USED IN PROCEDURES: LOWER'TO'UPPER
SEE ALSO: NOT, OR, THEN

CATEGORY: Type of OPEN
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Specifies that the file being OPENed already exists on disk. New data is written after the existing data in the file. In other words, data is appended or added to the end of the existing data. APPEND can only be used with sequential disk files. For more information about sequential files see Appendix C. The word FILE is optional and if omitted will be supplied by the system.

NOTE

See Appendix N for information on how to include the UNIT and SECONDARY ADDRESS information in the filename.

SYNTAX

```
OPEN [FILE] <filenum>,<filename>,APPEND

 <filename> may include UNIT and SECONDARY ADDRESS info
```

EXAMPLES

```
OPEN FILE 2,"TEST",APPEND
OPEN FILE 5+section,file'name$,APPEND
OPEN FILE 5,"1:NAMES.DAT",APPEND
```

SAMPLE PROGRAM

```
DIM name$ OF 20, add'file$ OF 20
add'file$:="0:VISITOR'INPUT"        name of file
OPEN FILE 2,add'file$,APPEND
REPEAT
  INPUT "VISITOR NAME (* TO END):" : name$
  IF name$<>"*" THEN WRITE FILE 2 : name$
UNTIL name$="*"
CLOSE FILE 2
PRINT "ALL DONE"

RUN
  (system opens previously existing disk file referred
   to as file 2 named VISITOR'INPUT. Positions to its end)
VISITOR NAME (* TO END):JOHN B. GOODE
  (system writes JOHN B. GOODE to disk file 2)
VISITOR NAME (* TO END):HAMILTON HALL
  (system writes HAMILTON HALL to disk file 2)
VISITOR NAME (* TO END):*
  (system closes disk file 2)
ALL DONE
```

SEE ALSO: CLOSE, FILE, OPEN, RANDOM, READ, WRITE

CATEGORY: Special
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Allows you to begin printing at any spot on the screen by specifying the row and column at which to start. This combines a CURSOR statement with a normal PRINT statement. PRINT AT can be combined with PRINT USING, yielding great flexibility. See PRINT for further explanation of the print list.

This capability is extended to the INPUT statement, allowing you to position your input prompt, if any, to any location on the screen. It also allows you to specify the maximum length (number of characters) permitted in the answer.

Using AT is similar to finding your seat at a movie theater. First you find the correct row, then the correct position in the row. This is similar to the ⟨row⟩,⟨col⟩ reference in matrix operations, but differs from ⟨x⟩,⟨y⟩ coordinates.

NOTE

PRINT AT and INPUT AT are not supported by version 0.14.

SYNTAX

```
PRINT AT <row>,<col>: [<print list>][<continue mark>]
   or
INPUT AT <row>,<col>[,<length>]: [<prompt>: ]<var name>

  <row> is a <numeric expression> whose value is from 0-25
    (0 means use the current row)
  <col> is a <numeric expression> whose value is from 0-80
    (only 0-40 on 40 column screens)
    (0 means use the current column)
  <length> is a <numeric expression> whose value is from
1-80
    (only 1-40 on 40 column screens)
    if omitted, the rest of the line is used
  <print list> can include one or more of the following
    separated by a comma or semicolon:
    TAB(<position>)
      <position> is a positive <numeric expression>
    <string expression>
    <numeric expression>
    USING <string expression>: <variable name list>
      <string expression> may be a constant or variable
        used to set up USING image with following reserved
          #  reserves a digit place
          .  the location of the decimal point
          -  floating minus sign (optional)
      <variable name list> is a list of the variables to use
        in filling the USING image.
  <continue mark> may be a comma (,) or a semicolon (;)
    if omitted, a carriage return and line feed result
  <prompt> is a <string expression>
```

EXAMPLES

```
PRINT AT 12,14: "*"
PRINT AT 3,10: USING "$####.##": amount
INPUT AT 3,10: amount
```

SAMPLE PROGRAM

```
REPEAT
  PAGE
  PRINT AT 24,1: "ENTER A ROW (1-23 OR 0 TO STOP):";
  INPUT "": row;
  IF row THEN
    INPUT "ENTER A COLUMN: " : col
    PRINT AT row,col: "*"
    FOR temp:=1 TO 500 DO NULL
  ENDIF
UNTIL row=0
PRINT AT 25,1: "ALL DONE"

RUN
  (screen clears and the following is printed on line 24:)
ENTER A ROW (1-23 OR 0 TO STOP): 2 ENTER A COLUMN: 4
  (a * is printed at row 2 column 4)
ENTER A ROW (1-23 OR 0 TO STOP): 0
  (the following is printed on line 25:)
ALL DONE
```

SAMPLE PROGRAM

```
//Using protected INPUT AT without specifying row / col
PAGE
INPUT AT 0,0,2: "Enter the current hour: " : hour$;
PRINT hour$;"O'Clock"
INPUT AT 0,0,9: "Social Security Number (digits only): ":
ssn$;
PRINT ssn$(1:3),"-",ssn$(4:5),"-",ssn$(6:9)
INPUT AT 0,0,1: "Yes or No (Y/N): " : reply$;
CASE reply$ OF
WHEN "Y","y"
  PRINT "YES"
WHEN "N","n"
  PRINT "NO"
OTHERWISE
  PRINT "ABEND"
ENDCASE
END "All done."
```

ADDITIONAL SAMPLES SEE: EXTERNAL
SEE ALSO: CURSOR, INPUT, PRINT, TAB, USING, ZONE

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the arctangent of the specified number in radians. One radian equals approximately 57°. ATN is the inverse of TAN (tangent). The following formulae may be used for radian/degree conversion:

radians = degrees * (π/180)	degrees = radians * (180/π)
radians = degrees * .0174532925	degrees = radians * 57.2957795

SYNTAX

```
ATN(<numeric expression>)
```

EXAMPLES

```
ATN(5)
ATN(num1+num2-2)
ATN(10-INT(num))
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "NUMBER (0 TO END): " : num
  IF num THEN PRINT "ARCTANGENT OF";num;"IS";ATN(num)
UNTIL num=0
PRINT "ALL DONE"

RUN
NUMBER (0 TO END): 5
ARCTANGENT OF 5 IS 1.37340077
NUMBER (0 TO END): 25
ARCTANGENT OF 25 IS 1.53081764
NUMBER (0 TO END): 0
ALL DONE
```

SEE ALSO: COS, SIN, TAN

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Provides automatic line numbering while entering a program. Valid COMAL line numbers are from 1 through 9999. AUTO is a keyboard command and can't be used as a statement within a program. AUTO will generate a new line number after each carriage return, placing the cursor in column 6, ready for entering the next program statement. Each line number will always be four digits, thus line 30 would appear as 0030. To terminate AUTO mode, hit the stop key in version 2.00, or hit ⟨return⟩ twice in version 0.14. In version 0.14 AUTO will begin with line number 10, unless you specify another starting line, and will increment each line by 10, unless you specify a different interval. In version 2.00 it begins with the next available line number (incrementing by the current step value), unless you specify another, and then increments by 10 unless you specify a different interval.

Version 2.00 also allows you go back to a previous line while in auto mode and correct previously entered lines. It will retain a correct new line number, by using the line number just entered as the point to increment from. Version 0.14 does not have this advanced feature and keeps incrementing the line prompted, even if you went back to a previous line.

NOTES

(1) In version 2.00, hitting ⟨RETURN⟩ with no other entry in response to the line number prompt will create a blank line, which will be listed in the program as just a line number.
(2) In version 0.14, AUTO will prompt a line number without regard to whether it already exists or not. It does not warn you of an existing line of the same number being prompted. Version 2.00 will print the line number in reverse field if a line already exists with the same line number. You then can hit the STOP key to leave that line unchanged, or you may proceed to enter a new line to replace it.
(3) A line entered in AUTO mode replaces any previously existing line of the same number.

SYNTAX

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

 <start line number> is a number from 1 - 9999;
   if omitted, starting line number will be 10 in vers 0.14
   the starting line number will be next available line in
   version 2.00 (incremented by the <increment value>)
 <increment value> is a number from 1 - 9999;
   if omitted, the increment value will be 10
```

EXAMPLES

COMMAND		RESULTS
AUTO	*	Gives line number series: 10, 20, 30,...
AUTO 200		Gives line number series: 200, 210, 220,...
AUTO ,5		Version 0.14 line numbers: 10, 15, 20,...
	**	Version 2.00 line numbers: 5, 10, 15, ...
AUTO 400,2		Gives line number series: 400, 402, 404,...

* If a program already exists, version 2.00 will begin with the next line number (10 past the current last line).

** If a program already exists, version 2.00 will begin with the next line number (5 past the current last line).

SEE ALSO: DEL, DISPLAY, EDIT, LIST, RENUM

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

Returns to BASIC mode of operation. BASIC can only be used as a direct command and cannot be part of a program. This command will reset the Commodore computer with a cold start, returning you to the original BASIC operating system.

SYNTAX

```
BASIC
```

EXAMPLE

```
BASIC
```

SAMPLE EXERCISE

```
BASIC
```

Screen clears — computer is reset.

```
### commodore basic 4.0 ###

31743 bytes free
```

This display will be the same as the one presented when you turn on the computer in BASIC mode.

SEE ALSO: NEW

CATEGORY: Operator
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the bitwise AND of the two numbers. BITAND performs the bitwise AND operation bit by bit on the two numbers. BITAND follows the following rules:

BITAND	00	01	10	11
00	00	00	00	00
01	00	01	00	01
10	00	00	10	10
11	00	01	10	11

Using the above chart, you see that %01 BITAND %11 has the result of %01. COMAL allows binary constant numbers if preceded by the %. While the above chart covers a two-digit BITAND operation, COMAL can handle up to 16 digits with BITAND.

NOTES

(1) The arguments for the BITAND operator must be in the range of 0-65535, decimal or $00-$ffff, hex.
(2) See Appendix M for more information on binary numbers.

SYNTAX

```
<argument> BITAND <argument>

  <argument> is a non-negative number in the range 0-65535.
```

EXAMPLES

```
RETURN PEEK($24B) BITAND %00000100
result:=number1 BITAND number2
```

SAMPLE PROGRAM

```
REPEAT
  Input "Decimal number (0 to STOP): " : number;
  PRINT binary$(number)
UNTIL number=0
END "All done."

FUNC binary$(num) CLOSED
  DIM res$ OF 8
  res$:="00000000"
  bit:=1 // init
  FOR temp:=8 TO 1 STEP -1 DO
    IF num BITAND bit THEN res$(temp):="1"
    bit:+bit // counts 1 2 4 8 16 32 64 128
  ENDFOR temp
  RETURN res$
ENDFUNC binary$

RUN
Decimal number (0 to STOP): 5 00000101
Decimal number (0 to STOP): 65 01000001
Decimal number (0 to STOP): 0 00000000
All done.
```

ADDITIONAL SAMPLES SEE: BITOR, BITXOR, POKE
SEE ALSO: BITOR, BITXOR, PEEK, POKE

CATEGORY: Operator
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the bitwise OR of the two numbers. BITOR performs the bitwise OR operation bit by bit on the two numbers. BITOR follows the following rules:

BITOR	00	01	10	11
00	00	01	10	11
01	01	01	11	11
10	10	11	10	11
11	11	11	11	11

Using the above chart, you see that %01 BITOR %11 has the result of %11. COMAL allows binary constant numbers if preceded by the %. While the above chart covers a two-digit BITOR operation, COMAL can handle up to 16 digits with BITOR.

NOTES

(1) The arguments for the BITOR operator must be in the range of 0-65535, decimal or $00-$ffff, hex.
(2) See Appendix M for more information on binary numbers.

SYNTAX

```
<argument> BITOR <argument>

  <argument> is a non-negative number in the range 0-65535.
```

EXAMPLES

```
RETURN PEEK($24B) BITOR %00000100
result:=number1 BITOR number2
```

SAMPLE PROGRAM

```
// print chart of BITAND BITOR BITXOR operations
DIM bin$(0:3) OF 2
bin$(%00):="00"; bin$(%01):="01"
bin$(%10):="10"; bin$(%11):="11"
PAGE
PRINT "A  B  ! AND  OR   XOR"
FOR a:=%00 TO %11 DO
  FOR b:=%00 TO %11 DO
    PRINT bin$(a);bin$(b);"!";
    PRINT bin$(a BITAND b);" ";
    PRINT bin$(a BITOR b);" ";
    PRINT bin$(a BITXOR b)
  ENDFOR b
ENDFOR a
END "All done."

RUN
A  B  ! AND  OR   XOR
00 00 ! 00   00   00
00 01 ! 00   01   01
00 10 ! 00   10   10
00 11 ! 00   11   11
01 00 ! 00   01   01
01 01 ! 01   01   00
01 10 ! 00   11   11
01 11 ! 01   11   10
10 00 ! 00   10   10
10 01 ! 00   11   11
10 10 ! 10   10   00
10 11 ! 10   11   01
11 00 ! 00   11   11
11 01 ! 01   11   10
11 10 ! 10   11   01
11 11 ! 11   11   00
All done.
```

ADDITIONAL SAMPLE SEE: POKE
SEE ALSO: BITAND, BITXOR, PEEK, POKE

CATEGORY: Operator
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the bitwise XOR of the two numbers. BITXOR performs the bitwise XOR operation bit by bit on the two numbers. BITXOR follows the following rules:

BITXOR	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

Using the above chart, you see that %01 BITXOR %11 has the result of %10. COMAL allows binary constant numbers if preceded by the %. While the above chart covers a two-digit BITXOR operation, COMAL can handle up to 16 digits with BITXOR.

NOTES

(1) The arguments for the BITXOR operator must be in the range of 0-65535, decimal or $00-$ffff, hex.
(2) See Appendix M for more information on binary numbers.

SYNTAX

```
<argument> BITXOR <argument>

  <argument> is a non-negative number in the range 0-65535
```

EXAMPLES

```
POKE $e84c,PEEK($e84c) BITXOR %00000010
result:=number1 BITXOR number2
 (the next line will reverse a PET/CBM screen)
FOR x:=$8000 TO $87d0 DO POKE x,PEEK(x) BITXOR %10000000
```

SAMPLE PROGRAM

```
//compute parity of 8 bits - odd parity
REPEAT
  INPUT "Number (0 to STOP): " : number;
  PRINT "Parity of";parity(number)
UNTIL number=0
END "All done."

FUNC parity(n) CLOSED
  n:=n BITXOR (n DIV 16)
  n:=n BITXOR (n DIV 4)
  n:=n BITXOR (n DIV 2)
  RETURN n BITAND 1
ENDFUNC

RUN
Number (0 to STOP): %0110 Parity of 0
Number (0 to STOP): %1110 Parity of 1
Number (0 to STOP): 0 Parity of 0
All done.
```

SAMPLE PROGRAM

```
PRINT "BITNOT simulation using BITXOR - with $ff"
REPEAT
  INPUT "Number (0 to STOP): " : number
  PRINT "BITNOT";number;"equals";number BITXOR $ff
UNTIL number=0
END "All done."

RUN
BITNOT simulation using BITXOR - with $ff
Number (0 to STOP): 10
BITNOT 10 equals 245
Number (0 to STOP): 245
BITNOT 245 equals 10
Number (0 to STOP): 0
BITNOT 0 equals 255
All done.
```

ADDITIONAL SAMPLE SEE: BITOR
SEE ALSO: BITAND, BITOR, PEEK, POKE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Begins a CASE structure. This structure allows a multiple-choice decision based on the current value of the ⟨control expression⟩. The current value of ⟨control expression⟩ is compared to each specific case in the structure. A specific case is a series of one or more values of the same type (string or numeric) as the ⟨control expression⟩, listed after the keyword WHEN. If omitted, the keyword OF will be provided by the system.

NOTES

(1) Once a case matches, the statements belonging to the matching case are executed and no further comparing is done.
(2) If no case matches, the statements belonging to the OTHERWISE are executed. If there is no OTHERWISE specified, an error condition will result.
(3) See Appendix A for an explanation of the CASE structure.

SYNTAX

```
CASE <control expression> [OF]

 <control expression> is an <expression>
   may be either string or numeric
```

EXAMPLES

```
CASE guess OF
CASE text$ OF
CASE money+bet OF
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "HOW MANY HAMBURGERS DID YOU EAT: " : number
UNTIL number>=0
CASE number OF
WHEN 0
  PRINT "YOU MIGHT STARVE"
WHEN 1
  PRINT "JUST AN APPETIZER"
WHEN 2,3
  PRINT "NOT BAD"
  PRINT "THAT IS ABOUT HOW MANY I ATE"
WHEN 4,5,6
  PRINT "BIG EATER, HUH."
OTHERWISE
  PRINT "I WON'T PAY YOUR FOOD BILL"
ENDCASE

RUN
HOW MANY HAMBURGERS DID YOU EAT: 5
BIG EATER, HUH.
```

ADDITIONAL SAMPLES SEE: CHAIN, ENDCASE, MOD, OTHERWISE, READ, REF
USED IN PROCEDURE: FETCH
SEE ALSO: ENDCASE, OF, OTHERWISE, WHEN

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Gives a catalog (directory) of all disk files on the diskette currently in the disk drive specified. If no specific drive is requested, version 0.14 uses both drives, while version 2.00 uses the current drive. In version 2.00 a READ ERROR results if a disk is not in a requested drive. To print the catalog (directory) of a disk onto your printer, issue the command SELECT OUTPUT "LP: " prior to your CAT command (this selects the Line Printer).

NOTES

(1) Version 0.14 does not support a unit specification or pattern matching.
(2) In version 2.00 hitting SPACE will pause the directory. Hit SPACE again to resume.
(3) A CAT command with no drive specified has the following result:
Version 0.14: both drives are listed.
Version 2.00: the current drive is listed.
(4) Version 0.14 for the Commodore 64 does not allow a SELECT OUTPUT "LP:" prior to a CAT command.

SYNTAX

```
CAT [<drive>]                          version 0.14

CAT [<filename>]                       version 2.00

 <filename> may include unit and drive information
   and may also include "pattern matching"
 <drive> is a <numeric expression> whose value is 1 or 0;
```

EXAMPLES

```
0.14          2.00
COMMAND       COMMAND        RESULT
CAT 0         CAT "0:"       catalog of drive 0
CAT 1         CAT "1:"       catalog of drive 1
              CAT "1:P*"     drive 1 files first letter P
              CAT "1:*=seq"  drive 1 sequential files
```

SAMPLE EXERCISE (version 2.00)

To print the directory of the current drive onto the printer:

```
SELECT OUTPUT "LP:"     select the printer for output
CAT
 1"WORD PRO 4 AUTO " W1 2A
 3   "LOADWP"            PRG       the catalog will
 72  "WP4"               PRG       print on your printer
 1   "PRINTCHARACTER"    PRG       not on the screen
 588 BLOCKS FREE.                  the catalog will vary
                                   depending upon the disk
SELECT OUTPUT "DS:"                return output to screen
```

ADDITIONAL SAMPLE SEE: SAVE
SEE ALSO: DIR, OPEN, SELECT OUTPUT

CATEGORY: Command / Statement
KERNAL: [NO] VERS 0.14 [+] VERS 2.00 [*]

LOADs and RUNs a program from disk or tape. The program must have previously been stored on disk or tape using the SAVE command. All variables are cleared, and as soon as the program is loaded into the computer's memory it is RUN. Parameter passing is not provided for but may be accomplished by using disk data files. CHAIN allows you to break a program too large for your computer's memory into smaller segments and link (or CHAIN) them together. It is also useful with MENU type selection programs. CHAIN may be used as a direct command (like RUN), causing a program to LOAD and immediately RUN.

NOTES

(1) CHAIN and ENTER commands are not compatible.
(2) CHAIN cannot be used to LOAD and RUN a program that was LISTed to disk or tape.
(3) Version 2.00 allows unit and secondary address information to be included in the ⟨filename⟩. See Appendix N for more information.
(4) For a better alternative to CHAIN with version 2.00 see EXTERNAL, PROC, and FUNC.

SYNTAX

```
CHAIN <filename>
```

EXAMPLES

```
CHAIN "TEST"
CHAIN prog$
```

SAMPLE PROGRAM

```
DIM choice$ OF 1
PRINT "E - EDIT DATA
PRINT "I - INPUT DATA
PRINT "P - PRINT REPORT
REPEAT
  INPUT "WHAT IS YOUR CHOICE: " : choice$
  CASE choice$ OF
  WHEN "I"
    CHAIN "INPUT'DATA"
  WHEN "P"
    CHAIN "REPORT"
  WHEN "E"
    CHAIN "EDIT'DATA"
  OTHERWISE
    PRINT "ENTER E, I, OR P"
  ENDCASE
UNTIL FALSE // FOREVER

RUN
E - EDIT DATA
I - INPUT DATA
P - PRINT REPORT
WHAT IS YOUR CHOICE: A
ENTER E, I, OR P
WHAT IS YOUR CHOICE: P
  (The program named "REPORT" is LOADed from disk and RUN.)
```

SEE ALSO: CON, EXTERNAL, FUNC, LOAD, PROC, RUN, SAVE

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the character that has the specified numeric (ASCII) code. May be used to send a special character to a printer (i.e., ESCAPE = CHR$(27)). A quote mark (") is represented as CHR$(34). The complementary function of CHR$ is ORD, which takes a character and returns its numeric (ASCII) code.

SYNTAX

```
CHR$(<numeric val>)

  <numeric val> is a <numeric expression>
    whose value is from 0-255
```

EXAMPLES

```
CHR$(65)
CHR$(char)
CHR$(char+128)
```

SAMPLE PROGRAM

```
//PRINT SOME RANDOM LETTERS
a:=ORD("A"); z:=ORD("Z")
FOR temp:=1 TO 30 DO PRINT CHR$(RND(a,z)),

RUN
GYNJOFQKOGJSXSGKPRLWBAJFJNMKKT
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "NUMBER (1-255 / 0 TO STOP) : " : num
  IF num THEN PRINT num;"IS REPRESENTED BY";CHR$(num)
UNTIL num=0
PRINT "ALL DONE"
```

```
RUN
NUMBER (1-255 / 0 TO STOP) : 65
65 IS REPRESENTED BY A
NUMBER (1-255 / 0 TO STOP) : 90
90 IS REPRESENTED BY Z
NUMBER (1-255 / 0 TO STOP) : 0
ALL DONE
```

WARNING: some values give effects other than a character printed on the screen. Try number 147, your screen will clear

ADDITIONAL SAMPLES SEE: DO, GET$, IN, REF
USED IN PROCEDURES: GET'CHAR, SCANKEY
SEE ALSO: ABS, INT, ORD, STR$, VAL

CATEGORY: Command / Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Closes files currently open. If a ⟨filenum⟩ is specified, only that file is closed. If no ⟨filenum⟩ is specified, all files are closed. If you leave out the keyword FILE, it will be supplied by the COMAL system. No error occurs if a CLOSE is issued for a file that is not currently open (i.e., the first line in your program could be CLOSE). It is very important always to CLOSE all disk or tape files that you OPEN, especially output (WRITE/PRINT) files. A disk file must be closed properly before it can be RENAMEd, COPYed, BACKUPed (DUPLICATEd), or DELETEd. Attempting to DELETE an open file may result in disk errors. COLLECT (VALIDATE) will remove all improperly closed files. See your Commodore Disk Manual for information on these commands. Examples of how to use the commands from COMAL are included with keyword PASS and in Appendix O.

SYNTAX

```
CLOSE [[FILE] <filenum>]

 if <filenum> is omitted, all files are closed
```

EXAMPLES

```
COMMAND             RESULT
CLOSE FILE 2        closes only file 2
CLOSE               closes all files
CLOSE FILE infile   closes only file with value of INFILE
```

SAMPLE PROGRAM

```
CLOSE              make sure all previous files are closed
DIM name$ OF 20, drive$ OF 1
outfile:=3;name$:="HAROLD";high'score:=58
PRINT name$;"HAS THE HIGH SCORE OF";high'score
drive$:="0" // default drive 0
OPEN FILE outfile,"@"+drive$+":SCORE'FILE",WRITE
WRITE FILE outfile : name$,high'score
CLOSE FILE outfile
PRINT "ALL DONE"

RUN
  (system closes any files left open)
HAROLD HAS THE HIGH SCORE OF 58
  (system opens a new disk file number 3 named SCORE'FILE)
  (system writes HAROLD to file 3)
  (system writes 58 to file 3)
  (system closes file 3)
ALL DONE
```

ADDITIONAL SAMPLES SEE: APPEND, OPEN, WRITE
SEE ALSO: APPEND, EOF, FILE, OPEN, PRINT, RANDOM, WRITE

CATEGORY: Type of procedure or function
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Declares that all variables and arrays within a procedure or function are to be local (closed to the main program). If parameters are used, they will receive their initial values from the calling statement. If a parameter is preceded by the keyword REF it will be used as an alias for the matching variable in the calling statement, which will be updated along with the alias procedure or function variable. Version 2.00 also requires that any procedure or function called from within a CLOSED procedure or function be declared global with an IMPORT statement or be nested. Procedures and functions are always global in version 0.14. Within a CLOSED procedure or function, only the variables in the ⟨parameter list⟩ or those listed in an IMPORT statement will receive an initial value from outside the procedure.

NOTES

(1) ZONE will automatically be shared with a CLOSED procedure or function.
(2) For more information on the procedure and function structure, see Appendix A.
(3) External procedures and functions are automatically considered CLOSED. See keyword PROC for the full syntax when using EXTERNAL.

SYNTAX

```
PROC <procedure name>[(<parameter list>)] [CLOSED]
  and
FUNC <function name>[(<parameter list>)] [CLOSED]

 <procedure name> is an <identifier>
 <function name> is an <identifier>
 <parameter list> is optional and represented by:
    [REF ]<variable name>{,[REF ]<variable name>}
```

EXAMPLES

```
FUNC counter(integer) CLOSED
PROC newpage CLOSED
    The next example is considered closed:
PROC quicksort(left',right',REF item$()) EXTERNAL "quick.e"
```

SAMPLE PROGRAM

```
test:=1
PRINT "MAIN TEST IS";test
EXEC sample
PRINT "MAIN TEST STILL IS";test
PROC sample CLOSED
  test:=2            no conflict with TEST in main section
  PRINT "PROC TEST IS";test
ENDPROC

RUN
MAIN TEST IS 1
PROC TEST IS 2
MAIN TEST STILL IS 1
```

ADDITIONAL SAMPLES SEE: GET$, STOP
USED IN PROCEDURES: MOST OF THE PROCEDURES AND FUNCTIONS
SEE ALSO: ENDFUNC, ENDPROC, EXEC, EXTERNAL, FUNC, IMPORT, PROC, REF

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Continues (or resumes) program execution following a break caused by hitting the STOP key, an error, a STOP statement, or an END statement (version 2.00 will not allow CON after an END statement, since it is considered the end of the program). Execution resumes with the line following the one executing at the break, except during an INPUT from keyboard statement, which will repeat the INPUT request. All variables are left intact. Program variables may be displayed and changed before resuming execution. If any program lines are added, deleted, or changed, or if new variables are added, it may not be possible to continue program execution (an error message will be displayed instead). CON may only be used as a direct command and may not be part of a COMAL program.

SYNTAX

```
CON
```

EXAMPLE

```
CON
```

SAMPLE EXERCISE

```
10 points:=2
20 PRINT "TESTING THE CON COMMAND"
30 PRINT "--------------------------"
40 PRINT "THERE WERE";points;"POINTS"
50 STOP
60 PRINT "BACK AGAIN"
70 PRINT "YOU CLAIM";points;"POINTS NOW"
80 PRINT "ALL DONE"

RUN
TESTING THE CON COMMAND
----------------------
THERE WERE 2 POINTS

STOP AT 0050

PRINT POINTS
2

POINTS:=0

CON
BACK AGAIN
YOU CLAIM 0 POINTS NOW
ALL DONE
```

SEE ALSO: CHAIN, END, RUN, STOP

CATEGORY: Command / Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Copies one or more files from one disk to another. COPY can also make a duplicate copy of a file on the same disk but with a different name.

NOTES

(1) This command is not supported by version 0.14. It can be simulated using the keyword PASS. See Appendix O for more information.
(2) Both files must be on the same disk unit; however, if the unit is a dual drive, both of its drives may be used. See Appendix N for information on how to specify drive numbers.

SYNTAX

```
COPY <source file name>,<target file name>

 <source file name> is a <filename>
 <target file name> is a <filename>
```

EXAMPLE

```
COPY "temp","mine"          make copy of "temp" called "mine"
COPY "0:r-*","1:*"          copy all file starting with "r-"
COPY "1:*","0:*"            copy all files
COPY "0:test","0:final"     copy test to final on drive 0
```

SAMPLE EXERCISE

(Copy all files names ending with .L)

```
DIM text$ of 80,name$ of 17
SELECT OUTPUT "0:TEMP.DIR" // output to disk file
DIR "0:" // directory goes to disk file
SELECT OUTPUT "DS:" // closes the TEMP.DIR disk file
OPEN FILE 2,"0:TEMP.DIR",READ
INPUT FILE 2: text$ // dummy read of disk name and id
WHILE NOT EOF(2) DO
  INPUT FILE 2: text$
  IF LEN(text$)>23 AND THEN NOT "blocks free" IN text$ THEN
    name$:=text$(8:24)
    name$:=name$(1:(CHR$(34) IN name$)-1)
    IF ".1" IN name$ THEN COPY "0:"+name$,"1:"+name$
  ENDIF
ENDWHILE
CLOSE FILE 2
DELETE "0:TEMP.DIR"
PRINT "ALL DONE"

RUN
(files ending with .L are copied from drive 0 to drive 1)
ALL DONE
```

SEE ALSO: PASS, RENAME

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the cosine of the specified number in radians. One radian equals approximately 57°. The following formulae may be used for radian/degree conversion:

degrees = radians * (180/π) radians = degrees * (π/180)
degrees = radians * 57.2957795 radians = degrees * .0174532925

SYNTAX

```
COS(<numeric expression>)
```

EXAMPLES

```
COS(2)
COS(num1*num2)
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "ENTER ANGLE IN RADIANS (0 TO STOP):" ; angle
  IF angle THEN PRINT "COSINE OF";angle;"IS";COS(angle)
UNTIL angle=0
PRINT "ALL DONE"

RUN
ENTER ANGLE IN RADIANS (0 TO STOP):5
COSINE OF 5 IS .283662186
ENTER ANGLE IN RADIANS (0 TO STOP):25
COSINE OF 25 IS .991202811
ENTER ANGLE IN RADIANS (0 TO STOP):0
ALL DONE
```

SEE ALSO: ATN, SIN, TAN

CATEGORY: Command / Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

CREATEs a relative file of size specified. Although it is possible to create a relative file using the OPEN command, and then expand it as you go, it is over 10 times faster during input/output to first CREATE the file using an estimated size, and then add the records.

NOTE

CREATE is not supported by version 0.14. You may use procedure CREATE listed in Appendix D instead.

SYNTAX

```
CREATE <filename>,<number of records>,<record length>

 <number of records> is a positive <numeric expression>
 <record length> is a positive <numeric expression>
```

EXAMPLES

```
CREATE "MAILLIST",500,140
CREATE file'name$,records,rec'len
```

SAMPLE EXERCISE

```
        create a random file on drive 0 with:
          551 records
          each record 48 bytes long
          file name of "PHONE'LIST"

CREATE "0:PHONE'LIST",551,48
```

SEE ALSO: OPEN, RANDOM

CATEGORY: Statement / Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Positions the CURSOR to the line and positions on that line as specified by the parameters ⟨line⟩ and ⟨position⟩. The screen lines are referred to as lines 1-25, with the top line as line 1. The columns (positions) on each line are referred to as 1-40 on 40 column screens, and as 1-80 on 80 column screens, with the first position on the left as column 1. Positioning the CURSOR is similar to finding your seat at a movie theater. First you find the correct row, then the correct position in the row. This is similar to the ⟨row⟩,⟨col⟩ reference in matrix operations, but differs from ⟨x⟩,⟨y⟩ coordinates.

NOTES

(1) The CURSOR keyword is not supported by version 0.14. It may be simulated using the procedure CURSOR listed in Appendix D.
(2) Specifying 0 as the row means to keep the same row that the cursor presently is on. Thus if you wish to back up the cursor to position 5, but don't know what row it is on, simply code CURSOR 0,5. This same idea also applies to the column position on a row; simply use 0 if you wish the column not to change.

SYNTAX

```
CURSOR <line>,<pos>

 <line> is a <numeric expression> whose value is from 0-25
   (0 means current line)
 <pos> is a <numeric expression> whose value is from 0-80
   or on 40 column screens the value is from 0-40
   (0 means current position)
```

EXAMPLES

```
CURSOR 9,2
CURSOR row,col
CURSOR 0,10          keeps current row and changes column
CURSOR 20,0          keeps current column and changes rows
```

SAMPLE PROGRAM

```
DIM hit$ OF 1
PAGE
PRINT "HIT ANY KEYS, HIT X TO EXIT"
REPEAT
  hit$:=KEY$                    see if a key is hit
  CURSOR 12,40                  40 column screens use 12,20
  PRINT hit$
UNTIL hit$="X"
PRINT "ALL DONE"

RUN
  (screen clears)
HIT ANY KEYS, HIT X TO EXIT
  (every key hit is printed in the center of the screen)
  (hit X and X is printed in the center of the screen
   then on the next line down is printed:)
ALL DONE
```

ADDITIONAL SAMPLES SEE: INTERRUPT, TIME
SEE ALSO: AT, GET$, INPUT, KEY$, PRINT, TAB

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Declares data constants that may be assigned to variables via a READ statement. Data can include both string and numeric values. Any number of numeric or string data, separated by commas, can follow the keyword DATA, up to the limit of the program line length. DATA statements are nonexecutable and may be placed anywhere within the program. When the last data item is read by the program, the system variable EOD is set equal to TRUE (a value of 1). String constants must be enclosed in quote marks. A quote mark (") can be made part of a string constant by using two consecutive quote marks (i.e., "abc""def" will be read as abc"def). Commas (,), colons (:), and semicolons (;) may be part of string data. Each data constant must be of the same type (string or numeric) as the variable it is being assigned to. DATA can be reused after issuing a RESTORE command.

NOTES

(1) In version 2.00, data within a closed procedure or function is regarded as local data.
(2) Numeric data may be hexadecimal or binary numbers in version 2.00. See Appendix M for more information.

SYNTAX

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

  <value> can be either a numeric constant or
    a string constant enclosed in quotes
```

EXAMPLES

```
DATA 1,0,0,1,9
DATA 28,"WALDO"
DATA $ff,$e2,$21,%0110      version 2.00 only
```

SAMPLE PROGRAM

```
DIM name$ OF 20
EXEC name'age
PRINT "ALL DONE"
//
PROC name'age
  RESTORE names                    not used in version 0.14
  REPEAT
    READ age,name$
    PRINT name$;"IS";age";"YEARS OLD"
  UNTIL EOD
names:
  DATA 28,"WALDO",42,"HILDA",57,"JOE ""THE CAT"" BAKER",5
  DATA "TINY"
ENDPROC name'age

RUN
WALDO IS 28 YEARS OLD
HILDA IS 42 YEARS OLD
JOE "THE CAT" BAKER IS 57 YEARS OLD
TINY IS 5 YEARS OLD
ALL DONE
```

ADDITIONAL SAMPLES SEE: PAGE, RESTORE, SELECT OUTPUT, SGN
SEE ALSO: EOD, LABEL, READ, RESTORE

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Deletes lines from the program currently in the computer's memory. Lines may be deleted one at a time, or in consecutive blocks, all at once.

NOTES

(1) To delete line number 10 you cannot just enter a null line 10 (as in Commodore BASIC). You must use the DEL command:

```
DEL 10
```

(2) Delete by procedure or function name is not supported by version 0.14.

SYNTAX

```
DEL <range>

 <range> is represented by
   <procname> or
   <funcname> or
   <line range> represented by
     [<start line>][-][<end line>]
       <start line> is a number from 1-9999
         and is less than <end line>
         if omitted, line 1 is used
       <end line> is a number from 1-9999
         and is greater than <start line>
         if omitted, line 9999 is used
```

SYNTAX VARIATIONS

SYNTAX	MEANING
DEL <line number>	delete one line number
DEL <start line number>-	delete from start line number to end of program
DEL -<end line number>	delete up to end line number from the beginning of program
DEL <start line>-<end line>	delete from the start line up to the last line
DEL <procname>	delete entire procedure
DEL <funcname>	delete entire function

EXAMPLES

COMMAND	RESULT
DEL 250	deletes line 250
DEL 500-	deletes lines 500 thru 9999 inclusive
DEL -50	deletes lines 1 thru 50 inclusive
DEL 100-200	deletes lines 100 thru 200 inclusive
DEL QUICKSORT *	deletes procedure named QUICKSORT

* version 2.00 only

SAMPLE EXERCISE

```
10 PRINT "10"
20 PRINT "20"
30 PRINT "30"

DEL 20
LIST
0010 PRINT "10"
0030 PRINT "30"
```

SEE ALSO: AUTO, EDIT, LIST, RENUM

CATEGORY: Command / Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

DELETEs a file from disk. Several files can be DELETEd at once by using wildcard specifications, as with the Commodore SCRATCH command.

NOTES

(1) Version 0.14 requires that the disk drive number be specified in the file name.
(2) Files may be DELETEd just as with the Commodore BASIC SCRATCH command, including using wildcard specifications.
(3) Version 0.14 may only delete files from drives 0 and 1, while version 2.00 can delete files from any valid drive. See Appendix N for information on how to specify drive numbers.

SYNTAX

```
DELETE <filename>
```

EXAMPLES

```
COMMAND                    RESULT
DELETE "0:MYFILE"          deletes MYFILE from drive 0
DELETE "1:TEMP*"           deletes all files on drive 1
                              beginning with TEMP
```

SAMPLE PROGRAM

```
DIM file'name$ OF 20, test$ OF 40
PRINT "DELETE FILES PROGRAM - USING DRIVE 0"
REPEAT
  INPUT "FILENAME (STOP=STOP):" ; file'name$;
  IF file'name$<>"STOP" THEN
    DELETE "0:"+file'name$
    test$:=status$
    IF test$(5:22)="FILES SCRATCHED,00" THEN
      PRINT "ERROR: NO FILES SCRATCHED"
    ELIF test$(1:2)<>"01" THEN
      PRINT "ERROR:";test$
    ELSE
      PRINT
    ENDIF
  ENDIF
UNTIL file'name$="STOP"
PRINT "ALL DONE"

RUN
DELETE FILES PROGRAM - USING DRIVE 0
FILENAME (STOP=STOP): MYFILE
FILENAME (STOP=STOP): TEMP
FILENAME (STOP=STOP): STOP
ALL DONE
```

SEE ALSO: PASS, RENAME

CATEGORY: Statement / Command
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allocates (dimensions) space for strings. Multiple DIMs, separated by commas without repeating the DIM keyword, may occur on one line. Redimensioning an already dimensioned string is not allowed. A DIM statement may be issued locally if inside a CLOSED procedure. Each time the procedure is executed, the string will be newly dimensioned, and after the procedure is finished, it will be "deallocated." When a string is initially declared (dimensioned), its value is set to "" (null). The actual value and length of the string may vary throughout the program, but it may never have more than the number of characters specified in the initial DIM statement (excess characters are truncated off the right - i.e., with DIM N$ of 2, "ABCDE" becomes "AB"). A string parameter of a procedure is automatically dimensioned to the length of the actual parameter passed to it when the procedure is executed. For more information on strings see APPENDIX B.

NOTES

(1) A DIM statement is not allowed inside a control structure (IF, CASE, FOR, REPEAT, WHILE, LOOP, TRAP).
(2) A string can be dimensioned for 0 characters. This might be useful in the case where you are reading data you are going to ignore. Example:

```
DIM dummy$ OF 0
OPEN FILE 5, "DATA'FILE", READ
dummy$: =GET$(5, 20) // skip first 20 characters
etc., ...
```

SYNTAX

```
DIM <string variable name> OF <max num of chars>

 <max num of chars> is a non-negative <numeric expression>
```

EXAMPLES

```
DIM name$ OF 20        allows up to 20 characters for NAME$
DIM player$ OF max     up to value of MAX chars allowed
DIM a$ OF 1,b$ OF 9    A$ is allowed only 1 character
                       B$ may have up to 9 characters
```

SAMPLE PROGRAM

```
DIM name$ OF 20,food$ OF 20
INPUT "WHAT IS YOUR NAME: " : name$
INPUT "WHAT IS YOUR FAVORITE FOOD: " : food$
PRINT "I SEE THAT YOU LIKE";food$,",";name$

RUN
WHAT IS YOUR NAME: SIMON
WHAT IS YOUR FAVORITE FOOD: PIZZA
I SEE THAT YOU LIKE PIZZA, SIMON
```

ADDITIONAL SAMPLES SEE: CLOSE, ORD
USED IN PROCEDURES: FETCH, LOWER'TO'UPPER
SEE ALSO: DIM (string arrays), DIM (numeric arrays), NEW, PROC, SIZE

CATEGORY: Statement / Command
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allocates (dimensions) space for string arrays. Multiple DIMs, separated by commas without repeating the DIM keyword, may occur on one line. Redimensioning an already dimensioned array is not allowed. A DIM statement may be issued locally if inside a CLOSED procedure. Each time the procedure is executed, the string array will be newly dimensioned, and after the procedure is finished, it will be "deallocated." When a string array is initially declared (dimensioned), the value of each of its elements is set to "" (null). The actual value and length of each string in the array may vary throughout the program, but it may never have more characters than specified in the initial DIM statement (excess characters are truncated off the right — i.e., with DIM N$ of 1, "ABC" becomes "A"). Total array size is limited by the available memory. Arrays may have up to 33 dimensions. This is due to the 80-character program line length limitation when entering the DIM statement. Each dimension may have whatever top and bottom limit you wish, with the available memory as a limitation. If no lower limit is specified, 1 is used. For more information about string arrays, see Appendix B.

NOTE

A DIM statement is not allowed inside a control statement (IF, CASE, FOR, REPEAT, WHILE, LOOP, TRAP).

SYNTAX

```
DIM <string array name>(<array index>) OF <max chars each>

 <string array name> is a <string variable name>
 <array index> is represented by:
   [<bottom lim>:]<top limit>{,[<bottom lim>:]<top limit>}
     <bottom lim> is an optional <numeric expression>
       if omitted, the default value is 1
     <top limit> is a <numeric expression>
       it must be greater than the <bottom limit>
 <max chars each> is a positive <numeric expression>
```

EXAMPLES

```
STATEMENT              MEANING
DIM play$(4) OF 20     4 strings of up to 20 chars each
DIM play$(1:4) OF 20   the same as the first
                    (strings are referrenced by 1, 2, 3, or 4)
DIM item$(5:7) OF 10   3 strings of up to 10 chars each
                    (strings are referrenced by 5, 6, or 7)
DIM a$(1:4,1:2) OF 6   two dimension array, same as below
DIM a$(4,2) OF 6       the same as the above
                  (strings referrenced by 1,1  1,2  1,3  1,4
                                       and 2,1  2,2  2,3  2,4)
```

SAMPLE PROGRAM

```
INPUT "HOW MANY PLAYERS: " : number
DIM name$(number) OF 20
FOR num:=1 TO number DO
  PRINT "PLAYER NUMBER";num;"NAME PLEASE",
  INPUT ": " : name$(num)
ENDFOR num
FOR num:=1 TO number DO PRINT "PLAYER";num;"IS";name$(num)

RUN
HOW MANY PLAYERS: 2
PLAYER NUMBER 1 NAME PLEASE: JIM
PLAYER NUMBER 2 NAME PLEASE: SUE
PLAYER 1 IS JIM
PLAYER 2 IS SUE
```

ADDITIONAL SAMPLES SEE: AND, NOT, READ, REF, RETURN, SGN
SEE ALSO: DIM (strings), DIM (numeric arrays), NEW, PROC, SIZE

CATEGORY: Statement / Command
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allocates (dimensions) space for numeric arrays. Multiple DIMs, separated by commas without repeating the DIM keyword, may occur on one line. Redimensioning an already dimensioned numeric array is not allowed. A DIM statement may be issued locally if inside a CLOSED procedure. Each time the procedure is executed, the array will be newly dimensioned, and after the procedure is finished, it will be "deallocated." The value of each element of the array is set to 0 when dimensioned. Total array size is limited by the memory available. Arrays may have up to 36 dimensions. This is due to the 80-character program line length limitation when entering the DIM statement. Each dimension may have whatever top and bottom limit you wish, with the available memory as a limitation. If no lower limit is specified, 1 is used. A DIM may be issued locally if inside a CLOSED procedure.

NOTE

A DIM statement is not allowed inside a control statement (IF, CASE, FOR, REPEAT, WHILE, LOOP, TRAP).

SYNTAX

```
DIM <array name>(<array index>)

 <array name> is a <numeric variable name>
 <array index> is represented by:
   [<bottom lim>:]<top limit>{,[<bottom lim>:]<top limit>}
     <bottom lim> is a <numeric expression>
       it is optional in version 0.14 and defaults to 1
     <top limit> is a <numeric expression>
       it must be greater than the <bottom limit>
```

EXAMPLES

```
DIM array(-3:0)                    a one dimensional array
DIM area(3,3,5)                    a three dimensional array
DIM marker(5:6,5:6)                a two dimensional array
DIM points(min:max),a(15),b(13)
```

SAMPLE PROGRAM

```
INPUT "LOW LIMIT: " : low
INPUT "HIGH LIMIT: " : high
DIM score(low:high)
REPEAT
  INPUT "ENTER SCORE (0 TO STOP): " : temp
  IF temp>=low AND temp<=high THEN score(temp):+1
UNTIL temp=0
FOR x:=low TO high DO PRINT "SCORE";x;"OCCURANCES";score(x)

RUN
LOW LIMIT: 86
HIGH LIMIT: 91
ENTER SCORE (0 TO STOP): 86
ENTER SCORE (0 TO STOP): 88
ENTER SCORE (0 TO STOP): 88
ENTER SCORE (0 TO STOP): 91
ENTER SCORE (0 TO STOP): 0
SCORE 86 OCCURANCES 1
SCORE 87 OCCURANCES 0
SCORE 88 OCCURANCES 2
SCORE 89 OCCURANCES 0
SCORE 90 OCCURANCES 0
SCORE 91 OCCURANCES 1
```

ADDITIONAL SAMPLE SEE: ORD
SEE ALSO: DIM (strings), DIM (string arrays), NEW, PROC, SIZE

CATEGORY: Command / Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Gives a DIRectory of all disk files on the diskette currently in the disk drive specified by ⟨drive⟩. If no specific drive is requested, the current drive is used. A READ ERROR results if a disk is not in a requested drive. To print the DIRectory of a disk onto your printer, issue the command SELECT OUTPUT "LP:" prior to your DIR command (this selects the Line Printer).

NOTES

(1) DIR is not supported in version 0.14. Use CAT instead.
(2) Hitting SPACE will pause the directory. Hit SPACE again to resume.
(3) Only disk drives may be used as the specified unit. See Appendix N for information on how to specify drive numbers.

SYNTAX

```
DIR [<filename>]
```

EXAMPLES

```
COMMAND       RESULT
DIR           directory of current drive
DIR "0:"      directory of drive 0
DIR "1:"      directory of drive 1
DIR "1:P*"    directory of drive 1 files beginning with P
DIR "1:*=seq" directory of sequential files on drive 1
```

SAMPLE EXERCISE (to print the directory of drive 1 onto a disk in drive 0)

```
SELECT OUTPUT "0:DIRECTORY"
DIR "1:"
SELECT OUTPUT "DS:"

 (sequential ASCII file named DIRECTORY now contains the
   directory of drive 1)
```

ADDITIONAL SAMPLE SEE: COPY
SEE ALSO: CAT, OPEN, SELECT OUTPUT

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

DISCARDs all resident machine language packages and libraries. In order to release the extra memory space, all names become undeclared. This also prevents calls to now DISCARDed routines. The name table for the resident COMAL program may be restored by either a SCAN or RUN command, but this will not restore the DISCARDed packages.

NOTES

(1) DISCARD is not supported by version 0.14.
(2) The current COMAL program is not lost or removed from memory by the DISCARD statement.

SYNTAX

```
DISCARD
```

EXAMPLES

```
DISCARD              all packages are removed from memory
```

SAMPLE EXERCISE

```
10 PROC test
20 PRINT "testing"
30 ENDPROC test
LINK "GRAPHICS.P"          link a package to the program
USE GRAPHICS               use the package
RUN
end at 0030
TEST                  call the procedure from direct mode
testing
DISCARD                    discard the package
TEST                call the procedure after discard issued
test: unknown statement or procedure
SCAN                       restore the name table
TEST                  call the procedure from direct mode
testing
DISPLAY                    display the current program
PROC test             the program was not lost after discard
  PRINT "testing"
ENDPROC test
```

SEE ALSO: LINK, USE

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [-] VERS 2.00 [*]

DISPLAYs the specified program lines without line numbers. If no specific lines are indicated, all lines are displayed. You may specify a procedure or function name instead of the specific line number range. To store program lines onto disk or tape in ASCII format simply include a ⟨filename⟩. To avoid confusion with files SAVEd or LISTed to disk, you should end the ⟨filename⟩ with .D when DISPLAYing lines to disk. A program DISPLAYed to disk or tape cannot be retrieved later via the ENTER or MERGE command. To DISPLAY lines to your printer, issue the command DISPLAY "LP:". COMAL structures are indented as shown in the sample programs in this Handbook. Output is returned to the screen after a DISPLAY command (DISPLAY cancels any previous "LP:" selection).

NOTES

(1) To slow down the listing speed on the screen hold down the left arrow key (CMB 8000 series), RVS key (CBM/PET 4000, 3000, and 2000 series), or CTRL key (CBM 64).
(2) Hit SPACE to pause a listing. Hit SPACE again to resume.
(3) DISPLAY is not supported by version 0.14.
(4) A program DISPLAYed to disk or tape can later be read with INPUT FILE statements.
(5) A program DISPLAYed to disk or tape cannot later be ENTERed or MERGEd.
(6) Use EDIT instead of DISPLAY if you do not want the structures indented.
(7) A program DISPLAYed to disk is a SEQ (sequential) type file.
(8) To DISPLAY a procedure or function simply type DISPLAY ⟨procedure or function name⟩.

SYNTAX

```
DISPLAY [<range>][ TO <filename>]

 <range> is represented by
   <procname> or
   <funcname> or
   <line range> represented by
     [<start line>][-][<end line>]
       <start line> is a number from 1-9999
         and is less than <end line>
         if omitted, line 1 is used as the start line
       <end line> is a number from 1-9999
         and is greater than <start line>
         if omitted, line 9999 is used
 <filename> is optional and may not be a variable
```

EXAMPLES

COMMAND	RESULT
DISPLAY	displays all lines
DISPLAY "LP:"	displays lines on printer
DISPLAY -500	displays lines 0-500
DISPLAY 9000-	displays lines 9000-9999
DISPLAY 300-400	displays lines 300-400
DISPLAY 8000- TO "TRANSFER.D"	displays lines 8000-9999 to disk file TRANS.D
DISPLAY quicksort TO "LP:"	displays QUICKSORT proc to the printer

SAMPLE EXERCISE

```
10 FOR temp:=1 TO 20
30 ENDFOR temp          note that the lines may be entered
20 PRINT temp;          in any order you wish

DISPLAY
FOR temp:=1 TO 20 DO
  PRINT temp;               indentation provided by COMAL
ENDFOR temp

RUN
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

SEE ALSO: AUTO, DEL, EDIT, ENTER, LIST, MERGE, RENUM

CATEGORY: Operator
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Provides division with an integer answer. X DIV Z is equivalent to INT(X/Z). DIV is a useful operator when you are not concerned with a full decimal point answer. It may be used in conjunction with the MOD operator to obtain division answers in the form of an integer with a remainder.

NOTE

Division by 0 is not allowed, therefore the ⟨divisor⟩ may not have a value of 0.

SYNTAX

```
<dividend> DIV <divisor>

 <dividend> is a <numeric expression>
 <divisor> is a non-zero <numeric expression>
```

EXAMPLES

```
25 DIV 4
score DIV number
```

SAMPLE PROGRAM

```
PRINT "ENTER 0 AS THE 1ST NUMBER TO STOP"
REPEAT
 INPUT "NUMBER: " : numb1;
 IF numb1 THEN
  INPUT "DIVIDED BY: " : numb2
  PRINT numb1;"DIVIDED BY";numb2;"IS";(numb1 DIV numb2);
  IF numb1 MOD numb2 THEN PRINT "REMAINDER";numb1 MOD numb2;
 ENDIF
 PRINT                        provide the carriage return
UNTIL numb1=0
PRINT "ALL DONE"

 RUN
 ENTER 0 AS THE 1ST NUMBER TO STOP
 NUMBER: 25 DIVIDED BY: 4
 25 DIVIDED BY 4 IS 6 REMAINDER 1
 NUMBER: 33 DIVIDED BY: 11
 33 DIVIDED BY 11 IS 3
 NUMBER: 0
 ALL DONE
```

ADDITIONAL SAMPLES SEE: IF, MOD
SEE ALSO: EXP, INT, MOD, SQR

CATEGORY: Special
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Following a FOR or WHILE, DO lets you know that ⟨statements⟩ to be executed follow. Except with one-line structures, DO is optional, and when left out will be supplied for you. See Appendix A for a description of the WHILE and FOR structures.

SYNTAX

```
      (one line)

FOR <var>:=<start> TO <end> [STEP <step>] DO <statement>
    or
WHILE <comparison> DO <statement>

      (multi-line)

FOR <var>:=<start> TO <end> [STEP <step>] [DO]
    or
WHILE <comparison> [DO]

  <var> is a numeric variable
  <start> is a <numeric expression>
  <end> is a <numeric expression>
  <step> is a <numeric expression>
  <comparison> is an <expression>
```

EXAMPLES

```
FOR temp:=1 TO 39 DO PRINT "*";
WHILE NOT EOF(2) DO READ boys(score)
```

SAMPLE PROGRAM

```
FOR temp:=1 TO 6 DO PRINT "*";

RUN
* * * * * *
```

SAMPLE

```
shift:=152                          C64 use 653
x:=ORD("X"); y:=ORD("Y")
PRINT "PRESS SHIFT TO STOP"
WHILE PEEK(shift)=0 DO PRINT CHR$(RND(x,y))

RUN
PRESS SHIFT TO STOP
XYYXYXYYXXYXYYXYXYYYXYXXYXYXX
  (stops when you depress the SHIFT key)
```

SAMPLE EXERCISE

```
10 FOR test:=1 TO 3
20 PRINT test
30 ENDFOR

LIST
0010 FOR test:=1 TO 3 DO      COMAL inserts the DO for you
0020   PRINT test             COMAL indents within
structures
0030 ENDFOR test

RUN
1
2
3
```

ADDITIONAL SAMPLES SEE: EXEC, NOT, READ
USED IN PROCEDURE: CURSOR
SEE ALSO: FOR, WHILE

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [+] VERS 2.00 [*]

Lists lines of the program in the computer's memory, similar to LIST but with no indentations. Version 0.14 lists the lines continuously, while version 2.00 lists them one line at a time. As each line is listed, the cursor is placed at the first position on the line after the line number, and you may edit it if you wish. Simply hit the return key and the next line will be displayed. In version 2.00 hold the return key down and the listing will be continuous. Or you may use the cursor up to move up to a previously displayed line and then cursor down to where you left off and continue EDIT. Any range of line numbers may be EDITed in this manner.

NOTES

(1) EDIT listing one line at a time is not supported by version 0.14.
(2) If a SELECT OUTPUT to printer or disk is in effect, all lines are listed without waiting for you to hit the return key.

SYNTAX

```
EDIT [<range>]

  <range> is represented by
    <procname> or
    <funcname> or
    <line range> represented by
      [<start line>][-][<end line>]
        <start line> is a number from 1-9999
          and is less than <end line>
          if omitted, line 1 is used as the start line
        <end line> is a number from 1-9999
          and is greater than <start line>
          if omitted, line 9999 is used as the end line
```

SYNTAX variations	MEANING
EDIT	edits all lines, one at a time
EDIT <line number>	edits only line specified
EDIT <start line number>-	edits from start line number to the end of the program one line at a time
EDIT <procedure name>	edits lines in the procedure one at a time

EXAMPLES

COMMAND	RESULT
EDIT 250	edits line 250
EDIT 9000-	edits lines 9000 thru end of program
EDIT 500-600	edits lines 500 - 600
EDIT quicksort	edits the procedure named QUICKSORT

SAMPLE EXERCISE

```
EDIT
0010 WHILE NOT EOD DO
0020 READ test          note: these are listed one at a time
0030 PRINT test;
0040 ENDWHILE
```

SEE ALSO: AUTO, DEL, DISPLAY, FIND, LIST, RENUM

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows conditional statement execution depending on the value of the ⟨expression⟩. ELIF is short for ELSE IF and is part of the IF structure. If the ⟨expression⟩ is TRUE (a value not equal to 0) the ‹statements› following the THEN are executed, otherwise they are skipped. A multilayered structure of ELIFs may be easier to understand than many nested IF structures. See Appendix A for a description of the IF structure. If omitted, the keyword THEN will be supplied by the system.

SYNTAX

```
ELIF <expression> [THEN]
  <statements>
```

EXAMPLE 1

```
ELIF item$="YES" THEN
  EXEC instructions
```

EXAMPLE 2

```
ELIF errors>5 THEN
  PRINT "OOPS!!!"
```

SAMPLE PROGRAM

```
DIM choice$ OF 1
REPEAT
  INPUT "ENTER A LETTER (* TO STOP): " ; choice$;
  IF choice$ IN "AEIOU" THEN
    PRINT "VOWEL"
  ELIF choice$ IN "BCDFGHJKLMNPQRSTVWXYZ" THEN
    PRINT "CONSONANT"
  ELIF choice$ IN "1234567890" THEN
    PRINT "DIGIT"
  ELSE
    PRINT "OTHER"
  ENDIF
UNTIL choice$="*"
PRINT "ALL DONE"

RUN
ENTER A LETTER (* TO STOP): R CONSONANT
ENTER A LETTER (* TO STOP): U VOWEL
ENTER A LETTER (* TO STOP): # OTHER
ENTER A LETTER (* TO STOP): 2 DIGIT
ENTER A LETTER (* TO STOP): * OTHER
ALL DONE
```

ADDITIONAL SAMPLES SEE: ELSE, THEN
SEE ALSO: IF, ELSE, ENDIF, THEN

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Provides alternative statements to execute when all IF and ELIF conditions in the IF structure evaluate to FALSE (a value of 0). See Appendix A for a description of the IF structure.

SYNTAX

```
ELSE
  <statements>
     or
<condition> OR ELSE <condition>

  <condition> is a <numeric expression>
```

EXAMPLE 1

```
ELSE
  EXEC instructions
```

EXAMPLE 2

```
ELSE
  PRINT "TRY AGAIN PLEASE"
```

EXAMPLE 3

```
IF spot>0 OR ELSE reply$(spot)="X" THEN PRINT "X is it"
```

SAMPLE PROGRAM

```
DIM choice$ OF 1
REPEAT
  INPUT "ENTER X-EXIT, Y-YES, OR N-NO: " : choice$;
  IF choice$="Y" THEN
    PRINT "YES"
  ELIF choice$="N" THEN
    PRINT "NO"
  ELIF choice$="X" THEN
    PRINT "EXIT"
  ELSE
    PRINT "TRY AGAIN"
  ENDIF
UNTIL choice$="X"
PRINT "ALL DONE"

RUN
ENTER X-EXIT, Y-YES, OR N-NO: Q TRY AGAIN
ENTER X-EXIT, Y-YES, OR N-NO: N NO
ENTER X-EXIT, Y-YES, OR N-NO: Y YES
ENTER X-EXIT, Y-YES, OR N-NO: X EXIT
ALL DONE
```

ADDITIONAL SAMPLES SEE: ELIF, ENDWHILE, IF, MOD, OR, THEN
USED IN PROCEDURES: SCANKEY, VALUE
SEE ALSO: IF, ELIF, ENDIF, OR, THEN

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [+] VERS 2.00 [*]

Terminates (halts) program execution and returns to interactive mode. An END statement may occur at any point in the program and there may be more than one END statement in a program. END and STOP both halt program execution. After halted by an END statement, program execution may be restarted with the CON command only in version 0.14. Version 2.00 considers END to be the end of the program which means there is nothing left to continue executing. Thus STOP should be used for "break points" during program development. Variables remain intact and may be displayed and changed before restarting program execution. If lines are changed, deleted, or added, or if variables are added, execution may not be restartable. Unless an ending message is supplied, the following message is displayed when the program ends (0030 represents the line number where the END was encountered):

```
END AT 0030
```

NOTE

The optional message is not supported by version 0.14. It allows you to have your program end without the abrupt 'END AT 0090' message. You may provide your own system ending message.

SYNTAX

```
END [<message>]

 <message> is a <string expression>
```

EXAMPLES

```
END
END "SAMPLE PROGRAM IS FINISHED"
```

SAMPLE EXERCISE

```
AUTO
0010 DIM test$ OF 1
0020 REPEAT
0030 INPUT "TYPE 0 TO STOP: " : test$
0040 UNTIL test$="0"
0050 END
0060 PRINT "ALL DONE"

 RUN
 TYPE 0 TO STOP: T
 TYPE 0 TO STOP: S
 TYPE 0 TO STOP: 0
                          note ALL DONE does not print due
 END AT 0050              to the end statement
```

ADDITIONAL SAMPLES SEE: ENDTRAP, PAGE
SEE ALSO: CHAIN, CON, RUN, STOP

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Marks the end of the CASE structure. All CASE structures require an ENDCASE statement to identify the end of the structure. After the statements in a matching CASE or in the OTHERWISE case are executed, the program continues with the statement after the ENDCASE. See Appendix A for a description of the CASE structure.

SYNTAX

```
ENDCASE
```

EXAMPLE

```
ENDCASE
```

SAMPLE PROGRAM

```
DIM choice$ OF 1
REPEAT
  INPUT "YOUR CHOICE (A,S,H,X): ": choice$;
  CASE choice$ OF
  WHEN "A"
    PRINT "ADD"
  WHEN "S"
    PRINT "SUBTRACT"
  WHEN "H"
    PRINT "HELP"
    PRINT "A=ADD, S=SUBTRACT, X=EXIT"
  WHEN "X"
    PRINT "EXIT"
  OTHERWISE
    PRINT "TRY AGAIN"
  ENDCASE
UNTIL choice$="X"
PRINT "ALL DONE"

RUN
YOUR CHOICE (A,S,H,X): Z TRY AGAIN
YOUR CHOICE (A,S,H,X): H HELP
A=ADD, S=SUBTRACT, X=EXIT
YOUR CHOICE (A,S,H,X): A ADD
YOUR CHOICE (A,S,H,X): S SUBTRACT
YOUR CHOICE (A,S,H,X): X EXIT
ALL DONE
```

ADDITIONAL SAMPLES SEE: CASE, CHAIN, MOD, OTHERWISE, READ, REF
USED IN PROCEDURE: FETCH
SEE ALSO: CASE, OF, OTHERWISE, WHEN

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

Terminates a multiline FOR loop structure. ENDFOR is not used with a one-line FOR statement. The ⟨control variable⟩ used with the FOR must correspond with the one used with its matching ENDFOR. You may leave the ⟨control variable⟩ out however, and the COMAL interpreter will supply it for you. See Appendix A for a description of the FOR structure. Note that the COMAL KERNAL currently specifies the use of NEXT to terminate a FOR loop. However, it appears that it may be updated soon to allow ENDFOR as the FOR terminator, since it then is compatible with the other multiline structure terminators.

NOTES

(1) The system will convert the keyword NEXT to ENDFOR for you.
(2) The ⟨control variable⟩ is considered LOCAL to the FOR structure in version 2.00 avoiding possible variable conflicts.
(3) In version 2.00 you may issue this POKE to have NEXT instead of ENDFOR in a listing: POKE $24B,PEEK($24B) BITOR %00000100.

SYNTAX

```
ENDFOR [<control variable>]

 <control variable> is a <numeric variable name>
   it matches <control variable> in matching FOR statement
   if omitted, it will be supplied by the system
```

EXAMPLES

```
ENDFOR
ENDFOR temp
ENDFOR sizes
```

SAMPLE PROGRAM

```
FOR num1:=1 TO 3 DO
  FOR num2:=3 TO 4 DO
    PRINT num1;"PLUS";num2;"IS";num1+num2
  ENDFOR num2
ENDFOR num1
PRINT "ALL DONE"

RUN
1 PLUS 3 IS 4
1 PLUS 4 IS 5
2 PLUS 3 IS 5
2 PLUS 4 IS 6
3 PLUS 3 IS 6
3 PLUS 4 IS 7
ALL DONE
```

ADDITIONAL SAMPLES SEE: OR, ORD, PEEK
USED IN PROCEDURE: LOWER'TO'UPPER
SEE ALSO: DO, FOR, STEP, TO

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Marks the end of a function. See Appendix A for a description of the PROCEDURE and FUNCTION structures.

SYNTAX

```
ENDFUNC [<function name>]

 <function name> is an optional <identifier>
   must match the function name in matching FUNC statment
   if you do not enter it, COMAL will supply it for you
     matching the function's name
```

EXAMPLES

```
ENDFUNC even
ENDFUNC gcd
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "WHAT NUMBER (0 TO STOP): " : number
  IF even(number)=TRUE THEN
    PRINT number;"IS AN EVEN NUMBER"
  ELSE
    PRINT number;"IS AN ODD NUMBER"
  ENDIF
UNTIL number=0
PRINT "ALL DONE"
//
FUNC even(num)
  IF num MOD 2 = 0 THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  ENDIF
ENDFUNC even

RUN
WHAT NUMBER (0 TO STOP): 15
15 IS AN ODD NUMBER
WHAT NUMBER (0 TO STOP): 20
20 IS AN EVEN NUMBER
WHAT NUMBER (0 TO STOP): 0
0 IS AN EVEN NUMBER
ALL DONE
```

ADDITIONAL SAMPLES SEE: FUNC, RETURN
USED IN FUNCTIONS: EVEN, FILE'EXISTS, ROUND
SEE ALSO: CLOSED, ENDPROC, EXTERNAL, FUNC, PROC, REF, RETURN

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Marks the end of the multiline IF structure. ENDIF is not used with a one-line IF statement. All other IF structures require an ENDIF to identify the end of the structure. See Appendix A for a description of the IF structure.

SYNTAX

```
ENDIF
```

EXAMPLE

```
ENDIF
```

SAMPLE PROGRAM

```
number:=RND(1,10)           a random integer between 1 & 10
trys:=0;lows:=0;highs:=0
REPEAT
  INPUT "YOUR GUESS: " : guess;
  trys:+1                    increment trys counter
  IF guess<number THEN
    PRINT "TOO LOW"
    lows:+1                  increment count of low guesses
  ELIF guess>number THEN
    PRINT "TOO HIGH"
    highs:+1                 increment count of high guesses
  ENDIF
UNTIL guess=number
PRINT "YOU GOT IT IN";trys;"TRYS"
PRINT "GUESSES TOO LOW:";lows;"AND TOO HIGH:";highs

RUN
YOUR GUESS: 7 TOO HIGH
YOUR GUESS: 1 TOO LOW
YOUR GUESS: 4 TOO HIGH
YOUR GUESS: 3 TOO HIGH
YOUR GUESS: 2 YOU GOT IT IN 5 TRYS
GUESSES TOO LOW: 1 AND TOO HIGH: 3
```

ADDITIONAL SAMPLES SEE: DIV, ELIF, ELSE, IF, MOD
USED IN PROCEDURES: FETCH, LOWER'TO'UPPER, SCANKEY
SEE ALSO: ELIF, ELSE, IF, THEN

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Marks the end of the multiline LOOP structure. This structure allows you to set up a loop with the one exit condition in the middle of the structure. This exit condition is specified by an EXIT statement. See Appendix A for a description of the LOOP structure.

NOTE

ENDLOOP is not supported by Version 0.14.

SYNTAX

```
ENDLOOP
```

EXAMPLE

```
ENDLOOP
```

SAMPLE PROGRAM

```
DIM temp$ OF 80, array$(100) OF 80
OPEN FILE 2,"TEST'LOOP",READ
pointer:=0
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: temp$
  EXIT WHEN temp$="*END*"                    version 2.00 only
  pointer:+1
  array$(pointer):=temp$
ENDLOOP
CLOSE FILE 2
FOR test:=1 TO pointer DO PRINT array$(test)
PRINT "ALL DONE"

RUN
THIS IS A SILLY LOOP
THIS IS A SILLY LOOP
THIS IS A SILLY LOOP
RECORD ONE
RECORD TWO
RECORD THREE
ALL DONE
```

ADDITIONAL SAMPLES SEE: LOOP, ERR, EXIT, EXTERNAL
SEE ALSO: EXIT, LOOP

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Marks the end of a procedure. All procedures require an ENDPROC statement to identify the end of the structure. See Appendix A for a description of the PROCEDURE structure.

SYNTAX

```
ENDPROC [<procedure name>]

 <procedure name> is an optional <identifier>
   must match the procedure name in matching PROC statment;
   if you do not enter it, COMAL will supply it for you
```

EXAMPLES

```
ENDPROC sort
ENDPROC take'in
```

SAMPLE PROGRAM

```
DIM name$ OF 20
INPUT "WHAT IS YOUR NAME: " : name$
EXEC welcome(name$)
PRINT "GOOD BYE FOR NOW"
//
PROC welcome(n$)
  PRINT "WELCOME TO COMAL,";n$
  PRINT "HOPE YOU ENJOY IT"
ENDPROC welcome

RUN
WHAT IS YOUR NAME: SYLVESTER
WELCOME TO COMAL, SYLVESTER
HOPE YOU ENJOY IT
GOOD BYE FOR NOW
```

ADDITIONAL SAMPLES SEE: CLOSED, EXEC, REF
USED IN PROCEDURES: IN ALL PROCEDURES
SEE ALSO: CLOSED, EXEC, EXTERNAL, PROC, REF

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Marks the end of an ERROR HANDLER structure. For more information about this structure see Appendix A.

NOTE

ENDTRAP is not supported by Version 0.14.

SYNTAX

```
ENDTRAP
```

EXAMPLE

```
ENDTRAP
```

SAMPLE PROGRAM

```
PAGE // clear the screen
LOOP
  TRAP // catch input error in next statement
    INPUT "Enter a number (0 to stop): ": number
    EXIT WHEN number=0 // won't exit until you enter 0
  HANDLER
    PRINT
    PRINT "Input error. The number 0 will be used instead."
    number:=0 // input error, use 0 for value
  ENDTRAP
  PRINT "Your number is";number
ENDLOOP
END "End of demonstration."

RUN
 (screen clears)
Enter a number (0 to stop): 15
Your number is 15
Enter a number (0 to stop): ABC
Input error. The number 0 will be used instead.
Your number is 0
  (note the error bypassed the EXIT WHEN condition check)
Enter a number (0 to stop): -54.6
Your number is -54.6
Enter a number (0 to stop): 0
End of demonstration.
  (note the print message was bypassed by the EXIT WHEN)
```

ADDITIONAL SAMPLES SEE: ERR, ERRFILE, ERRTEXT$, TRAP
USED IN FUNCTION: FILE'EXISTS
SEE ALSO: ERR, ERRFILE, ERRTEXT$, HANDLER, REPORT, TRAP

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Marks the end of a multiline WHILE structure. ENDWHILE is not used with a one-line WHILE statement. All other WHILE structures require an ENDWHILE statement to identify the end of the structure. See Appendix A for a description of the WHILE structure.

SYNTAX

```
ENDWHILE
```

EXAMPLE

```
ENDWHILE
```

SAMPLE PROGRAM

```
missed:=FALSE
count:=0
WHILE NOT missed DO
  count:+1                                  increment count
  numb1:=RND(1,9)
  numb2:=RND(1,9)
  PRINT count,"> WHAT IS";numb1;"PLUS";numb2;
  INPUT "---> " : reply;
  IF reply=numb1+numb2 THEN
    PRINT "YES"
  ELSE
    PRINT "OOPS"
    missed:=TRUE
  ENDIF
ENDWHILE
PRINT "ANSWERS WERE RIGHT UP TO PROBLEM";count

RUN
1> WHAT IS 4 PLUS 2 ---> 6 YES
2> WHAT IS 5 PLUS 1 ---> 6 YES
3> WHAT IS 9 PLUS 5 ---> 14 YES
4> WHAT IS 8 PLUS 3 ---> 12 OOPS
ANSWERS WERE RIGHT UP TO PROBLEM 4
```

ADDITIONAL SAMPLES SEE: EOD, EOF, GET$, INPUT, NOT, WHILE, WRITE
SEE ALSO: DO, WHILE

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Enters into the computer's memory a program that was stored on disk in ASCII form via a LIST command. In version 0.14 ENTER does *not* first clear the present program from memory, thus allowing you to merge segments from disk into the program currently in the computer. Lines are entered into the program as if they were typed in on the keyboard. If a line being entered via ENTER already exists, the current line will be replaced by the one being ENTERed. To avoid confusion, end the file name of any file LISTed to disk with .L to provide easy identification. Version 2.00 *will* clear the memory before ENTERing the program. To merge program segments in this version, the keyword MERGE is used.

NOTES

(1) LOAD and ENTER commands are not compatible.
(2) To "transfer" a COMAL program from one version to another, first LIST it to disk or tape, then ENTER it into the other version.
(3) ENTER from tape is not supported by version 0.14.
(4) Version 2.00 has ENTER perform an automatic NEW first. Use keyword MERGE to merge sections, without regard to line numbers in the file being merged from disk or tape.
(5) When ENTERing a program from another version, it is possible that some lines may yield SYNTAX errors. If this happens the system will list the line in question with the error message below it. Either correct the line or make the line a remark by inserting a ! or // just after the line number and hit ‹return›. The system will continue to enter the remainder of the program. When it is finished, you then can go back to the problem lines and correct them to match your version.

SYNTAX

```
ENTER ‹filename›

 ‹filename› cannot be a variable
```

EXAMPLES

```
ENTER "GET'CHAR.L"
ENTER "INPROC.L"
ENTER "1:SPECIALPROG.L"
```

SAMPLE EXERCISE

```
                    Version 0.14 only - version 2.00 use
MERGE
 10 EXEC sample

 LIST
 0010 EXEC sample

 ENTER "TESTPROC.L"   contents of file TESTPROC.L may vary

 LIST
 0010 EXEC sample
 9000 PROC sample
 9010   PRINT "THIS IS THE SAMPLE"
 9020 ENDPROC sample

 RUN
 THIS IS THE SAMPLE
```

SEE ALSO: EDIT, LIST, LOAD, MERGE, SAVE

CATEGORY: System function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

EOD means End Of Data. It is used while reading data from DATA statements. As long as more data is available to be read, EOD is set to be equal to FALSE (a value of 0). As soon as the last data item is read, EOD is set to be equal to TRUE (a value of 1). The command RESTORE resets EOD back to FALSE (a value of 0).

NOTE

If there are no data statements in the program, EOD is always equal to TRUE (a value of 1).

SYNTAX

```
EOD
```

EXAMPLE

```
EOD
```

SAMPLE PROGRAM

```
count:=0
WHILE NOT EOD DO
  READ dummy
  count:+1                          increment count
ENDWHILE
PRINT "THERE ARE";count;"DATA ITEMS"
RESTORE                             reset data pointer
WHILE NOT EOD DO
  READ number
  PRINT number;
ENDWHILE
PRINT "ALL DONE"
DATA 1,2,5,99
DATA 78

RUN
THERE ARE 5 DATA ITEMS
1 2 5 99 78 ALL DONE
```

ADDITIONAL SAMPLES SEE: DATA, RESTORE, SELECT OUTPUT, WRITE
SEE ALSO: DATA, EOF, LABEL, READ, RESTORE

CATEGORY: System function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

EOF means End Of File. It is used while reading data from a sequential tape or disk file. When a file to be read is opened, its EOF is set to be equal to FALSE (a value of 0). Once the end of file is reached, EOF for that file is set to be equal to TRUE (a value of 1). Since there is a different EOF for every open file, a file number must be specified (there is no default file). Only one file may be checked for end of file at a time. See Appendix C for more information on sequential files.

NOTE

Using an EOF for a file that is not open will create an error condition.

SYNTAX

```
EOF(<filenum>)
```

EXAMPLES

```
EOF(3)
EOF(infile)
```

SAMPLE PROGRAM

```
DIM visitor$ OF 80
OPEN FILE 2,"VISITORFILE",READ
WHILE NOT EOF(2) DO
  READ FILE 2 : visitor$
  PRINT visitor$
ENDWHILE
CLOSE FILE 2
PRINT "ALL DONE"

RUN
  (disk file # 2 named VISITORFILE is opened for input)
COMAL USERS GROUP       contents of VISITORFILE will vary
  (file number 2 is closed)
ALL DONE
```

ADDITIONAL SAMPLES SEE: INPUT, OPEN
SEE ALSO: CLOSE, EOD, FILE, OPEN, READ, STATUS

CATEGORY: System function
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the error number when an error occurs within an ERROR HANDLER structure. When the error occurs, the HANDLER part of the structure is executed and ERR is assigned the value of the error number that occurred. Outside of the HANDLER section, ERR has the value 0. Thus ERR is only valuable inside the HANDLER section of the TRAP structure. For more information on the ERROR HANDLER structure see Appendix A.

NOTE

ERR is not supported by version 0.14.

SYNTAX

```
ERR
```

EXAMPLES

```
PRINT "ENCOUNTERED ERROR NUMBER";ERR
PRINT "ERROR NUMBER ",ERR,": ",ERRTEXT$
CASE ERR OF
```

SAMPLE PROGRAM

```
PAGE // clear screen
LOOP
  TRAP
    INPUT "Enter a number: ": number
    EXIT // no error occurred
  HANDLER
    PRINT
    CASE ERR OF
    WHEN 206 // input error
      PRINT "A number is expected"
    WHEN 2 // overflow
      PRINT "The number is too large"
    OTHERWISE
      REPORT // Can't handle this error now
    ENDCASE
  ENDTRAP
ENDLOOP
END "The number was "+STR$(number)

RUN
  (screen clears)
Enter a number: ABC
A number is expected
Enter a number: 99e9999
The number is too large
Enter a number: 56
The number was 56
```

SEE ALSO: ENDTRAP, ERRFILE, ERRTEXT$, HANDLER, TRAP

CATEGORY: System function
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the file that was in use when an error occurred within an ERROR HANDLER structure. When the error occurs, the HANDLER part of the structure is executed and ERRFILE is assigned the value of the file in use. If there is no file in use, or if the error was not an input/output error, then the value of ERRFILE will be 0. For more information on the ERROR HANDLER structure see Appendix A.

NOTE

ERRFILE is not supported by version 0.14.

SYNTAX

```
ERRFILE
```

EXAMPLES

```
IF ERRFILE=in'file THEN PRINT "INPUT FILE ERROR"
IF ERRFILE THEN PRINT ERRTEXT$
CASE ERRFILE OF
```

SAMPLE PROGRAM

```
// copy the test data file to a new final file on the disk
// copy 5K at a time - does not use the COPY statement
// APPENDIX A has an expanded example of this concept
TRAP
OPEN FILE 2,"TEST.DAT",READ // your file name may differ
OPEN FILE 3,"FINAL.DAT",WRITE
  WHILE NOT EOF(2) DO
    PRINT FILE 3: GET$(2,5000),
  ENDWHILE
HANDLER
  PRINT
  IF ERRFILE=2 THEN
    PRINT "Error occured with the input file"
  ELIF ERRFILE=3 THEN
    PRINT "Error occured with the output file"
  ELSE
    PRINT "Another type of error has occurred"
  ENDIF
  PRINT errtext$
ENDTRAP
CLOSE
END "Example program finished"

RUN
  (File is copied)
  (The appropriate error message will be printed if needed)
Example program finished
```

SEE ALSO: ENDTRAP, ERR, ERRTEXT$, HANDLER, REPORT

CATEGORY: String system function
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the error message as a text string describing the error which caused the HANDLER part of the ERROR HANDLER structure to be executed. If a HANDLER section is not included in the structure, ERRTEXT$ will be set to equal the null string (" "). For more information on the ERROR HANDLER structure see Appendix A.

NOTE

ERRTEXT$ is not supported by version 0.14.

SYNTAX

```
ERRTEXT$
```

EXAMPLES

```
PRINT ERRTEXT$
error$:=ERRTEXT$
```

SAMPLE PROGRAM

```
// trick the system into listing its error messages
ZONE 6
FOR temp:=0 TO 10 DO // try 1 TO 300 for longer list
  TRAP
    REPORT temp
  HANDLER
    PRINT temp;">",ERRTEXT$
  ENDTRAP
ENDFOR temp

RUN
0 >   no error to report
1 >   function argument error
2 >   overflow
3 >   division by zero
4 >   substring error
5 >   value out of range
6 >   step = 0
7 >   illegal bound
8 >   error in print using
9 >   error
10 >  index out of range
```

SEE ALSO: ENDTRAP, ERR, ERRFILE, REPORT, TRAP

CATEGORY: System variable
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

Allows a COMAL program to disable the STOP key. Use TRAP statements to change the effect of the STOP key. TRAP ESC+ means to enable the STOP key, which is how the system starts. While the STOP key is enabled, hitting the STOP key will stop the program. To disable the STOP key, the statement TRAP ESC- is used. Now when the STOP key is pressed the program is not stopped, but the value of the system variable ESC is set to TRUE (a value of 1). A program can test ESC to watch for the pressing of the STOP key. ESC is set to FALSE (a value of 0) while the STOP key is not depressed (except in version 2.00 where ESC remains TRUE once set to TRUE until its value is checked).

NOTES

(1) While the STOP key is enabled (TRAP ESC+ in effect) the value of ESC will always be FALSE (a value of 0) in a running COMAL program. If the STOP key is hit, the program is stopped and ESC is set to TRUE (a value of 1).
(2) While the STOP key is disabled (TRAP ESC- in effect) in a running COMAL program, the value of the system variable ESC will indicate whether or not the STOP key has been pressed. It will be FALSE UNTIL the STOP key is depressed. Then, version 2.00 sets ESC to TRUE and it remains TRUE until the value of ESC is checked by the program. Version 0.14 returns the value of ESC to FALSE as soon as you no longer are pressing the STOP key.

SYNTAX

```
      (testing the ESC variable)

ESC

      (changing the ESC value)

TRAP ESC<type>

 <type> is one of two symbols, + or -
   + means enable the STOP key
   - means disable the STOP key
```

EXAMPLES

```
STATEMENT       RESULT
TRAP ESC+       enable the STOP key
TRAP ESC-       disable the STOP key
test:=ESC       assign the variable TEST the value of ESC
PRINT ESC       prints the value of ESC
```

SAMPLE PROGRAM

```
TRAP ESC-                       disable the STOP key
OPEN FILE 3,"TESTING",WRITE
PRINT "SAMPLE FILE IS NOW OPEN - HIT STOP KEY WHEN READY"
REPEAT                          wait till STOP is depressed
UNTIL ESC
PRINT "SO YOU WISH TO STOP NOW"
PRINT "FIRST I WILL CLOSE YOUR FILES FOR YOU"
CLOSE
TRAP ESC+                       enable the STOP key again
PRINT "ALL DONE"

RUN
  (STOP key is disabled)
SAMPLE FILE IS NOW OPEN - HIT STOP KEY WHEN READY
  (nothing happens until you hit the STOP key)
SO YOU WISH TO STOP NOW
FIRST I WILL CLOSE YOUR FILES FOR YOU
  (all files are closed)
  (the STOP key is enabled again)
ALL DONE
```

ADDITIONAL SAMPLES SEE: KEY$, TRAP
SEE ALSO: TRAP

CATEGORY: Statement/Command
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Executes a procedure. Allows actual values to be passed to the procedure's formal parameters. Each value passed to the procedure must be of the same type as specified by its matching procedure formal parameter. The same number of values must be passed as expected by the procedure or an error will occur. An entire array may be passed simply by using its array name without the parentheses section. Also, make sure the formal parameter list in the procedure requests an array with the same dimensions. Once the procedure execution is complete, the program continues with the statement after the EXEC statement. Multiple EXEC statements may be made on one line separated by semicolons.

NOTES

(1) The word EXEC is optional, and if omitted will be supplied by the system [except in procedures called in direct mode in version 2.00 with names identical to command keywords (LIST, DISPLAY, EDIT, RUN, SCAN, etc.)].
(2) COMAL can suppress the word EXEC in a listing. See the keyword SETEXEC for more information about this option.
(3) See Appendix A for a description of the PROCEDURE structure.
(4) The system will "remember" all procedure and function names once a program is run (or SCANned in version 2.00). Any procedure can then be executed from direct mode with the EXEC command. This is very powerful, similar to adding your own keywords or program function keys, yet remaining compatible from one system to the next.

SYNTAX

```
[EXEC] <procedure name>[(<actual parameter list>)]

 EXEC is optional
 <procedure name> is an <identifier>
 <actual parameter list> is one or more <expressions>
   each separated by commas
   their values will be passed to the procedure specified
   If called by REF in the procedure the value must be
     a variable or element in an array
```

EXAMPLES

```
EXEC instructions
EXEC error(5)
```

SAMPLE PROGRAM

```
DIM word$ OF 80
FOR temp:=1 TO 3 DO
  INPUT "WORD: " : word$;
  INPUT "COUNT: " : count
  EXEC demo(count,word$)
ENDFOR temp
PRINT "ALL DONE"
//
PROC demo(n,t$) CLOSED
  FOR temp:=1 TO n DO PRINT t$;     variable TEMP is local
  PRINT                             it won't conflict with
ENDPROC demo                        main program

RUN
WORD: COMAL COUNT: 6
COMAL COMAL COMAL COMAL COMAL COMAL
WORD: FANTASTIC COUNT: 2
FANTASTIC FANTASTIC
WORD: EXEC COUNT: 9
EXEC EXEC EXEC EXEC EXEC EXEC EXEC EXEC EXEC
ALL DONE
```

ADDITIONAL SAMPLES SEE: CLOSED, ENDPROC, ENTER, IMPORT, REF
USED IN PROCEDURES: FETCH, GET'VALID, TAKE'IN
SEE ALSO: ENDPROC, LINK, PROC, REF, SETEXEC, USE

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

EXIT provides a method for leaving the LOOP structure. When an EXIT is executed program execution continues with the statement following the ENDLOOP statement. A loop structure should only have one exit or it is not part of structured programming. The EXIT can be conditional as part of an IF statement, or may include the WHEN phrase. If there are no statements prior to your EXIT statement, you should be using a WHILE loop. If there are no statements after your EXIT statement, you should be using a REPEAT loop. See Appendix D for more information on the LOOP structure.

NOTE

The LOOP structure is not supported by version 0.14.

SYNTAX

```
EXIT [WHEN <condition>]

 <condition> is a <numeric expession>
```

EXAMPLES

```
EXIT WHEN trys>3
IF text$="*END*" THEN EXIT
```

SAMPLE PROGRAM

```
DIM temp$ OF 80, array$(100) OF 80
OPEN FILE 2,"TEST'LOOP",READ
pointer:=0
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: temp$
  EXIT WHEN temp$="*END*"
  pointer:+1
  array$(pointer):=temp$
ENDLOOP
CLOSE FILE 2
FOR test:=1 TO pointer DO PRINT array$(test)
PRINT "ALL DONE"

RUN
THIS IS A SILLY LOOP
THIS IS A SILLY LOOP
THIS IS A SILLY LOOP
RECORD ONE
RECORD TWO
RECORD THREE
ALL DONE
```

ADDITIONAL SAMPLES SEE: ENDLOOP, ERR, EXTERNAL, LOOP
SEE ALSO: ENDLOOP, LOOP

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Computes the natural logarithm's base value *e* raised to the power specified. A good representation of *e* is 2.718281828.

SYNTAX

```
EXP(<numeric expression>)
```

EXAMPLES

```
EXP(2)
EXP(5+num)
```

SAMPLE PROGRAM

```
PRINT "EXP PRACTICE"
REPEAT
  INPUT "POWER (0 TO STOP): " : num
  IF num THEN PRINT "ANSWER IS";EXP(num)
UNTIL num=0
PRINT "ALL DONE"

RUN
EXP PRACTICE
POWER (0 TO STOP): 5
ANSWER IS 148.413159
POWER (0 TO STOP): 2
ANSWER IS 7.3890561
POWER (0 TO STOP): 0
ALL DONE
```

SEE ALSO: LOG

CATEGORY: Special
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Identifies that the procedure or function is an EXTERNAL one. When the program is run, the procedure or function body is retrieved and used only when needed. This allows you to have a procedure library disk that you can update as needed. Thus each program will always be calling the latest version of each procedure without the need to update each program individually. Each program will also be significantly smaller, since the procedure body is not included in the program. This also allows several different procedures called at different times to share memory space in a running program. See Appendix A for details on how to create and use EXTERNAL procedures and functions.

NOTES

(1) EXTERNAL is not supported by version 0.14.
(2) There are no restrictions on parameter passing.
(3) An EXTERNAL procedure or function is considered CLOSED, and the keyword CLOSED is not allowed in a program's external PROC or FUNC header statement. (However, the actual procedure or function stored on disk *must* include the word CLOSED in its header.)
(4) An EXTERNAL procedure or function may be called from direct mode once the program it is in is RUN or SCANned.
(5) A procedure SAVEd to disk may be used as an EXTERNAL procedure. It may have remarks or blank lines before it starts. Statements are also permitted to exist after the ENDPROC or ENDFUNC, but these statements will not be executed.
(6) It is suggested to end the ⟨filename⟩ of an EXTERNAL procedure or function with .E or .EXT to avoid future confusion with files.
(7) An EXTERNAL procedure or function may not include any IMPORT statements.

SYNTAX

```
PROC <proc name>[(<formal parms>)][ EXTERNAL <filename>]
   or
FUNC <func name>[(<formal parms>)][ EXTERNAL <filename>]

 <proc name> is an <identifier>
 <func name> is an <identifier>
 <formal parms> is optional and represented by:
   [REF ]<variable name>{,[REF ]<variable name>}
     the initial value of each <variable name> will be
     assigned from the value in the calling statement
     these variables are considered LOCAL
```

EXAMPLES

```
PROC quicksort(left',right',REF item$) EXTERNAL "1:QUICK.E"
FUNC gcd(a,b) EXTERNAL "1:GCD.E"
PROC setup EXTERNAL "1:SETUP.E"
```

SAMPLE EXERCISE

```
//Use large procedures as subsections for the program
DIM reply$ of 1
LOOP
  PAGE // clear screen
  PRINT AT 1,1: "A - ADD"
  PRINT AT 2,1: "S - SUBTRACT"
  PRINT AT 3,1: "X - EXIT"
  REPEAT
    INPUT AT 5,1: "Your choice please: ": reply$
  UNTIL reply$>"" AND THEN reply$ IN "ASXasx"
  CASE reply$ OF
  WHEN "A","a"
    EXEC add'practice
  WHEN "S","s"
    EXEC subtract'practice
  OTHERWISE
    EXIT // finished now
  ENDCASE
ENDLOOP
END "End of example."

PROC add'practice EXTERNAL "1:ADD'PRACTICE.E"
PROC subtract'practice EXTERNAL "1:SUB'PRACTICE.E"

RUN
  (screen clears)
A - ADD
S - SUBTRACT
X - EXIT
```

```
Your choice please: A
  (add'practice module is now executed)
  (this procedure is external --- is on the disk)
  (when the module is done, the screen clears)
A - ADD
S - SUBTRACT
X - EXIT

Your choice please: X
End of example.
```

SEE ALSO: CLOSED, ENDFUNC, ENDPROC, EXEC, FUNC, IMPORT, PROC, REF

CATEGORY: System constant
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

A predefined constant that is always equal to 0 (it can't be changed). It may be used as a numeric expression (i.e., TYPE$(FALSE) has the same meaning as TYPE$(0)).

SYNTAX

```
FALSE
```

EXAMPLE

```
FALSE
```

SAMPLE PROGRAM

```
DIM password$ OF 20, reply$ OF 20
done:=FALSE
password$:="WIDGIT"
PRINT "THE PASSWORD IS";password$
FOR pause:=1 TO 500 DO NULL // WASTE SOME TIME
PRINT "[CLR]" // REPLACE [CLR] WITH CLEAR SCREEN KEY
REPEAT
  INPUT "PASSWORD: " : reply$
  IF reply$=password$ THEN done:=TRUE
UNTIL done
PRINT "YOU GOT IT"
PRINT "ALL DONE"

RUN
THE PASSWORD IS WIDGIT
  (the screen clears)
PASSWORD: WHAT
PASSWORD: PASS
PASSWORD: WIDG
PASSWORD: WIDGIT
YOU GOT IT
ALL DONE
```

ADDITIONAL SAMPLES SEE: CHAIN, ENDWHILE, NOT, OF, OR, RETURN
USED IN PROCEDURE: FETCH
SEE ALSO: TRUE

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Used while editing a program to FIND a variable name or other part of a program line. If a SELECT OUTPUT is in effect, each program line containing the ⟨text$⟩ will be directed to the printer or output file. Otherwise, each line is displayed on the screen, one at a time, and the cursor is placed at the beginning of the ⟨text$⟩ in that line. The line can now be edited if desired. Once the RETURN key is pressed, the search process continues. If the RETURN key is held down, the found lines will be continuously listed.

NOTES

(1) FIND is not supported by version 0.14.
(2) Upper and lower case are considered different by the FIND command. Thus you may should keep this in mind when choosing to FIND all occurences of the keyword REPEAT. If keywords are listed in lower case, then you must search for "repeat" instead of "REPEAT".
(3) If SELECT OUTPUT to disk or printer is in effect, all lines are output one after another without waiting for the RETURN key:

```
SELECT OUTPUT "lp:"     output to printer
FIND " PROC "           all PROC headers print on printer
SELECT OUTPUT "ds:"     output to screen
```

SYNTAX

```
FIND [<range>] <text$>

 <range> is represented by:
   <procname>   or
   <funcname>   or
   <line range> represented by:
     [<start line>][-][<end line>]
       <start line> is a number from 1-9999
         and is less than <end line>
         if omitted, line 1 is used as the start line
       <end line> is a number from 1-9999
         and is greater than <start line>
         if omitted, line 9999 is used as the end line
 <text$> is a string constant
```

EXAMPLES

```
FIND "test"
FIND print'label "name"
FIND 300-400 "temp"
```

SAMPLE EXERCISE (List all the PROC headers on the printer.)

(enter the following program first:)

```
PROC print'stars(number) CLOSED
  FOR temp:=1 to number DO PRINT "*",
ENDPROC print'stars
PROC print'sum(num1,num2) CLOSED
  PRINT num1;"PLUS";num2;"EQUALS";num1+num2
ENDPROC print'sum
PROC double'print(text$) CLOSED
  PRINT text$+text$
ENDPROC double'print
PROC sort(left',right',REF item$) EXTERNAL "1:QUICK.E"

SELECT OUTPUT "LP:"       output now goes to printer
FIND " PROC "
0010 PROC print'stars(number) CLOSED   these 4 lines print
0040 PROC print'sum(num1,num2) CLOSED
0070 PROC double'print(text$) CLOSED
0100 PROC sort(left',right',REF item$) EXTERNAL "1:QUICK.E"
SELECT OUTPUT "DS:"          return output to the screen
```

SEE ALSO: DISPLAY, EDIT, LIST, RENUM

CATEGORY: special
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Specifies that the OPEN, CLOSE, INPUT, PRINT, READ, or WRITE in the statement is to or from a FILE. The FILE is OPENed with an OPEN statement. Multiple variables per INPUT or READ statement are allowed, separated by commas. Multiple variables and/or constants per PRINT or WRITE statement are allowed separated by commas. The word FILE is optional in the OPEN statement, and if omitted will be supplied by the system. The word FILE and the file number are optional in the CLOSE statement and when omitted, all files are closed.

NOTES

(1) INPUT FILE and READ FILE are not compatible.
(2) PRINT FILE and WRITE FILE are not compatible.
(3) INPUT FILE is used to read a file created with PRINT FILE. This type of file is compatible with a standard Commodore BASIC sequential file created with PRINT# statements.
(4) READ FILE is used to read a file created with WRITE FILE.
(5) A FILE may be to a printer, or even to or from the screen.
(6) See Appendix C for more information about sequential files.
(7) The # sign will be converted into the word FILE, but must be preceded by a blank space.
(8) Version 2.00 allows numeric and string constants in the WRITE statement. The numeric constants will be written as real numbers rather than integers.

SYNTAX

(INPUT)

```
INPUT FILE <filenum>[,<recnum>]: <variable list>
```

(PRINT)

```
PRINT FILE <filenum>[,<recnum>]: <value list>
```

(READ)

```
READ FILE <filenum>[,<recnum>]: <variable list>
```

(WRITE)

```
WRITE FILE <filenum>[,<recnum>]: <variable list>  vers 0.14
  or
WRITE FILE <filenum>[,<recnum>]: <value list>     vers 2.00
```

(OPEN)

```
OPEN [FILE] <filenum>,<filename>[,<type>]
```

(CLOSE)

```
CLOSE [[FILE] <filenum>]
```

<variable list> is one or more <variable names> to be used for the operation, separated by commas
<value list> is one or more expressions for the operation, separated by commas
<type> is READ, WRITE, APPEND, or RANDOM <record length>
<record length> is a positive <numeric expression>
<recnum> is the record number for random file access

EXAMPLES

```
INPUT FILE 2 : a$
READ FILE 7 : name$,score
READ FILE 2: text$
WRITE FILE outfile: score
WRITE FILE 2: num1, num2, num3
WRITE FILE 5 : "TESTING",x,y,z          version 2.00 only
PRINT FILE 2 : address
```

SAMPLE PROGRAM

```
DIM text$ OF 80
INPUT "WRITE SOMETHING: " : text$
OPEN FILE 2,"TEXTFILE",WRITE
WRITE FILE 2 : text$
CLOSE FILE 2
PRINT "ALL DONE"
RUN
WRITE SOMETHING: WHAT FUN
  (disk file number 2 named TEXTFILE is opened for output)
  (WHAT FUN is written to file number 2)
  (file number 2 is closed)
ALL DONE
```

ADDITIONAL SAMPLES SEE: CLOSE, GET$
USED IN FUNCTION: FILE'EXISTS
SEE ALSO: APPEND, CLOSE, EOF, INPUT, OPEN, PRINT, READ, WRITE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

A FOR loop is used when a block of statements are to be executed a predefined number of times. The loop uses a ⟨control variable⟩ which is initialized to the ⟨start⟩ value before beginning execution of the block of statements in the loop. If the value of the variable doesn't exceed the ⟨end⟩ value, the statements are executed and the ⟨control variable⟩ is then incremented by the STEP value. If the STEP value is negative, the ⟨control variable⟩ is decremented instead of incremented. The loop is terminated by an ENDFOR statement, except for the one-line FOR statement which must not include an ENDFOR. Note that CBM COMAL converts the keyword NEXT to ENDFOR as the FOR loop terminator.

NOTES

(1) The STEP value may be negative (i.e., STEP -5).
(2) The STEP value may be a noninteger (i.e., STEP .2).
(3) See Appendix A for a description of the FOR structure.
(4) The ⟨controlvar⟩ is considered LOCAL to the FOR structure in version 2.00 avoiding possible variable conflicts.
(5) An integer variable may be used for the ⟨controlvar⟩. This yields much faster execution time.

SYNTAX

```
      (one line FOR)

FOR <var>:=<start> TO <end> [STEP <step>] DO <statement>

      (multi line FOR)

FOR <controlvar>:=<start> TO <end> [STEP <step>] [DO]
  <statements>

 <var> is a <controlvar>
 <controlvar> is a <numeric variable name>
   may also be an <integer variable name>
 <start> is a <numeric expression>
 <end> is a <numeric expression>
 <step> is a <numeric expression>;
   if omitted, the default value is 1
```

EXAMPLES

```
FOR temp:= low TO high STEP 2 DO
FOR score:= 1 TO max DO
FOR spaces:=1 TO 40 DO PRINT " ",
FOR items#:=1 TO 10 DO
```

SAMPLE PROGRAM

```
INPUT "HOW MANY SCORES: " : number'scores
total:=0
FOR temp#:=1 TO number'scores DO
  INPUT "SCORE: " : score
  total:+score                    add SCORE to TOTAL
ENDFOR temp#
PRINT "TOTAL WAS";total;"AVERAGE OF";total/number'scores

RUN
HOW MANY SCORES: 3
SCORE: 80
SCORE: 75
SCORE: 91
TOTAL WAS 246 AVERAGE OF 82
```

ADDITIONAL SAMPLES SEE: EXEC, NEXT, OR, ORD, PEEK, READ, SGN
USED IN PROCEDURES: CURSOR, LOWER'TO'UPPER
SEE ALSO: DO, ENDFOR, STEP, TO

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Start of a function, allowing parameter passing and local or global variables. Functions can be numeric, integer, or string. Somewhere within the function a RETURN statement is used to return a value to the calling statement. Functions can be very flexible, versatile, and easy to use. However, using all the options (REF, CLOSED, IMPORT, and parameter passing), can be complex. A function may call other functions or even call itself. To make the best use of a function, a good programming tutorial is recommended.

Parameter passing is optional. A simple function without parameters is often all that may be needed. A function can be made CLOSED, so that all variables in it are local (unknown to all other parts of the program) except for specific variables made global (shared with other parts of the program) in version 2.00 with an IMPORT statement. ZONE is automatically global even in a CLOSED function.

Strings and entire arrays may be used as parameters. They are dimensioned automatically as they are passed into the function. They can be local (except arrays in version 0.14 must use REF) or used as an alias for the string variable or array in the main program using the keyword REF before the variable name (using REF will use up less memory). Specify that a variable is to be used as an array simply by including the parentheses after the variable name (i.e., SCORES() will be a single dimension array). For multiple dimension arrays, simply include the same number of commas as used when the array was dimensioned (i.e., TABLE(,,) will be a three dimension array).

The values or arrays passed to each function parameter variable or array must of course be of the same type. An error will occur if the function expects a two dimensional array and receives only a single value. An array must be passed in reference (via REF) in version 0.14, while version 2.00 also allows it to be passed as a value parameter.

If a parameter is preceded by the keyword REF, the variable or array in the calling statement will change as the matching function variable or array elements change, even though the function may use a different variable or array name (an alias). If a variable or array is made global with an IMPORT statement, any changes made inside the function will have global effect. Version 2.00 requires that all

functions called from within a closed procedure or function be declared global with an IMPORT statement. Version 0.14 does not support the IMPORT statement, and automatically makes all procedures and functions global within each closed procedure or function. See Appendix A for a description of the function structure including details on how to create and use EXTERNAL functions. Both integer and string functions are allowed in addition to the usual numeric function.

NOTES

(1) The system will "remember" all procedure and function names once a program is run (or SCANned in version 2.00). Any function can then be called from direct mode with the normal function call. This is very powerful, similar to adding keywords or program function keys.
(2) String functions are not supported by version 0.14.
(3) CLOSED and EXTERNAL cannot both be used in the same statement.
(4) An EXTERNAL function is considered CLOSED.

SYNTAX

```
FUNC <funcname>[(<formal parms>)] [EXTERNAL <filename>]
      or
FUNC <funcname>[(<formal parms>)] [CLOSED]

 <funcname> is an <identifier>
   may by <identifier># if used as an integer function
   may by <identifier>$ if used as a string function
 <formal parms> is optional and represented by:
   [REF ]<variable name>{,[REF ]<variable name>}
```

EXAMPLES

```
FUNC even(N) CLOSED
FUNC gcd#(X,Y) EXTERNAL "GCD#.E"
FUNC jiffies
FUNC compact$(text$)
```

SAMPLE PROGRAM

```
DIM type$(FALSE:TRUE) OF 4
type$(FALSE):="ODD"; type$(TRUE):="EVEN"
REPEAT
  INPUT "NUMBER (0 TO STOP): " : number;
  PRINT type$(even(number))
UNTIL number=0
PRINT "ALL DONE"
//
FUNC even(num)
  IF num MOD 2 THEN
    RETURN FALSE
  ELSE
    RETURN TRUE
  ENDIF
ENDFUNC even

RUN
NUMBER (0 TO STOP): 4 EVEN
NUMBER (0 TO STOP): 3 ODD
NUMBER (0 TO STOP): 0 EVEN
ALL DONE
```

SAMPLE PROGRAM (version 2.00 only)

```
DIM text$ of 3
PAGE                                clear the screen
PRINT "I will chop off the first letter of your text"
PRINT "Please type in any 3 letters - or END to stop"
REPEAT
  PRINT AT 10,5: "    "             clear input field
  PRINT AT 12,5: "   "              erase previous reply
  INPUT AT 10,5,3: ">": text$
  PRINT AT 12,5: but'first$(text$)
  FOR temp:=1 TO 999 DO NULL        pause
UNTIL text$="END" OR text$="end"
END "ALL DONE"
//
FUNC but'first$(text$) CLOSED
  IF len(text$)>1 THEN
    RETURN text$(2:len(text$))
  ELSE
    RETURN ""
  ENDIF
ENDFUNC but'first$
```

```
RUN
   (screen clears)
I will chop off the first letter of your text
Please type in any 3 letters - or END to stop
   (the following input and replys are formatted on screen)
>ASD
SD
>QWE
WE
>xyz
yz
>END
ND
ALL DONE
```

ADDITIONAL SAMPLES SEE: ENDFUNC, HANDLER, RETURN
USED IN FUNCTIONS: EVEN, FILE'EXISTS, ROUND
SEE ALSO: CLOSED, ENDFUNC, EXTERNAL, IMPORT, REF, RETURN

CATEGORY: Function
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

GETs the specified number of characters from the keyboard or the selected sequential file. The file must previously have been opened as a READ type file. The length of the string returned will be the length specified, unless the end of file is reached first. Then the string will be only those characters retrieved prior to the end of file (an end of file error will not occur unless you attempt to read from the file one more time). To get a string of characters from the keyboard simply specify a file name of "KB:" in a previous OPEN statement. It will then get the specified number of characters from the keyboard before returning the string. (This is different than KEY$ which simply scans the keyboard buffer once and returns a CHR$(0) if no key has been typed.)

NOTES

(1) GET$ is not supported by version 0.14; however, procedures DISK'GET, GET'CHAR, and SCANKEY can be used as viable substitutes (see Appendix D).
(2) When GETting one character at a time from a disk or tape file, a buffer is used, allowing efficient operation.
(3) See Appendix C for more information about sequential files.
(4) A blinking cursor is turned on during a GET from the keyboard.
(5) The stop key is disabled during a GET.

SYNTAX

```
GET$(<filenum>,<number of characters>)

 <number of characters> is a <numeric expression>
```

EXAMPLES

```
PRINT GET$(2,20)
char$:=GET$(infile,1)
```

SAMPLE EXERCISE

```
PROC directory(drive,mask$) CLOSED
  ZONE 0
  DIM ch$ of 1
  OPEN FILE 2,"u8:$"+STR$(drive)+":"+mask$+"/s0/t+/d+",READ
  ch$:=GET$(2,2)
  WHILE GET$(2,2)<>CHR$(0)+CHR$(0) DO
    PRINT " ",ORD(GET$(2,1))+ORD(GET$(2,1))*256;
    REPEAT
      ch$:=GET$(2,1)
      PRINT ch$,
    UNTIL ch$=chr$(0) // end of directory
  ENDWHILE
  CLOSE FILE 2
ENDPROC directory

RUN
END AT 0140              nothing was executed yet

directory(0,"a*")        direct command

0 "my disk          " ID 2c
12   "add'drill"          prg
31   "assembler"          prg
9    "a2"                 seq
589 blocks free.
```

SAMPLE PROGRAM

```
DIM text$ OF 80, character$ OF 1
file'num:=3
OPEN FILE file'num,"kb:",READ
PRINT "PLEASE TYPE A FEW LETTERS"
LOOP
  character$:=GET$(file'num,1)
  EXIT WHEN character$=CHR$(13)    wait for carriage return
  text$:+character$
ENDLOOP
PRINT
PRINT "YOU JUST TYPED IN THIS MESSAGE:"
PRINT text$
END "ALL DONE"

RUN
PLEASE TYPE A FEW LETTERS
QWERTY HI THERE
YOU JUST TYPED IN THIS MESSAGE:
QWERTY HI THERE
ALL DONE
```

ADDITIONAL SAMPLE SEE: ERRFILE
SEE ALSO: CURSOR, INPUT, KEY$, READ

CATEGORY: Statement
COMAL KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Transfers program execution to the line with the specified ⟨label name⟩. GOTO is rarely needed when other COMAL structures such as REPEAT, WHILE, CASE, and EXEC PROC are used. You may not jump via a GOTO into the middle of a procedure (your computer may lock out). Trying to jump into the middle of a FOR loop will yield an error. However, you are allowed to jump into the other COMAL structures, as well as jump out of any structure, except a procedure. Trying to jump out of a procedure via a GOTO will yield an error message. It must be emphasized that the GOTO is not needed except in rare situations. This is because GOTO is not a structured statement, and COMAL is a structured language. GOTO tends to make a program listing hard to follow.

SYNTAX

```
GOTO <label name>

 <label name> is an <identifier>
```

EXAMPLES

```
GOTO response
GOTO quick'quit
```

SAMPLE PROGRAM

```
DIM item$ OF 80
REPEAT
  INPUT "NEXT ITEM (0 TO STOP): " ; item$
  IF item$="0" THEN GOTO quit
  PRINT item$
UNTIL FALSE                          loop forever
quit:
PRINT "I QUIT. THIS IS BAD PROGRAMMING."

RUN
NEXT ITEM (0 TO STOP): TEST
TEST
NEXT ITEM (0 TO STOP): ITEM NUMBER TWO
ITEM NUMBER TWO
NEXT ITEM (0 TO STOP): 0
I QUIT. THIS IS BAD PROGRAMMING.
```

SEE ALSO: LABEL

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Marks the beginning of the error handling section of the ERROR HANDLER structure. The HANDLER section of the structure is only executed when an error occurs within the TRAPped section of the structure. Before executing the HANDLER section, ERR is assigned the error number, ERRFILE is assigned the file number in use if it was an input/output error, and ERRTEXT$ is assigned the text of the error message. For more information on how the ERROR HANDLER structure works see Appendix A.

NOTE

HANDLER is not supported by version 0.14.

SYNTAX

```
HANDLER
```

EXAMPLE

```
HANDLER
```

SAMPLE PROGRAM

```
// function for calculation of factorial
FUNC factorial(number) CLOSED
  TRAP // catch numbers too large overflow
    RETURN fac(number)
  HANDLER
    RETURN 1e+38 // a very large number
  ENDTRAP
  //
  FUNC fac(number)
    IF number=0 THEN
      RETURN 1
    ELSE
      RETURN fac(number-1)*number // recursive call
    ENDIF
  ENDFUNC fac
ENDFUNC factorial

SCAN      system now knows this function from direct mode
PRINT FACTORIAL(3)
6
PRINT FACTORIAL(99)
1e+38
```

ADDITIONAL SAMPLES SEE: ENDTRAP, ERR, ERRFILE, ERRTEXT$, REPORT
USED IN FUNCTION: FILE'EXISTS
SEE ALSO: ENDTRAP, ERR, ERRFILE, ERRTEXT$, TRAP, REPORT

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows conditional statement execution, depending on the value of the ⟨condition⟩. If the ⟨condition⟩ is TRUE (a value not equal to 0), the statements following the THEN are executed; otherwise they are skipped. For multiple conditions, IF can be combined with ELSE via the ELIF statement. A simple IF statement may be placed on one line and does not need the terminating ENDIF. For a more complete description of the IF structure, see Appendix A.

SYNTAX

```
      (One line IF)

IF <condition> THEN <statement>

      (Multi line IF)

IF <condition> [THEN]
  <statements>

 <condition> is a <numeric expression>
```

EXAMPLES

```
IF guess$="END" THEN
IF num=0 THEN
IF more THEN EXEC fetch
```

SAMPLE PROGRAM

```
total:=0;count:=0
REPEAT
  INPUT "NUMBER (0 TO STOP): " : number;
  IF number>0 THEN
    count:+1;total:+number          increment count & total
    PRINT "THAT MAKES THE TOTAL";total
  ELSE
    PRINT "OK - NO MORE NUMBERS"
  ENDIF
UNTIL number=0
avg:=total DIV count
PRINT "THE TOTAL FOR";count;"NUMBERS WAS";total
PRINT "THAT MEANS THE AVERAGE WAS";
IF total/count <> avg THEN PRINT "ABOUT";
PRINT avg

RUN
NUMBER (0 TO STOP): 25 THAT MAKES THE TOTAL 25
NUMBER (0 TO STOP): 246 THAT MAKES THE TOTAL 271
NUMBER (0 TO STOP): 1 THAT MAKES THE TOTAL 272
NUMBER (0 TO STOP): 0 OK - NO MORE NUMBERS
THE TOTAL FOR 3 NUMBERS WAS 272
THAT MEANS THE AVERAGE WAS ABOUT 90
```

ADDITIONAL SAMPLES SEE: DIV, ELIF, ELSE, ENDIF, ENDWHILE, EXP, MOD, SGN, THEN
USED IN PROCEDURES: FETCH, LOWER'TO'UPPER, SCANKEY
SEE ALSO: ELIF, ELSE, ENDIF, THEN

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [-] VERS 2.00 [*]

Allows a closed procedure or function to use variables, arrays, procedures, and functions from outside the procedure or function. The word IMPORT is used since the latter are imported into a closed procedure or function from the outside program. (they are global for this particular procedure or function). You may have more than one IMPORT statement inside a procedure or function.

NOTES

(1) IMPORT is not supported by version 0.14.
(2) IMPORT cannot be used in procedures or functions that are not declared CLOSED.
(3) IMPORT can occur anywhere within the procedure or function; however, to keep the program readable, it should occur on the first line, immediately after the heading statement.
(4) ZONE is automatically shared with a CLOSED procedure or function without the need of IMPORT.
(5) When specifying an array in an IMPORT statement, simply use the array's name without the parentheses or commas. This differs from an alias array via the REF parameter in the PROC or FUNC statement.
(6) Any procedure or function used by a closed procedure in version 2.00 must be declared global by using the procedure name (the ⟨identifier⟩) in an IMPORT statement. In version 0.14, all procedures and functions are recognized in a closed procedure without the need for an IMPORT statement.
(7) IMPORT is not allowed in an external procedure or function.
(8) See Appendix A for a description of the PROCEDURE structure.

SYNTAX

```
IMPORT <identifier>{,<identifier>}

 <identifier> is the name of a
   variable, array, function, or procedure
```

EXAMPLES

```
IMPORT name$,city
IMPORT price
```

SAMPLE PROGRAM

```
DIM name$ OF 40, my'name$ OF 40
my'name$:="COMMODORE"
INPUT "WHAT IS YOUR NAME: " : name$
EXEC printout(my'name$)
PRINT "NOW YOUR NAME IS";name$
PRINT "BUT MY NAME IS STILL";my'name$
PRINT "ALL DONE"
//
PROC printout(my'name$) CLOSED
  IMPORT name$
  PRINT "THIS IS JUST AN EXAMPLE";name$
  PRINT "SO LETS CHANGE YOUR NAME"
  name$:="MUD"; my'name$:="COMPUTER"
ENDPROC printout

RUN
WHAT IS YOUR NAME: SIMON
THIS IS JUST AN EXAMPLE SIMON
SO LETS CHANGE YOUR NAME
NOW YOUR NAME IS MUD
BUT MY NAME IS STILL COMMODORE
ALL DONE
```

SEE ALSO: CLOSED, ENDFUNC, ENDPROC, EXEC, FUNC, PROC, REF

CATEGORY: Operator
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Locates the position of ⟨string 1⟩ within ⟨string 2⟩ and returns the position (an integer) of the first character of ⟨string 1⟩ within ⟨string 2⟩. If ⟨string 1⟩ is not found within ⟨string 2⟩, it returns 0. If the length of ⟨string 1⟩ is 0 (i.e., is the null string ("")) then the value returned will be the length of ⟨string 2⟩ plus 1 (i.e., LEN (⟨string 2⟩)+1). See Appendix B for more information about string handling.

SYNTAX

```
<string1> IN <string2>

 <string1> and <string2> are <string expressions>
```

EXAMPLES

```
"NO" IN valid'answers$
item$ IN choices$
IF reply$ IN "YNyn" AND reply$>"" THEN PRINT "Good."
```

SAMPLE PROGRAM

```
DIM hit$ OF 5, letter$ OF 1
count:=0
a:=ORD("A"); z:=ORD("Z")
PRINT "I WILL RANDOMLY PICK AND PRINT LETTERS"
PRINT "UNTIL I HIT ONE OF 5 LETTERS YOU CHOOSE."
REPEAT
  INPUT "ENTER YOUR 5 HIT LETTERS: " : hit$
UNTIL len(hit$)=5
REPEAT
  letter$:=CHR$(RND(a,z))
  PRINT letter$,
  count:+1
UNTIL letter$ IN hit$
PRINT             provide a carriage return
PRINT "A HIT --- AFTER";count;"LETTERS"

RUN
I WILL RANDOMLY PICK AND PRINT LETTERS
UNTIL I HIT ONE OF 5 LETTERS YOU CHOOSE.
ENTER YOUR 5 HIT LETTERS: QTSPE
ARWBNXYLS
A HIT --- AFTER 9 LETTERS
```

ADDITIONAL SAMPLES SEE: AND, CLOSE, COPY, ELIF, KEY$, THEN
USED IN PROCEDURES: GET'VALID, LOWER'TO'UPPER
SEE ALSO: CHR$, ORD, STR$

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows the user to enter data into a running program from a sequential file that was created by Commodore BASIC or by COMAL PRINT FILE statements. The INPUT values will be assigned to the specified INPUT variables. Multiple variables can be part of one INPUT line. The file must have been opened previously as a READ type file. INPUT FILE may also be used to read characters directly off the screen. In version 0.14 the file must be opened with a UNIT of 3. Example: OPEN FILE 7,"",UNIT 3,READ; or in version 2.00: OPEN FILE 7,"DS:",READ.

NOTES

(1) INPUT FILE is used to read sequential files created by PRINT FILE, LIST, DISPLAY, or SELECT OUTPUT or Commodore BASIC files created with PRINT#. Version 0.14 can input up to 120 characters at a time. Version 2.00 does not have this limitation.
(2) INPUT FILE cannot be used to read files created by WRITE FILE since a write file is in binary while INPUT requires an ASCII file (as created by PRINT).
(3) INPUT FILE can read characters directly off the screen, a line at a time.
(4) INPUT FILE can be used to read, one line at a time, a program that was LISTed to tape or disk.
(5) See Appendix C for more information about sequential files.

SYNTAX

```
INPUT FILE <filenum>: <variable list>

 <variable list> is one or more <variable names> to be used
   for the operation separated by commas
```

EXAMPLES

```
INPUT FILE 2: number
INPUT FILE infile : customer$
INPUT FILE 3: customer'numb,zip,rate
```

SAMPLE PROGRAM

```
DIM text$ OF 100
OPEN FILE 2,"VISITOR'INPUT",READ
WHILE NOT EOF(2) DO
  INPUT FILE 2 : text$
  PRINT text$
ENDWHILE
CLOSE
PRINT "ALL DONE"

RUN
  (file 2 named VISITOR'INPUT is opened for input)
COMAL Users Group
ALL DONE
```

SEE ALSO: FILE, GET$, OPEN, PRINT, READ

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows the user to enter data into a running program from a random access file that was created by Commodore BASIC or by COMAL PRINT FILE statements. The INPUT values will be assigned to the specified INPUT variables. Multiple variables can be part of one INPUT line. The file must have been opened previously as a RANDOM type file.

NOTES

(1) INPUT FILE is used to read random files created by PRINT FILE or Commodore BASIC files created with PRINT#.
(2) INPUT FILE is not compatible with READ FILE since it expects an ASCII file, not a binary file.
(3) If you attempt to INPUT from a record not yet containing information via a previous PRINT FILE statement, you will get either garbage (nonsense) or a possible run time error.
(4) If you attempt to INPUT from a record past the last created record, you get a disk error.
(5) In version 2.00 if you don't specify a record number, COMAL will give you the next field, as if the file were a sequential file (COMAL can treat a random file as a sequential file of a fixed length).

SYNTAX

```
INPUT FILE <file#>[,<record#>[,<offset>]]: <variable list>

 <record#> is a positive <numeric expression>
 <offset> is a positive <numeric expression>;
   if omitted, no bytes will be skipped
 <variable list> is one or more <variable names> to be used
   for the operation separated by commas
```

EXAMPLES

```
INPUT FILE 2,rec : number
INPUT FILE infile,inrec : customer$
INPUT FILE 3,5 : customer'numb,zip,rate
```

SAMPLE PROGRAM

```
DIM text$ OF 80
OPEN FILE 2,"RANDOM'INPUT",RANDOM 80
PRINT "READ SOME RANDOM TEXT RECORDS"
REPEAT
  INPUT "WHAT RECORD NUMBER (0 TO STOP): " : number
  IF number THEN
    INPUT FILE 2,number: text$
    PRINT text$
  ENDIF
UNTIL number=0
CLOSE
PRINT "ALL DONE"

RUN
  (file 2 named RANDOM'INPUT is opened for random access)
READ SOME RANDOM TEXT RECORDS
WHAT RECORD NUMBER (0 TO STOP): 5
  (record number 5 is read from file 2)
THIS IS THE FIFTH RECORD
WHAT RECORD NUMBER (0 TO STOP): 9
  (record number 9 is read from file 2)
THIS IS THE NINTH RECORD
WHAT RECORD NUMBER (0 TO STOP): 0
  (file 2 is closed)
ALL DONE
```

SEE ALSO: FILE, GET$, OPEN, PRINT, READ, UNIT

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Allows the user to enter data into a running program from the keyboard that will be assigned to the specified INPUT variables. A comma is accepted as a delimiter following numeric data being input from the keyboard. The COMAL string INPUT accepts every character typed, and can only be terminated with a carriage return. It will accept commas, semicolons, colons, and quote marks. Multiple variables can be part of a single INPUT statement. Commas are used to separate several ⟨numeric variables⟩ within one INPUT statement. You may not have more than one string variable in a single INPUT statement, and it must be the last variable in the list. If a second string variable is included it is ignored by the COMAL interpreter. The prompt for the INPUT may be a string expression. If no prompt is provided, a question mark is printed as the prompt. No question mark is printed if you have supplied a prompt. A blinking cursor is provided after the prompt to signal the user that input is expected. If the INPUT statement is ended with a semicolon (or comma in version 2.00), no line feed or carriage return is provided when the user hits the RETURN key after his or her input.

NOTES

(1) The STOP key can be used even during an INPUT request.
(2) Version 0.14 does not allow a comma to be used as the ⟨mark⟩.
(3) Version 0.14 does not support the AT option.
(4) Version 2.00 allows hexadecimal and binary numbers to be typed as input to a numeric input request (precede the hex number with a $ and the binary number with a %). See Appendix M for more information.
(5) Version 2.00 has a "protected" input field. It does not allow cursor up or down or reverse on or off. In addition, it will not allow you to go into "quote" mode or insert mode. Finally, HOME CURSOR has been redefined during input to mean "go to beginning of the input field." CLEAR SCREEN has been redefined to mean "clear the input typed in thus far and then go to the beginning of the input field."
(6) If an input field ⟨length⟩ of 0 is used, only a carriage return is accepted.
(7) INPUT of null (just a carriage return) is acceptable.
(8) The input prompt cannot be longer than 39 characters on 40-column computers (PET 4032 and Commodore 64) due to a bug in the computer operating system.

SYNTAX

```
INPUT [AT <row>,<col>[,<len>]:][<prompt>:<varlist>[<mark>]

 <row> is a <numeric expression> whose value is 0-25
   (0 means current row)
 <col> is a <numeric expression> whose value is 0-80
   (or on 40 column screens, 0-40)
   (0 means current column)
 <len> is a <numeric expression> whose value is 0-80
   or on 40 column screens, 0-40
   if omitted, the rest of the line is used
 <prompt> is a <string expression>;
   if omitted, a question mark will be supplied by system
 <varlist> is one or more <variable names> to be used
   for the operation separated by commas
 <mark> is a semicolon or a comma;
   if a semicolon is used, a space will be issued after the
    users input instead of a carriage return
   if comma is used, spaces will be issued up to next zone
    instead of a carriage return
```

EXAMPLES

```
INPUT guess
INPUT "WHAT IS YOUR NAME:" : name$
INPUT prompt$ : answer;
INPUT "COORDINATES: " : x,y,z
INPUT "AGE, NAME: " : age, name$
INPUT AT 10,3: prompt$: answer
```

SAMPLE PROGRAM

```
DIM prompt$ OF 20, name$ OF 20
prompt$:="WHAT IS YOUR NAME: "
INPUT prompt$ : name$;
PRINT "THANK YOU."
PRINT "HELLO THERE,";name$

RUN
WHAT IS YOUR NAME: MUD THANK YOU.
HELLO THERE, MUD
```

ADDITIONAL SAMPLES SEE: EXTERNAL, INT, RND, SELECT OUTPUT, THEN
SEE ALSO: CURSOR, FILE, GET$, KEY$, READ

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the nearest integer that is less than or equal to the specified number. Both positive and negative numbers are "rounded down." For example, -8.3 becomes integer -9. When a real number is assigned to an integer variable, it is rounded to the nearest integer, different than the INT function. Thus to round a number to the nearest integer, you can use the ROUND function listed in Appendix D.

SYNTAX

```
INT (<numeric expression>)
```

EXAMPLES

```
INT(guess)
INT(5.65)
INT(num1/num2)
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "NUMBER PLEASE (0 TO STOP): " : number
  PRINT "INTEGER OF";number;"IS";INT(number)
UNTIL number=0
PRINT "ALL DONE"

RUN
NUMBER PLEASE (0 TO STOP): 5.3
INTEGER OF 5.3 IS 5
NUMBER PLEASE (0 TO STOP): 78.95
INTEGER OF 78.95 IS 78
NUMBER PLEASE (0 TO STOP): -3.2
INTEGER OF -3.2 IS -4
NUMBER PLEASE (0 TO STOP): 0
INTEGER OF 0 IS 0
ALL DONE
```

USED IN FUNCTION: ROUND
SEE ALSO: ABS, SGN, VAL

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Interfaces with interrupt requests from the IEEE-488 bus. It makes sure that a call of ⟨procedure name⟩ is made if a signal is sensed on the SRQ line on the IEEE-488 bus.

NOTES

(1) INTERRUPT is not supported by version 0.14.
(2) INTERRUPT without a procedure name following it turns off any previously set INTERRUPT.
(3) Only one INTERRUPT may be set at one time.

SYNTAX

```
INTERRUPT [<procedure name>]

 <procedure name> is an <identifier>;
   if omitted, any previous INTERRUPT is disabled
```

EXAMPLES

```
INTERRUPT flasher
INTERRUPT attention
INTERRUPT                turn off interrupt monitoring
```

SAMPLE PROGRAM

```
 INTERRUPT                clear pending interrupts
 PAGE
 PRINT "MONITORING THE IEEE 488 BUS FOR AN INTERRUPT"
 PRINT "TIME IN JIFFIES:"
 INTERRUPT blinker        turn on the interrupt
 LOOP
   PRINT AT 2,18: TIME,SPC$(7)     print number of jiffies
 ENDLOOP                   forever
 //
 PROC blinker
   REPEAT
     CURSOR 12,20                              position cursor
     IF RND(1,2)=1 THEN PRINT "[RVS]",         random reverse on
     PRINT "INTERRUPT REQUEST - HIT SHIFT WHEN READY"
     FOR pause:=1 TO 99 DO NULL // PAUSE
   UNTIL PEEK(152) // c64 use 653 ---           wait for SHIFT
   INTERRUPT                                   disable interrupt
   PRINT AT 12,20: "-INTERRUPT NOW ACKNOWLEDGED AND
DISABLED-"
 ENDPROC blinker                               returns to
previous place

 RUN
   (screen clears)
 MONITORING THE IEEE 488 BUS FOR AN INTERRUPT
 TIME IN JIFFIES: 458924
   (time is continually updated until a signal is sensed on
   SRQ line. INTERRUPT REQUEST then blinks in screen center)
   (hit SHIFT to stop - and it returns to printing the time)
```

SEE ALSO: PROC

CATEGORY: Function
KERNAL: [NO] VERS 0.14 [+] VERS 2.00 [*]

Scans the keyboard buffer once and returns the last key typed. If no key has been pressed, it returns a CHR$(0). If you want to actually wait for a key to be hit, you should use GET$.

NOTE

KEY$ is not supported by PET/CBM version 0.14; however, procedures GET'CHAR and SCANKEY (listed in Appendix D) can be used as viable substitutes.

SYNTAX

```
KEY$
```

EXAMPLES

```
code$:=KEY$
PRINT KEY$
temp:=ORD(KEY$)
```

SAMPLE PROGRAM

```
TRAP ESC-                          disable the STOP key
DIM choice$ OF 1
PRINT "PLEASE ENTER A VOWEL:";
REPEAT
  choice$:=KEY$                 get the key hit
UNTIL choice$ IN "AEIOU"        repeat looking at keyboard
PRINT choice$                   until A E I O or U is hit
PRINT "ALL DONE"
TRAP ESC+                       enable the STOP key

RUN
PLEASE ENTER A VOWEL: E           hit keys other than a vowel
ALL DONE                            and nothing happens
```

ADDITIONAL SAMPLE SEE: CURSOR
USED IN PROCEDURE: GET'CHAR
SEE ALSO: CURSOR, GET$, INPUT, READ

CATEGORY: Identifier
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Assigns a label name to the line. LABEL is an implied KEYWORD, and does not need to be typed. It will be ignored if included. This label can only be referenced by a GOTO (or by a RESTORE in version 2.00) and under normal situations is rarely used. A label statement is nonexecutable and may be placed anywhere within the program as a one-line statement. Thus it may be used to identify program sections similar to a remark.

NOTE

(1) ⟨label name⟩ is not indented if it occurs within an indented block of statements.
(2) ⟨label name⟩ can be used as a reference for a RESTORE statement in version 2.00.
(3) A label: inside any procedure is considered local to that procedure in version 2.00. In version 0.14 it is always considered global.

SYNTAX

```
<label name>:

 <label name> is an <identifier>
```

EXAMPLES

```
response:
quick'quit:
```

SAMPLE PROGRAM (version 2.00 only)

```
DIM weekdays$(7) OF 10
DATA "This data line will not be used"
days:
DATA "Sunday","Monday","Tuesday","Wednesday"
DATA "Thursday","Friday","Saturday"
RESTORE days
FOR temp:=1 TO 7 DO READ weekdays$(temp)
PRINT "Data all read"
FOR temp:=1 TO 7 DO PRINT weekdays$(temp)
PRINT "ALL DONE"

RUN
Data all read
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
ALL DONE
```

ADDITIONAL SAMPLE SEE: DATA
SEE ALSO: RESTORE

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the length of the specified string. All characters, even spaces and non-printing characters, are counted. A string constant must be enclosed in quotes ("). The length of a null string ("") is 0. See Appendix B for more information on string handling.

SYNTAX

```
LEN(<string expression>)
```

EXAMPLES

```
LEN(text$)
LEN("TESTING")
LEN(item$+location$)
```

SAMPLE PROGRAM

```
DIM text$ OF 80
REPEAT
  INPUT "TYPE SOMETHING: " : text$
  IF text$<>"" THEN PRINT text$;"IS";LEN(text$);"LONG"
UNTIL text$=""
PRINT "ALL DONE"

RUN
TYPE SOMETHING: LEN LINDSAY LIKES THIS KEYWORD
LEN LINDSAY LIKES THIS KEYWORD IS 30 LONG
TYPE SOMETHING: IS YOUR FIRST NAME A KEYWORD
IS YOUR FIRST NAME A KEYWORD IS 28 LONG
TYPE SOMETHING:          here just hit [RETURN] with no input
ALL DONE
```

ADDITIONAL SAMPLES SEE: IN, NULL, ORD, PROC
USED IN PROCEDURES: LOWER'TO'UPPER, VALUE
SEE ALSO: CHR$, ORD, SPC$, STR$

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

LET is an implied KEYWORD. You never need to use it explicitly in your programs, and if you do COMAL will just ignore it. But the concept of LET is one of assignment; that is, a value is assigned to a variable. In a COMAL program listing, := is used instead of = to show assignment rather than comparison. If you use a plain = for assignment, COMAL will convert it to := for you. Multiple assignments may be placed on one line separated by a semicolon (;). A :+ is used for incremental assignment (to add 5 to the variable TOTAL you would code: TOTAL:+5), and :- is used for decremental assignment (to subtract your lost BET from your MONEY you would code: MONEY:-BET). Partial strings (substrings) can be assigned without disturbing the rest of the string.

NOTES

(1) String concatenation using :+ is not supported by version 0.14.
(2) The KEYWORD LET is not listed in a program listing.

SYNTAX

```
       (direct assignment)

<variable or array element> := <value>

  <value> is an <expression> of the same type (string or
    numeric) as the <variable or array element>

       (partial string assignment)

<string variable>(<substring>):=<string expression>
       or
<string array>(<index>)(<substring>):=<string expression>

  <string array> is the string array name
  <index> specifies which element in the array:
    <index number>{,<next index number>}
  <substring> specifies which part of the string:
    <start>[:<end>]
      <start> is the position in the string where the
        substring will begin
      <end> is the position in the string where the
        substring will end
        if omitted only the <start> character is used

       (incremental assignment)

<numeric variable name> :+ <numeric expression>

       (string concatenation)

<string1> :+ <string expression>      version 2.00 only
       or
<string1> := <string1>+<string expression>

  <string1> is a string variable or string array element

       (decremental assignment)

<numeric variable or array element> :- <numeric expression>
```

EXAMPLES

```
count:=5;total:=0;temp$:=""
text$:="TEST"
count:+1                    increment by 1
total:-15                   subtract 15 from TOTAL
reply$:=reply$+mark$        concatenate REPLY$ and MARK$
reply$:+mark$               as above (ver 2.00 only)
text$(1:3):="XYZ"           characters 1, 2, and 3 of TEXT$
                                 are changed to "XYZ"
name$(2):="COMPUTER"        second element in array is set to
                                 equal "COMPUTER"
```

SAMPLE EXERCISE

```
10 LET count=0
20 INPUT "WHAT SHOULD COUNT EQUAL: " : count
30 PRINT "COUNT IS NOW";count
40 count:+2
50 PRINT "ADDING 2 MAKES";count

LIST
0010 count:=0              the = is converted to :=
0020 INPUT "WHAT SHOULD COUNT EQUAL: ": count
0030 PRINT "COUNT IS NOW";count
0040 count:+2               note this is adding 2 to COUNT
0050 PRINT "ADDING 2 MAKES";count

RUN
WHAT SHOULD COUNT EQUAL: 4
COUNT IS NOW 4
ADDING 2 MAKES 6
```

ADDITIONAL SAMPLES SEE: DIM (numeric arrays), ENDWHILE, FOR
USED IN PROCEDURES: MOST PROCEDURES INCLUDE ASSIGNMENT (LET)
SEE ALSO: DIM, PROC, REF

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

LOADs an assembler object file (hex file) containing machine language packages. The machine code is automatically protected and cannot be destroyed or overwritten by other LINK commands. A SAVE command will save both the program and the machine code as one file. Then any future LOAD will LOAD both the program and machine code automatically. See Appendices K and P for more information about machine language and LINK.

NOTES

(1) LINK is not supported by version 0.14.
(2) After a LINK is executed, all names in the current program become undeclared. A SCAN or RUN will once again set up the name table.

SYNTAX

```
LINK <filename>
```

EXAMPLES

```
LINK "MY'SYSTEM.OBJ"
LINK "ROUTINE.OBJ"
```

SAMPLE EXERCISE

```
USE bitpac // user supplied machine language package
INPUT "Number please: ": number
PRINT "Your number OR 128 is:";lor(number,128)
PRINT "All done"
SIZE
prog  data  free
00073 00027 38292
LINK "bitpac.p"
SIZE
prog  data  free
00073 00027 34196
RUN
Number please: 20
Your number OR 128 is: 148
All done
```

ADDITIONAL SAMPLE SEE: DISCARD

SEE ALSO: USE

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

LISTs the specified program lines. If no specific lines are indicated, all lines are listed. You may specify a procedure or function name instead of the specific line number range. To store program lines onto disk or tape in ASCII format simply include a ⟨filename⟩. To avoid confusion with files SAVEd to disk, you should end the ⟨filename⟩ with .L when LISTing lines to disk. A program LISTed to disk or tape can be retrieved later via the ENTER command. To LIST lines to your printer, issue the command LIST "LP:" or SELECT OUTPUT "LP:" just prior to your LIST command (this selects the Line Printer). COMAL structures are indented as shown in the sample programs in this Handbook. If program lines are broken into two or more screen lines using the LIST command, use EDIT instead of LIST. Output is returned to the screen after a LIST command (LIST performs a SELECT OUTPUT "DS:" when finished).

NOTES

(1) To slow down the listing speed on the screen, hold down the left arrow key (CMB 8000 series), RVS key (CBM/PET 4000, 3000, and 2000 series), or CTRL key (CBM 64).
(2) In version 2.00, hit SPACE to pause a listing. Hit SPACE again to resume.
(3) LIST to tape is not supported by version 0.14.
(4) A program LISTed to disk or tape can later be read with INPUT FILE statements.
(5) Use EDIT instead of LIST if you do not want the indentation of structures.
(6) A program LISTed to disk is a SEQ (sequential) ASCII type file.
(7) Valid line numbers for a COMAL program are from 1 through 9999.
(8) To list a procedure or function simply type LIST ⟨procedure or function name⟩. This is not supported by version 0.14.
(9) When LISTing a 2.00 program to disk to be able to transfer it to 0.14 make sure that you are in the all unshifted (lower case) mode. Issue the following poke before LISTing to disk: POKE $24B,PEEK($24B) BITAND %00111111.

IMPORTANT NOTE

LIST to tape or disk is very useful. If you LIST a program to tape or disk, type NEW, and then ENTER it back again, you will clean up the COMAL NAME TABLE as well as get more free memory. SAVE and LOAD keep the old NAME table. Each time you make a spelling mistake (such as LIT instead of LIST) that name goes into the table. Version 0.14 has only 255 names allowed, so if you make alot of spelling mistakes, you may get a TOO MANY NAMES error. Then you must perform this name table cleanup operation.

SYNTAX

```
LIST [<range>][[<specifier>]<filename>]

  <range> is represented by
    <procname> or
    <funcname> or
    <line range> represented by
      [<start line>][-][<end line>]
        <start line> is a number from 1-9999
          and is less than <end line>
          if omitted, line 1 is used as the start line
        <end line> is a number from 1-9999
          and is greater than <start line>
          if omitted, line 9999 is used
  <specifier> is  represented by
    either a space or a comma in version 0.14
    either a space or the keyword TO in version 2.00
  <filename> is optional and may not be a variable
```

EXAMPLES

```
COMMAND                          RESULT
LIST                          lists all lines.
LIST -500                     lists lines 0-500.
LIST 9000-                    lists lines 9000-9999.
LIST 300-400                  lists lines 300-400.
LIST 9000-,"CHECK'INPUT.L"    lists lines 9000-9999 to disk.
LIST sort TO "QUICK.L"        lists procedure named SORT
                                   to disk file "QUICK.L"
```

SAMPLE EXERCISE

```
10 FOR temp:=1 TO 20
30 ENDFOR temp           note that the lines may be entered
20 PRINT temp;           in any order you wish

LIST
0010 FOR temp:=1 TO 20 DO
0020   PRINT temp;             indentation provided by COMAL
0030 ENDFOR temp

RUN
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

ADDITIONAL SAMPLES SEE: EDIT, END, ENTER, LET
SEE ALSO: AUTO, DEL, DISPLAY, EDIT, ENTER, RENUM

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

LOADs a program from disk or tape into the computer's memory. Any program currently in the computer is deleted first, and all variables are cleared. The program being LOADed must have previously been SAVEd to disk or tape via the SAVE command (not via a LIST to disk or tape).

NOTES

(1) LOAD and ENTER commands are not compatible.
(2) LOAD from tape is not supported by version 0.14.
(3) You cannot LOAD a program SAVEd by a different COMAL version. You should use LIST and ENTER to transfer the program from one version to the other.
(4) LOAD can load a COMAL program combined with a machine language package. It will automatically locate the machine language correctly.

SYNTAX

```
LOAD <program name>

 <program name> is a <filename>
   it cannot be a variable name
```

EXAMPLES

```
LOAD "PROCESS"
LOAD "0:SPECIALS"
```

SAMPLE EXERCISE

```
10 PRINT "THIS IS THE FIRST PROGRAM"

LIST
0010 PRINT "THIS IS THE FIRST PROGRAM"

LOAD "ANOTHER"

LIST
0200 PRINT "THIS IS ANOTHER PROGRAM"
0210 PRINT "IT JUST CAME FROM DISK"

RUN
THIS IS ANOTHER PROGRAM
IT JUST CAME FROM DISK
```

SEE ALSO: ENTER, LIST, SAVE, VERIFY

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the natural logarithm of the number supplied. This is log to the base *e*. A good representation of *e* is 2.718281828. The number supplied must be positive or an error will result. Three conversion formulae are:

```
LOG(X)/LOG(10) results in log to the base 10.
natural log * .434295 = common log
common log * 2.3026 = natural log
```

SYNTAX

```
LOG(<positive number>)

 <positive number> is a <numeric expression>
   it must have a value greater than 0
```

EXAMPLES

```
LOG(numb*test)
LOG(39)
LOG(ABS(entry))
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "NUMBER (0 TO STOP): " : number
  IF number>0 THEN PRINT "LOG OF";number;"IS";LOG(number)
UNTIL number=0
PRINT "ALL DONE"

RUN
NUMBER (0 TO STOP): 5
LOG OF 5 IS 1.60943791
NUMBER (0 TO STOP): 25
LOG OF 25 IS 3.21887582
NUMBER (0 TO STOP): 0
ALL DONE
```

SEE ALSO: ATN, COS, EXP, SIN, TAN

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Provides a loop structure with one exit condition inside the body of the structure. When an EXIT statement is executed, program execution continues with the statement following the ENDLOOP statement. A loop structure should only have one exit to be consistent with structured programming. If there are no loop statements prior your EXIT statement, you should be using a WHILE loop. If there are no loop statements after your EXIT statement, you should be using a REPEAT loop. See Appendix D for more information on the loop structure.

NOTE

LOOP is not supported by version 0.14.

SYNTAX

```
LOOP
```

EXAMPLE

```
LOOP
```

SAMPLE PROGRAM

```
DIM temp$ OF 80, array$(100) OF 80
OPEN FILE 2,"TEST'LOOP",READ
pointer:=0
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: temp$
  EXIT WHEN temp$="*END*"
  pointer:+1
  array$(pointer):=temp$
ENDLOOP
CLOSE FILE 2
FOR test:=1 TO pointer DO PRINT array$(test)
PRINT "ALL DONE"

RUN
THIS IS A SILLY LOOP
THIS IS A SILLY LOOP
THIS IS A SILLY LOOP
RECORD ONE
RECORD TWO
RECORD THREE
ALL DONE
```

ADDITIONAL SAMPLES SEE: ENDLOOP, ERR, EXIT, EXTERNAL
SEE ALSO: ENDLOOP, EXIT

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

MAIN is needed only when using external procedures or functions. If the program is stopped for any reason while executing the external procedure or function, COMAL will consider that external section as a separate "program." You then can locally LIST, EDIT, FIND, SAVE, etc. this external section. Then to get back to the main program simply issue the command MAIN. Once you return to the main program, the external section is inaccessible.

SYNTAX

```
MAIN
```

EXAMPLE

```
MAIN
```

SAMPLE EXERCISE

```
AUTO          first write a procedure to store externally
0010 PROC pause CLOSED
0020   FOR delay:=1 TO 99999 DO NULL
0030 ENDPROC pause

SAVE "PAUSE.E"    save it as external before testing
NEW               ready to write a new program
AUTO              write the test program next
0010 PRINT "Ready to test the pause external procedure"
0020 pause
0030 END "Done with test"
0040 PROC pause EXTERNAL "PAUSE.E"
0050              hit stop key here
RUN             try a sample run of the pause test
Ready to test the pause external procedure
                  hit stop key - pause is much too long
END IN 0020
LIST
0010 PROC pause CLOSED
0020   FOR delay:=1 TO 99999 DO NULL
0030 ENDPROC pause
EDIT 20           fix the length to just 999
0020 FOR delay:=1 TO 99999 DO NULL       change to 999
DISPLAY
PROC pause CLOSED
  FOR delay:=1 TO 999 DO NULL
ENDPROC pause
DELETE "PAUSE.E"         delete old version
SAVE "PAUSE.E"          save version 2 of pause as external
MAIN                    we forgot to SAVE this test program
LIST
0010 PRINT "Ready to test the pause external procedure"
0020 pause
0030 END "Done with test"
0040 PROC pause EXTERNAL "PAUSE.E"
SAVE "TEST'PAUSE"        save our test program
RUN                     try it again
Ready to test the pause external procedure
Done with the test
```

SEE ALSO: EXTERNAL

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Merges, without regard for line numbers, into the computer's memory a program that was stored on disk in ASCII form via a LIST command. The program segment is placed immediately after the last line in the current program unless you specify otherwise. This capability allows easy merging of procedures from a "procedure library." Lines are entered into the program as if they were typed in on the keyboard. To MERGE a segment into a specific place in your existing program, first use RENUM, specifying the starting line number as the line just after the spot where you wish to open the "hole" for the new segment: RENUM 300;9000,1. Then MERGE the new segment specifying the same starting line number: MERGE 300 "MY'PROC.L".

NOTES

(1) LOAD and MERGE commands are not compatible.
(2) To "transfer" a COMAL program from one version to another, first LIST it to disk or tape, then MERGE or ENTER it into the other version.
(3) MERGE is not supported by version 0.14; use ENTER instead.
(4) Version 2.00 has ENTER perform an automatic NEW first, and uses MERGE to merge sections, without regard to line numbers in the file being merged from disk or tape.
(5) When MERGEing a program from another version, it is possible that some lines may yield SYNTAX errors. If this happens the system will list the line in question with the error message below it. Either correct the line or make the line a remark by inserting a ! or // just after the line number and hit ⟨return⟩. The system will continue to enter the remainder of the program. When it is finished, you then can go back to the problem lines and correct them to match your version.

SYNTAX

```
MERGE [<target startline>][,<increment>] <filename>

 <target startline> is a <numeric expression>
   whose value is 1-9999; if omitted, the last line number
   plus the increment is used
 <increment> is a <numeric expression>
   whose value is 1-9999; if omitted, 10 is used
 <filename> cannot be a variable
```

EXAMPLES

```
MERGE "GET'CHAR.L"
MERGE 300 "INPROC.L"
```

SAMPLE EXERCISE

```
AUTO
0010 clear'screen
0020 PRINT "This is only the beginning"
0030 PROC clear'screen
0040 PRINT CHR$(147),
0050 ENDPROC clear'screen
0060                                  hit the STOP key here
RENUM 30;1000,5
LIST
0010 clear'screen
0020 PRINT "This is only the beginning"
1000 PROC clear'screen
1005   PRINT CHR$(147),
1010 ENDPROC clear'screen
MERGE 30,1 "SAMPLE.L"
LIST
0010 clear'screen
0020 PRINT "This is only the beginning"
0030 PROC sample
0031   PRINT "This is only the sample"
0032 ENDPROC sample
1000 PROC clear'screen
1005   PRINT CHR$(147),
1010 ENDPROC clear'screen
RENUM
LIST
0010 clear'screen
0020 PRINT "This is only the beginning"
0030 PROC sample
0040   PRINT "This is only the sample"
0050 ENDPROC sample
0060 PROC clear'screen
0070   PRINT CHR$(147),
0080 ENDPROC clear'screen
```

SEE ALSO: ENTER, LIST, LOAD, SAVE

CATEGORY: Operator
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the arithmetic remainder of a division. This means that A MOD B is the same as A-INT(A/B)*B. May be used with a division problem in combination with DIV to produce an integer answer with a remainder.

NOTE

Since you cannot divide by 0, the ⟨divisor⟩ may not have a value of 0.

SYNTAX

```
<dividend> MOD <divisor>

 <dividend> is a <numeric expression>
 <divisor> is a <numeric expression>
```

EXAMPLES

```
turn MOD count
score MOD 3
```

SAMPLE PROGRAM

```
DIM ending$ OF 3
REPEAT
  INPUT "INTEGER (0 TO STOP): " : n1
  IF n1 THEN
    INPUT "INTEGER: " : n2
    ones:=n2 MOD 10 // GET ONES DIGIT
    answer:=n1 DIV n2
    remainder:=n1 MOD n2
    IF n2=11 OR n2=12 OR n2=13 THEN
      ending$:="TH"
    ELSE
      CASE ones OF
      WHEN 0,4,5,6,7,8,9
        ending$:="TH"
      WHEN 1
        ending$:="ST"
      WHEN 2
        ending$:="ND"
      WHEN 3
        ending$:="RD"
      ENDCASE
    ENDIF
    IF remainder<>1 THEN ending$:=ending$+"S" // PLURAL
    PRINT n1;"DIVIDED BY";n2;"IS";answer;
    IF remainder THEN PRINT "AND";remainder,"/",n2,ending$;
    PRINT // CARRIAGE RETURN
  ENDIF
UNTIL n1=0
PRINT "ALL DONE"
```

```
RUN
INTEGER (0 TO STOP): 5
INTEGER: 3
5 DIVIDED BY 3 IS 1 AND 2/3RDS
INTEGER (0 TO STOP): 25
INTEGER: 6
25 DIVIDED BY 6 IS 4 AND 1/6TH
INTEGER (0 TO STOP): 0
ALL DONE
```

ADDITIONAL SAMPLES SEE: DIV, FUNC
USED IN FUNCTION: EVEN
SEE ALSO: DIV, INT

CATEGORY: Command / Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Initializes the disk in the specified drive. This is useful when you remove a disk and replace it with another one. This avoids any problems due to disks with the same ID number.

NOTES

(1) MOUNT is not supported by version 0.14. It may be simulated for drives 0: and 1: by the procedure MOUNT listed in Appendix D.
(2) MOUNT "cs:" is allowed by COMAL, but is not implemented in CBM COMAL, since only disk drives need initializing.

SYNTAX

```
MOUNT [<disk drive>]

 <disk drive> is <string expression> consisting of
   the drive number (0-15) followed by a colon
```

EXAMPLES

```
MOUNT "0:"              drive 0 of unit 8
MOUNT "1:"              drive 1 of unit 8
MOUNT                   current drive
MOUNT "3:"              drive 1 of unit 9
```

SAMPLE PROGRAM

```
DIM reply$ of 1
PAGE // clear screen
LOOP
  TRAP
    INPUT "Insert disk# 5 in drive 0 and hit return":reply$
    MOUNT "0:"
    EXIT // no error occurred
  HANDLER
    PRINT
    PRINT ERRTEXT$
  ENDTRAP
ENDLOOP
END "ALL DONE"

RUN                (no disk in drive 0 for first try)
  (screen clears)
Insert disk# 5 in drive 0 and hit return      hit return
21,READ ERROR,18,00     (now put in the disk)
Insert disk# 5 in drive 0 and hit return      hit return
ALL DONE
```

SEE ALSO: PASS

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Erases the program currently in the computer's memory and clears all variables. The COMAL interpreter itself is not erased. The system variables (i.e., ZONE) are reset to their default values.

NOTE

In version 2.00 NEW also erases all unprotected packages.

SYNTAX

```
NEW
```

EXAMPLE

```
NEW
```

SAMPLE EXERCISE

```
10 PRINT "THIS IS A NEW LINE"
20 PRINT "SO IS THIS ONE"
30 PRINT "TO ERASE THIS PROGRAM USE THE COMMAND:  NEW"

LIST
0010 PRINT "THIS IS A NEW LINE"
0020 PRINT "SO IS THIS ONE"
0030 PRINT "TO ERASE THIS PROGRAM USE THE COMMAND:  NEW"

NEW

LIST
  (nothing will appear since the program is erased)
```

ADDITIONAL SAMPLE SEE: MAIN
SEE ALSO: BASIC, DEL

CATEGORY: Operator
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Logical math operator reverses the TRUE/FALSE evaluation of the ⟨numeric expression⟩. If the ⟨numeric expression⟩ is TRUE (a value not equal to 0) the NOT becomes FALSE (a value of 0). If the ⟨numeric expression⟩ is FALSE (a value of 0), the NOT becomes TRUE (a value of 1). The second sample program listed below produces a chart that illustrates the effect of NOT.

SYNTAX

```
NOT <condition>
```

EXAMPLES

```
NOT temp
NOT (a AND b)
```

SAMPLE PROGRAM 1

```
done:=FALSE
total:=0
WHILE NOT done DO
  INPUT "NUMBER (0 TO STOP): " : number
  IF number=0 THEN done:=TRUE
  total:+number
ENDWHILE
PRINT "THE TOTAL IS";total

RUN
NUMBER (0 TO STOP): 5
NUMBER (0 TO STOP): 45
NUMBER (0 TO STOP): 23
NUMBER (0 TO STOP): 0
THE TOTAL IS 73
```

SAMPLE PROGRAM 2

```
DIM type$(FALSE:TRUE) OF 5
type$(FALSE):="FALSE";type$(TRUE):="TRUE"
ZONE 6
PRINT "NOT CHART"
PRINT "---------"
FOR a:=FALSE TO TRUE DO
  PRINT "A=";type$(a),"NOT A=";type$(NOT a)
ENDFOR a
ZONE 0

RUN
NOT CHART
---------
A= FALSE    NOT A= TRUE
A= TRUE     NOT A= FALSE
```

ADDITIONAL SAMPLES SEE: ENDWHILE, EOD, PEEK, REF, SPC$
SEE ALSO: AND, BITAND, BITOR, BITXOR, OR

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

This statement does nothing. It can be used when you need an empty statement in a structure, or to waste time in a pause loop, without affecting any of the programs variables.

SYNTAX

```
NULL
```

EXAMPLE

```
NULL
```

SAMPLE PROGRAM

```
DIM text$ OF 80
INPUT "MESSAGE: " : text$
FOR temp:=1 TO LEN(text$) DO
  PRINT text$(temp),
  EXEC pause(9)
ENDFOR temp
PRINT
PRINT "ALL DONE"
//
PROC pause(duration) CLOSED
  FOR temp:=1 TO 75*duration DO NULL
ENDPROC pause

RUN
MESSAGE: TESTING SLOW WRITING
  (each character is printed slowly on the screen)
TESTING SLOW WRITING
ALL DONE
```

ADDITIONAL SAMPLE SEE: INTERRUPT
SEE ALSO: TIME

CATEGORY: Special
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

OF is part of the CASE structure as well as part of the DIM statement when dimensioning space for strings or string arrays. For more information on the CASE structure see CASE and Appendix A. For more information on the DIM statement see DIM (strings) or DIM (string arrays).

SYNTAX

```
      (CASE structure)

CASE <expression> [OF]

      (DIM structure - strings)

DIM <string variable> OF <maximum string length>

      (DIM structure - string arrays)

DIM <string variable>(<array index>) OF <max chars each>

 <max chars each> is a positive <numeric expression>
 <maximum string length> is a positive <numeric expression>
 <array index> is represented by:
   [<bottom lim>:]<top limit>{,[<bottom lim>:]<top limit>}
     <bottom limit> is an optional <numeric expression>
       if omitted, the default value is 1
     <top limit> is a <numeric expression>
       it must be greater than the <bottom limit>
```

EXAMPLES

```
CASE response OF
DIM player'name$(3) OF 40
DIM text$ OF 80
```

SAMPLE PROGRAM

```
DIM reply$ OF 1
REPEAT
  INPUT "ARE YOU DONE YET: " : reply$
  CASE reply$ OF
  WHEN "Y"
    done:=TRUE
  OTHERWISE
    done:=FALSE
  ENDCASE
UNTIL done
PRINT "ALL DONE"

RUN
ARE YOU DONE YET: N
ARE YOU DONE YET: Y
ALL DONE
```

ADDITIONAL SAMPLES SEE: MOD, OTHERWISE, READ
USED IN PROCEDURES: FETCH, LOWER'TO'UPPER
SEE ALSO: CASE, DIM, ENDCASE, OTHERWISE, WHEN

CATEGORY: Command / Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows you to identify files for data input or output. You may have more than one file open at a time, but they each must have a different file number. To create a sequential file, you use WRITE as the ⟨type⟩. To add to a sequential disk file use APPEND as the ⟨type⟩. Then use WRITE FILE or PRINT FILE statements to write to the file. To read a sequential file, use READ as the ⟨type⟩. Then use READ FILE or INPUT FILE statements to read the data. To read and write from a direct access disk file use RANDOM as the ⟨type⟩. Then use READ FILE or WRITE FILE statements to access the file. Random access files reserve a fixed length for each record. The actual record may be shorter, but the same amount of space is used. The word FILE is optional and if left out will be supplied by the COMAL system.

NOTES

(1) A file may be opened to the screen or printer, as well as tape and disk.
(2) See Appendix C for more information on sequential files.
(3) To OPEN a file as type PRG or USR, instead of SEQ, include the type as part of the file name, preceded by a comma (i.e., OPEN FILE 3,"TEST,PRG",READ).
(4) ⟨type⟩ is optional if a specific unit is specified in the filename.
(5) See Appendix N for more information on filenames.
(6) Version 0.14 does not have the complete file name system as version 2.00 does. Thus the only place you can specify a device number or unit is in the OPEN statement as a tack on to the ⟨filename⟩. Simply add ,UNIT ⟨num⟩[,⟨secaddr⟩] to the file name, yielding:

```
OPEN FILE ⟨num⟩,⟨filename⟩[,UNIT ⟨num⟩[,⟨secaddr⟩]] [,⟨type⟩]
OPEN FILE 3,"my'prog",UNIT 9,READ  open device 9
OPEN FILE 7,"",UNIT 4,7            open device 4 sec addr 7
```

SYNTAX

```
OPEN [FILE] <filenum>,<filename>[,<type>]

 <type> is READ, WRITE, APPEND, or RANDOM <record len>
   <record len> is a positive <numeric expression>
```

EXAMPLES

```
OPEN FILE 2,"TESTFILE",READ
OPEN FILE infile,INNAME$,RANDOM rec'size
OPEN FILE infile,UNIT$+INNAME$,WRITE     version 2.00 only
```

SAMPLE PROGRAM

```
DIM filename$ OF 20, item$ OF 40
filename$:="VISITOR'READ"
infile:=4
OPEN FILE infile,filename$,READ
REPEAT
  READ FILE infile : item$
  PRINT item$
UNTIL EOF(infile)
CLOSE FILE infile
PRINT "ALL DONE"

RUN
  (system opens disk file # 4 named VISITOR'READ for input)
COMAL USERS GROUP        varies depending on file contents
  (file number 4 is closed)
ALL DONE
```

ADDITIONAL SAMPLES SEE: CLOSE, WRITE
USED IN FUNCTION: FILE'EXISTS
SEE ALSO: APPEND, CLOSE, EOF, FILE, INPUT, LOAD, PRINT, RANDOM, READ, SAVE, WRITE

CATEGORY: Operator
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Logical math operator evaluates to TRUE (a value of 1) if either (or both) ⟨numeric expression⟩ is TRUE (a value not equal to 0). Otherwise it evaluates to FALSE (a value of 0). The second sample program listed below produces a chart showing each of the four possible combinations. Version 2.00 adds the extended OR ELSE operator. It does not evaluate the second condition if the first is TRUE, since the result will be TRUE regardless of the outcome of the second condition. This will result in faster program execution and help avoid evaluation errors.

NOTES

(1) OR is not a bitwise operator as it is in Commodore BASIC. See BITOR for a bitwise operator.
(2) OR ELSE is not supported by version 0.14.

SYNTAX

```
<condition> OR <condition>
    or
<condition> OR ELSE <condition>      version 2.00 only

 <condition> is a <numeric expression>
```

EXAMPLES

```
letter<65 OR letter>90
item$:="Y" OR ELSE item$:="N"
```

SAMPLE PROGRAM 1

```
DIM letter$ OF 1
REPEAT
  INPUT "ENTER A LETTER (0 TO STOP): " : letter$;
  IF letter$<"A" OR letter$>"Z" THEN PRINT "NO";
  PRINT // CARRIAGE RETURN
UNTIL letter$="0"
PRINT "ALL DONE"

RUN
ENTER A LETTER (0 TO STOP): D
ENTER A LETTER (0 TO STOP): 5 NO
ENTER A LETTER (0 TO STOP): Z
ENTER A LETTER (0 TO STOP): 0 NO
ALL DONE
```

SAMPLE PROGRAM 2

```
DIM type$(FALSE:TRUE) OF 5
type$(FALSE):="FALSE";type$(TRUE):="TRUE"
ZONE 6
PRINT "OR CHART"
PRINT "--------"
FOR a:=FALSE TO TRUE DO
  FOR b:=FALSE TO TRUE DO
    PRINT "A=";TYPE$(a),"B=";TYPE$(b),
    PRINT "A OR B=";TYPE$(a OR b)
  ENDFOR b
ENDFOR a
ZONE 0

RUN
OR CHART
--------
A= FALSE    B= FALSE     A OR B= FALSE
A= FALSE    B= TRUE      A OR B= TRUE
A= TRUE     B= FALSE     A OR B= TRUE
A= TRUE     B= TRUE      A OR B= TRUE
```

ADDITIONAL SAMPLES SEE: MOD, RESTORE
SEE ALSO: AND, BITAND, BITOR, BITXOR, NOT

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns an integer representing the ordinal number (ASCII value) of the supplied string. If the string is longer than one character, ORD only looks at the first character. An error will result if you attempt to use the null string with ORD.

SYNTAX

```
ORD(<string expression>)
```

EXAMPLES

```
ORD(item$)
ORD(item$(num))
```

SAMPLE PROGRAM

```
ZONE 12                              for column spacing of 12
a:=ORD("A");z:=ORD("Z")
DIM word$ OF 40, letters(a:z)
INPUT "WORD: " : word$
FOR temp:=1 TO LEN(word$) DO
  char:=ORD(word$(temp))
  letters(char):+1                    increment letter count
ENDFOR temp
FOR letter:=a TO z DO
  IF letters(letter) THEN
    PRINT "LETTER:";CHR$(letter),"OCCURS:";letters(letter)
  ENDIF
ENDFOR letter
PRINT "ALL DONE"

RUN
WORD: CATALYST
LETTER: A   OCCURANCES: 2
LETTER: C   OCCURANCES: 1
LETTER: L   OCCURANCES: 1
LETTER: S   OCCURANCES: 1
LETTER: T   OCCURANCES: 2
LETTER: Y   OCCURANCES: 1
ALL DONE
```

ADDITIONAL SAMPLES SEE: CHR$, DO, GET$, IN, POKE, REF
USED IN FUNCTION: VALUE
SEE ALSO: CHR$, STR$, VAL

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows a default set of statements following the OTHERWISE to be executed if none of the WHEN comparisons in the CASE structure are TRUE (a value not equal to 0). OTHERWISE is optional and may be left out. However, if none of the WHEN conditions are met and there is no OTHERWISE, an error will result. See Appendix A for a description of the CASE structure.

SYNTAX

```
OTHERWISE
  <statements>
```

EXAMPLE

```
OTHERWISE
  PRINT "I DON'T UNDERSTAND YOUR ANSWER"
```

SAMPLE PROGRAM

```
DIM reply$ OF 1
REPEAT
  INPUT "GOOD, BAD, OR UGLY (G,B,U): " : reply$;
  CASE reply$ OF
  WHEN "G"
    PRINT "GOOD BYE"
  WHEN "B"
    PRINT "BAD"
  WHEN "U"
    PRINT "UGLY"
  OTHERWISE
    PRINT "TRY AGAIN"
  ENDCASE
UNTIL reply$="G"
PRINT "ALL DONE"

RUN
GOOD, BAD, OR UGLY (G,B,U): Z TRY AGAIN
GOOD, BAD, OR UGLY (G,B,U): U UGLY
GOOD, BAD, OR UGLY (G,B,U): B BAD
GOOD, BAD, OR UGLY (G,B,U): G GOOD BYE
ALL DONE
```

ADDITIONAL SAMPLES SEE: CASE, CHAIN, ENDCASE, READ, REF
USED IN PROCEDURE: FETCH
SEE ALSO: CASE, ENDCASE, OF, WHEN

CATEGORY: Statement / Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

If output is to the screen, the screen is cleared. If output is to a printer, a form feed (page eject) is issued.

NOTES

(1) PAGE is not supported by version 0.14.
(2) If output is to the screen a CHR$(147) is issued. If output is to the printer a CHR$(12) is issued. If you know where you output is currently going, you could simulate a PAGE statement in version 0.14.

SYNTAX

```
PAGE
```

EXAMPLES

```
PAGE
IF lines'left<4 THEN PAGE
```

SAMPLE PROGRAM

```
DIM name$ OF 30, company$ OF 30, reply$ OF 1
DATA "Robert Grimm","Universal Electronics",1
DATA "Sam Thompson","Global Technology",3
DATA "Paul Peterson","Modern Repairs",0
PAGE // clear screen
INPUT "Output to Screen or Printer (S/P): ": reply$
CASE reply$ OF
WHEN "S","s"
  SELECT OUTPUT "DS:"
WHEN "P","p"
  SELECT OUTPUT "LP:"
OTHERWISE // unknown reply
  SELECT OUTPUT "DS:" // default choose screen
ENDCASE
WHILE NOT EOD DO
  PAGE // clear screen or page feed
  READ name$,company$,level
  PRINT name$
  PRINT company$
  PRINT "Level:";level
  PRINT "(0=technician 1=Engineer 2=V.P. 3=President)"
  INPUT "Hit RETURN for next name": reply$
ENDWHILE
SELECT OUTPUT "DS:" // return the output to the screen
END "ALL DONE"
```

```
RUN
Output to Screen or Printer (S/P): P
  (printer opened for output, the following is printed)
Robert Grimm
Universal Electronics
Level: 1
(0=technician 1=Engineer 2=V.P. 3=President)
Hit RETURN for next name      prompt on screen - hit return
  (page eject on printer now)
Sam Thompson
Global Technology
Level: 3
(0=technician 1=Engineer 2=V.P. 3=President)
Hit RETURN for next name      prompt on screen - hit return
  (page eject on printer now)
Paul Peterson
Modern Repairs
Level: 0
(0=technician 1=Engineer 2=V.P. 3=President)
Hit RETURN for next name      prompt on screen - hit return
  (end of data - so output is returned to screen)
ALL DONE
```

ADDITIONAL SAMPLES SEE: ERR, EXTERNAL, FUNC, REPORT
SEE ALSO: SELECT OUTPUT

CATEGORY: Command / Statement
KERNAL: [NO] VERS 0.14 [+] VERS 2.00 [*]

PASSes a string expression to the CBM disk. These strings will be interpreted by the disk drive as DOS commands as documented by the CBM disk drive manual. To scratch a file see the keyword DELETE. See Appendix O for a list of disk commands.

NOTES

(1) Version 0.14 does not support special unit numbers.
(2) See Appendix O for a categorized list of disk commands.
(3) Version 2.00 allows a unit to be specified at the beginning of the string (i.e., "u9:v0" would use unit 9 for the validate command). COMAL accepts the first three characters of the string as a unit specifier if they are Un:, with n representing a digit. This presents a potential conflict with the DOS USER command. However, this conflict can be avoided easily. To send the DOS command "U2:⟨parms⟩" you should add the unit specifier in front of it—"U8:U2:⟨parms⟩"; or if using the current unit—UNIT$+"U2:⟨parms⟩".

SYNTAX

```
PASS <disk command>

 <disk command> is a <string expression>
  version 2.00 may include unit information in the string
```

EXAMPLES

```
PASS "I0"                  initialize drive 0
PASS "N1:WORK DISK,A1"     new (format) disk in drive 1
PASS "V0"                  validate (collect) drive 0
PASS "D1=0"                duplicate drive 0 to drive 1
PASS "D0=1"                duplicate drive 1 to drive 0
PASS "R0:NEW'NAME=OLD'NAME"          rename a file
PASS "C0:FILENAME=1:FILENAME"        copy a file
PASS disk'com$             string variables may be used
```

SAMPLE PROGRAM

```
DIM text$ OF 80
REPEAT
  INPUT "DISK COMMAND (STOP TO STOP): " : text$
  IF text$<>"STOP" THEN PASS text$
UNTIL text$="STOP"
PRINT "ALL DONE"

RUN
DISK COMMAND (STOP TO STOP): C0=1
  (the disk in drive 1 is copied to drive 0)
DISK COMMAND (STOP TO STOP): STOP
ALL DONE
```

SEE ALSO: CAT, COPY, CREATE, DELETE, MOUNT, RENAME

CATEGORY: Function
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

Returns the decimal value of the contents in the specified memory location. The result is always an integer from 0 to 255. PEEK tends to be very machine dependent, i.e., not very portable among the various models of Commodore computers or other COMAL computers. A chart listing some memory locations of interest follows:

CBM/PET	PURPOSE OF LOCATION	CBM 64
141-143	JIFFY CLOCK	160-162
151	LAST KEY PRESSED (255=NONE)	197
152	SHIFT KEY (0=UP 1=DOWN)	653
158	KEYSTROKE BUFFER COUNT	198
159	REVERSE FIELD (0=OFF 1=ON)	145
198	POSITION OF CURSOR ON LINE (0-79)	211
205	QUOTE MODE (0=OFF 1=ON)	212
216	LINE THAT THE CURSOR IS ON (0-24)	214
623-632	KEYBOARD BUFFER	631-640
32768-33767	VIDEO SCREEN MEMORY (40 COLUMN)	1024–2023
32768-34767	VIDEO SCREEN MEMORY (80 COLUMN)	n/a

SYNTAX

```
PEEK(<memory address>)

 <memory address> is a <numeric expression>
   whose value is from 0-65535 (valid memory addresses)
```

EXAMPLES

```
PEEK(32768)
PEEK(loc)
```

SAMPLE PROGRAM 1 (for PET or CBM, not Commodore 64)

```
videostart:=32768
videolength:=2000                    40 column screen use 1000
starting:=videostart
ending:=starting+videolength
FOR x:=starting TO ending DO
  IF PEEK(x)<128 THEN
    POKE x,PEEK(x)+128
  ELSE
    POKE x,PEEK(x)-128
  ENDIF
ENDFOR x
PRINT "ALL DONE"

RUN
   (each character on the screen reverses)
ALL DONE
```

SAMPLE PROGRAM 2

```
shift:=152           Commodore 64 use 653
REPEAT
  IF NOT PEEK(shift) THEN PRINT CHR$(147)
  IF PEEK(shift) THEN PRINT "[HOME]SHIFT IS DOWN"
UNTIL FALSE       note: replace [HOME] with cursor home key

RUN
  (screen clears)
  (depress the SHIFT key and words SHIFT IS DOWN appears)
  (HIT the RUN/STOP key to stop this program)
```

ADDITIONAL SAMPLE SEE: DO
USED IN PROCEDURES: DISK'GET, GET'CHAR, JIFFIES, POS, SCANKEY, SHIFT
SEE ALSO: POKE, SYS

CATEGORY: Function
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

Places the specified decimal value into the specified decimal memory location. If the specified location is unalterable (i.e., ROM or empty) then nothing happens. WARNING: POKEing a wrong value into some locations may "lock out" your machine, or cause it to act in a very peculiar way. The following chart lists some memory locations that you may be interested in POKEing around with.

CBM/PET	PURPOSE OF LOCATION	CBM 64
141-143	JIFFY CLOCK	160-162
158	KEYSTROKE BUFFER COUNT	198
159	REVERSE FIELD (0=OFF 1=ON)	145
198	POSITION OF CURSOR ON LINE (0-79)	211
205	QUOTE MODE (0=OFF 1=ON)	212
216	LINE THAT THE CURSOR IS ON (0-24)	214
623-632	KEYBOARD BUFFER	631-640
32768-33767	VIDEO SCREEN MEMORY (40 COLUMN)	1024-2023
32768-34767	VIDEO SCREEN MEMORY (80 COLUMN)	n/a
59468	12=GRAPHIC MODE 14=LOWER CASE MODE	n/a
n/a	21=GRAPHIC MODE 23=LOWER CASE MODE	53272

To clear the keyboard buffer you could use the following statement:

```
POKE 158,0 (PET/CMB) or POKE 198,0 (C64)
```

CBM COMAL version 2.00 has a control location at hex $24B allowing:
Bit 0: 0= normal cbm mode 1=convert control codes to ""⟨ctrl⟩""
Bit 1: 0= normal cbm mode 1= ignore quote mode and insert mode
Bit 2: 0= list ENDFOR 1= list NEXT as FOR terminator
Bit 3: 0= do not list EXEC 1= List EXEC keyword
Bit 4: reserved for future use
Bit 5: reserved for future use
Bit 6: 0= identifiers in lower case 1= identifiers in UPPER case
Bit 7: 0= keywords in lower case 1= keywords in UPPER case

⟨ctrl⟩ is replaced by the ASCII control code number.

Default values of the 8 bits are:

```
Decimal 130, Binary %10000010
```

To have your program listings show NEXT instead of ENDFOR try:
POKE $24B,PEEK($24B) BITOR %00000100

SYNTAX

```
POKE <memory address>,<contents>

 <memory address> is a <numeric expression>
   whose value is from 0-65535 (a valid memory address)
 <contents> is a <numeric expression> whose value is 0-255
```

EXAMPLES

```
POKE 59468,14    (PET/CBM - not C64) switch to lower case
POKE loc,contents
```

SAMPLE PROGRAM 1 (for PET or CBM, not Commodore 64)

```
videostart:=32768
videolength:=2000                40 column screen use 1000
a:=ORD("A"); z:=ORD("Z")
PRINT "[CLR]"       note:replace [CLR] with clear screen key
FOR temp:=1 TO 99 DO
  POKE videostart+RND(0,videolength),RND(a,z)
ENDFOR temp
PRINT "ALL DONE"

RUN
  (the screen clears)
  (the screen now fills randomly with random letters)
  (99 random screen locations are changed)
ALL DONE
```

SAMPLE PROGRAM 2 (Version 2.00 only)

```
//Setup Modes - how programs list and are edited
PAGE //         use this to set up the system YOUR way
modes:=$024B
POKE modes,on'off("CONTROL CODES listed as number",0,modes)
POKE modes,on'off("QUOTE MODE and INSERT MODE off",1,modes)
POKE modes,on'off("List the keyword EXEC",2,modes)
POKE modes,on'off("List NEXT instead of ENDFOR",3,modes)
POKE modes,on'off("Identifiers in UPPER case",6,modes)
POKE modes,on'off("Keywords in UPPER case",7,modes)
END "MODES NOW SET"

FUNC on'off(prompt$,bit,modes) CLOSED
  DIM reply$ of 1
  INPUT prompt$+SPC$(34-LEN(prompt$))+"> " : reply$
  IF reply$="N" or reply$="n" THEN
    RETURN PEEK(modes) BITAND $ff-2^bit
  ELSE
    RETURN PEEK(modes) BITOR 2^bit
  ENDIF
ENDFUNC

RUN
CONTROL CODES listed as a number > N
QUOTE MODE and INSERT MODE off   > Y
List the keyword EXEC            > N
List NEXT instead of ENDFOR      > N
Identifiers in UPPER case        > N
Keywords in UPPER case           > Y
MODES NOW SET
```

ADDITIONAL SAMPLE SEE: PEEK
USED IN PROCEDURES: GET'CHAR, SCANKEY
SEE ALSO: PEEK, SYS

CATEGORY: Command / Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Prints items as specified. These items may be either constants or variables, string or numeric. More than one item may be specified per PRINT statement. Multiple items must be separated either by a semicolon (;), which yields one space between items, or by a comma (,), which yields spaces up to the next ZONE set. (Default ZONE is 0 for no extra spaces.) TAB(X) may be used, as well as PRINT USING, which allows formatted output. Items will print on a printer or the screen in the same way, as well as to a file if the word FILE and a ⟨file number⟩ is included. Version 2.00 allows you to begin printing at any spot on the screen via the PRINT AT statement, where you specify what row and column to start at. Shorthand for PRINT is ;.

NOTES

(1) When creating data files, it is preferred to use WRITE FILE statements instead of PRINT FILE statements.
(2) PRINT AT is not supported by version 0.14.
(3) PRINT USING is not supported by PET/CBM version 0.14.
(4) To print to disk or tape include the word FILE and a file number (the file must previously have been opened as a WRITE or APPEND).
(5) PRINT FILE and WRITE FILE are not compatible because PRINT files are written in ASCII while WRITE files are written in binary.
(6) Use INPUT FILE to read a file created with PRINT FILE.
(7) PRINT FILE is compatible with PET/CBM BASIC PRINT#.
(8) Version 0.14 uses CHR$(13)+CHR$(10) as a delimiter between records. Version 2.00 uses only CHR$(13) unless a file attribute of /L+ is specified.

COMAL fully supports the special editing functions available on the CBM 8032 and CBM 8096. The following functions can be included in a PRINT statement:

CHR$(7)	ring the bell in the computer
CHR$(14)	enter lower case mode
CHR$(15)	set cursor position as upper left corner of window
CHR$(21)	delete the cursor line
CHR$(22)	erase cursor line from cursor position to line's end
CHR$(25)	scroll screen up

CHR$(142) enter graphic mode (UPPER case and graphics)
CHR$(143) set cursor position as bottom right corner of window
CHR$(149) insert a line at current cursor line
CHR$(150) erase cursor line up to cursor position
CHR$(153) scroll screen down

SYNTAX

```
PRINT [FILE <filenum>:] [<print list>][<mark>]
PRINT AT <row>,<col>: [<print list>][<mark>]     vers 2.00

 <print list> can include one or more of the following
   separated by a comma or semicolon:
   TAB(<pos>)
     <pos> is a <numeric expression> whose value is 1-255
   <string expression>
   <numeric expression>
   USING <string expression>: <variable name list>
     <string expression> may be a constant or variable
       used to set up USING image, the following reserved
         #  reserves a digit place
         .  the location of the decimal point
         -  floating minus sign (optional)
     <variable name list> is a list of the variables to use
       in filling the USING image.
 <mark> may be either a comma (,) or a semicolon (;)
   if omitted, a carriage return and line feed result
 <row> is a <numeric expression> whose value is from 0-25
   (0 means current row)
 <col> is a <numeric expression> whose value is from 0-80
   on 40 column screens, the value is from 0-40
```

EXAMPLES

```
PRINT
PRINT "YOUR SCORE WAS";score;
PRINT USING price$: item'price
PRINT player'name$,TAB(30),score
PRINT FILE outfile: name$
PRINT 2,"NEWS","DATE",company$,
PRINT AT 12,14: USING "##.##": amount
```

SAMPLE PROGRAM

```
DIM word$ OF 40
word$:="VARIABLES";money:=25.6
PRINT TAB(5),"CONSTANTS AND";word$
PRINT USING "TOTAL DUE $###.##" : money
PRINT                                blank line
ZONE 5                               new zone
PRINT "A","B","C"                    commas use zone positions
ZONE 0
PRINT "A","B","C";                   no carriage return at end
PRINT "END OF LINE"
PRINT "ALL DONE"

RUN
    CONSTANTS AND VARIABLES
TOTAL DUE $ 25.60

A    B    C
ABC END OF LINE
ALL DONE
```

ADDITIONAL SAMPLES SEE: MOST KEYWORD SAMPLE PROGRAMS
USED IN PROCEDURES: CURSOR, FETCH, TAKE'IN
SEE ALSO: AT, FILE, INPUT, OPEN, READ, SELECT OUTPUT, TAB, USING, WRITE, ZONE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Prints the value of any expression, string or numeric, to a previously opened RANDOM file. More than one expression may be specified per PRINT statement. Multiple items must be separated either by a semicolon (;), which yields one space between items, or by a comma (,) which yields spaces up to the next set ZONE. (Default ZONE is 0 for no extra spaces.) TAB(X) may be used, as well as PRINT USING which allows formatted output. A CHR$(13) and CHR$(10) are included after each record by the system. Shorthand for the keyword PRINT is ;.

NOTES

(1) PRINT USING is not supported by version 0.14 for the PET/CBM.
(2) PRINT FILE and WRITE FILE are not compatible.
(3) Use INPUT FILE to read a record created with PRINT FILE.
(4) Version 2.00 allows you to omit the record number in the statement. You will then continue in the same record.

SYNTAX

```
PRINT [FILE] <file#>[,<rec#>[,<offset>]]: [<plist>][<mark>]

 <rec#> is a positive <numeric expression>
   it may be omitted in version 2.00.
 <offset> is a positive <numeric expression>;
   if omitted, no bytes will be skipped
 <plist> can include one or more of the following
   separated by a comma or semicolon:
   TAB(<pos>)
     <pos> is a <numeric expression> whose value is 1-255
     <string expression>
   <numeric expression>
   USING <string expression>: <numeric expression list>
     <string expression> may be a constant or variable
       used to set up USING image, the following reserved
         #  reserves a digit place
         .  the location of the decimal point
         -  floating minus sign (optional)
     <numeric expression list> is list of expressions
       in filling the USING image.
 <mark> may be either a comma (,) or a semicolon (;)
   if omitted, a carriage return and line feed result
   if included, same record continues
```

EXAMPLES

```
PRINT FILE 2,5 : "NAME:"+NAME$
PRINT FILE outfile,recnum : text$
```

SAMPLE PROGRAM 1

```
DIM text$ OF 80
OPEN FILE 2,"RANDOM'INPUT",RANDOM 80
PRINT "WRITE SOME RANDOM TEXT RECORDS"
REPEAT
  INPUT "WHAT RECORD NUMBER (0 TO STOP): " : number
  IF number THEN
    INPUT "TEXT:" : text$
    PRINT FILE 2,number: text$
  ENDIF
UNTIL number=0
CLOSE
PRINT "ALL DONE"

RUN
  (file 2 named RANDOM'INPUT is opened for random access)
WRITE SOME RANDOM TEXT RECORDS
WHAT RECORD NUMBER (0 TO STOP): 5
TEXT:THIS IS THE FIFTH RECORD
  (THIS IS THE FIFTH RECORD is written to file 2, record 5)
WHAT RECORD NUMBER (0 TO STOP): 9
TEXT:THIS IS THE NINTH RECORD
  (THIS IS THE NINTH RECORD is written to file 2, record 9)
WHAT RECORD NUMBER (0 TO STOP): 0
  (file 2 is closed)
```

SAMPLE PROGRAM 2 (version 2.00 only)

```
DIM text$ of 20
OPEN FILE 2,"ran",RANDOM 50
PRINT FILE 2,20: 1/4 // first element
PRINT FILE 2: "mystring" // second element of record 20
PRINT FILE 2: 10+2 // third element of record 20
INPUT FILE 2,20: num // read element 1 of record 20
INPUT FILE 2: text$,sum // read elements 2 & 3 of record 20
CLOSE FILE 2
PRINT num;text$;sum
PRINT "All done."

RUN
 (Random file number 2 is opened)
 (Writes 3 fields into record 20)
 (Reads values for 3 variables from record 20)
.25 mystring 12
All done.
```

SEE ALSO: FILE, INPUT, OPEN, RANDOM, TAB, USING, WRITE, ZONE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Start of a procedure, allowing parameter passing and local or global variables. Procedures can be very flexible, versatile, and easy to use. However, using all the options (REF, CLOSED, IMPORT, EXTERNAL, and parameter passing) can be complex but powerful. Procedures may call other procedures, or even call themselves. Version 2.00 also allows procedures to be nested (i.e., you can define a procedure within another procedure). To make the best use of a procedure, a good programming tutorial is recommended.

Parameter passing is optional. A simple procedure without parameters is often all that may be needed. A procedure can be made CLOSED, so that all variables in it are local (unknown to all other parts of the program) except for specific variables made global (shared with other parts of the program) in version 2.00 with an IMPORT statement. ZONE is automatically global even in a CLOSED procedure.

Strings and entire arrays may be used as parameters. They are dimensioned automatically as they are passed into the procedure. They can be local or used as an alias for the string variable or array in the main program using the keyword REF before the variable name (using REF will use up less memory). To specify that a parameter is to be used as an array simply include the parentheses after the name (i.e., SCORES() will be a single dimension array). For multiple dimension arrays, simply include the same number of commas as used when the array was dimensioned (i.e., TABLE(,,) will be a three dimension array).

The values passed to each procedure parameter must of course be of the same type. An error will occur if the procedure expects a two dimensional array and receives only a single value.

If a parameter is preceded by the keyword REF, the variable or array in the calling statement will change as the procedure variable or array elements change, even though the procedure may use a different variable or array name (an alias name). If a variable or array is declared to be global with an IMPORT statement, its name in the procedure will be the same as outside the procedure and any changes made inside the procedure will be global in effect. Version 2.00 also requires that

all procedures or functions called from within a closed procedure or function be declared global with an IMPORT statement. Version 0.14 does not support the IMPORT statement, and automatically makes all procedures and functions global within closed procedures. See Appendix A for a description of the procedure structure.

A procedure can also be "external" to the current program. In this case, all that is needed in the current program is the PROC header statement, with the EXTERNAL specification. When the program is run, the procedure or function body is retrieved and used only when needed. This allows you to have a "procedure library disk" that you can update when needed. Thus each program will always be calling the latest version of each procedure without the need to update each program individually. Each program will also be significantly shorter, since the procedure body is not included in the program. This also allows several different procedures to share memory space in a running program. See Appendix A for details on how to create and use EXTERNAL procedures and functions.

NOTES

(1) The system will "remember" all procedure names once a program is run. Any procedure can then be executed from direct mode with the EXEC command. This is very powerful, similar to adding keywords or program function keys, without the worry of incompatibility between users. This also includes any EXTERNAL procedures.
(2) There are no restrictions on parameter passing with EXTERNAL procedures.
(3) EXTERNAL procedures must be CLOSED, but do NOT include the keyword CLOSED in the PROC header of the calling program, since the CLOSED keyword is in the external file instead.
(4) A procedure SAVEd to disk may be used as an external procedure. It may have remarks or blank lines before it starts. Statements are also permitted to exist after the ENDPROC, but these statements will not be executed. See Appendix A for more information on this.
(5) It is suggested to end the ⟨filename⟩ of an EXTERNAL procedure or function with .E or .EXT to avoid future confusion with files.
(6) EXTERNAL is not supported by version 0.14.
(7) IMPORT statements are not allowed in external procedures.

(8) Arrays must be passed by reference (REF) in version 0.14. If an array is passed by value in version 2.00, it is totally copied in memory, using up twice as much memory.

SYNTAX

```
PROC <procname>[(<formal parms>)] [EXTERNAL <filename>]
        or
PROC <procname>[(<formal parms>)] [CLOSED]

 <procname> is an <identifier>
 <formal parms> is optional and represented by:
   [REF ]<variable name>{,[REF ]<variable name>}
     initial value of each <variable name> will be assigned
     from the value in the calling statement
     these variables will be considered LOCAL
```

EXAMPLES

```
PROC find'it
PROC errorprint(e,REF er$(,))
PROC sort CLOSED
PROC print'it(n$,a$,c$,s$,z$) EXTERNAL "PRINT'IT.E"
```

SAMPLE PROGRAM

```
DIM word$ OF 80
REPEAT
  INPUT "WORD (0 TO STOP): " : word$;
  EXEC backwards(word$)
UNTIL word$="QUIT"
PRINT "ALL DONE"
//
PROC backwards(a$) CLOSED
  FOR temp:=LEN(a$) TO 1 STEP -1 DO PRINT a$(temp),
  PRINT                    provide carriage return
ENDPROC backwards

RUN
WORD (0 TO STOP): COMAL LAMOC
WORD (0 TO STOP): NOWHERE EREHWON
WORD (0 TO STOP): QUIT TIUQ
WORD (0 TO STOP): 0 0
ALL DONE
```

ADDITIONAL SAMPLES SEE: CLOSED, ENDPROC, EXEC, GET$, IMPORT, INTERRUPT
USED IN PROCEDURES: ALL PROCEDURES
SEE ALSO: CLOSED, ENDFUNC, ENDPROC, EXEC, FUNC, IMPORT, INTERRUPT, REF, RETURN

CATEGORY: Type of file
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Identifies the disk file being opened as a RANDOM (direct access) file. The record length is a fixed length specified following the keyword RANDOM. The actual record may be shorter, but the same amount of space is used. This type of file allows both reading and writing any record, one at a time, also known as direct access. The word FILE is optional and if omitted will be supplied by the system. See Appendix L for more information on random files.

NOTES

(1) When a string is written to a record using a WRITE FILE statement, the actual string is preceded by a two byte character counter. Although this counter is transparent to your program, you do lose two characters of the total record length. Thus to be sure to fit an entire 80 character string, use a record length of 82.
(2) The CREATE statement and CREATE procedure in Appendix D both place a CHR$(255) in the first position of the last record of the file.

SYNTAX

```
OPEN [FILE] <filenum>,<filename>,RANDOM <record length>

 <record length> is a <numeric expression>
   whose value is 1-254
```

EXAMPLES

```
OPEN FILE 2,"CUSTOMERS",RANDOM 100
OPEN FILE infile,file'name$,RANDOM rec'size
```

SAMPLE PROGRAM

```
DIM item$ OF 80
OPEN 2,"0:RANDOM'READ",RANDOM 80
PRINT "SAMPLE RANDOM ACCESS"
REPEAT
  INPUT "WHAT RECORD NUMBER (0 TO STOP): " : number
  IF number>0 AND number<10 THEN      allow only 10 records
    INPUT "READ OR WRITE IT: " : item$
    CASE item$ OF
    WHEN "R","READ"
      EXEC read'it
    WHEN "W","WRITE"
      EXEC write'it
    OTHERWISE
      PRINT "NO SUCH OPTION"
    ENDCASE
  ENDIF
UNTIL number=0
CLOSE
PRINT "ALL DONE"
END
//
PROC read'it
  READ FILE 2,number: item$
  PRINT item$
ENDPROC read'it
//
PROC write'it
  PRINT "ENTER UP TO 80 CHARACTERS TO WRITE"
  INPUT ">" : item$
  WRITE FILE 2,number: item$
ENDPROC write'it
```

```
RUN
  (file 2 is opened for random access)
SAMPLE RANDOM ACCESS
WHAT RECORD NUMBER (0 TO STOP): 4
READ OR WRITE IT: W
ENTER UP TO 80 CHARACTERS TO WRITE
>ITEM FOUR
  (ITEM FOUR is written to record number 4)
WHAT RECORD NUMBER (0 TO STOP): 2
READ OR WRITE IT: WRITE
ENTER UP TO 80 CHARACTERS TO WRITE
>ITEM NUMBER TWO
  (ITEM NUMBER TWO is written to record number 2)
WHAT RECORD NUMBER (0 TO STOP): 4
READ OR WRITE IT: R
  (record number 4 is read)
ITEM FOUR
WHAT RECORD NUMBER (0 TO STOP): 0
  (file 2 is closed)
ALL DONE
```

SEE ALSO: APPEND, CLOSE, CREATE, FILE, OPEN, READ, WRITE

CATEGORY: Statement / Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

RANDOMIZEs the random number generator. If a ⟨seed⟩ is not specified, TIME is used (the number of jiffies will always be different), which begins a truly random sequence. A specific ⟨seed⟩ will start the same sequence each time it is called. The system converts the ⟨seed⟩ into its negative equivalent before using it.

NOTES

(1) RANDOMIZE is normally used without specifying the ⟨seed⟩. The ⟨seed⟩ parameter is useful only if you need exactly the same sequence of random numbers from one run of the program to the next.
(2) If 0 is used as the ⟨seed⟩, it is the same as not specifying the ⟨seed⟩ (TIME is used for the seed).
(3) COMAL converts any positive ⟨seed⟩ into its negative before using it.
(4) RANDOMIZE is not supported by version 0.14. It can be simulated with the RANDOMIZE procedure listed in Appendix D.

SYNTAX

```
RANDOMIZE [<seed>]

 <seed> is a <numeric expression>
   if omitted, TIME is used
```

EXAMPLES

```
RANDOMIZE                 true randomize
RANDOMIZE 9               pseudo randomize with seed of 9
```

SAMPLE PROGRAM

```
// heads or tails coin flip - stop when one occurs 5 times
heads:=0; tails:=0
RANDOMIZE 9                    pseudo random numbers
REPEAT
  flip:=RND(0,1)
  IF flip=0 THEN
    PRINT "HEADS"
    heads:+1
  ELSE
    PRINT "TAILS"
    tails:+1
  ENDIF
UNTIL heads>4 OR tails>4
winner(heads,tails) //execute procedure to chose the winner
END "ALL DONE"
PROC winner(head,tail) CLOSED
  IF head>tail THEN
    PRINT "The winner is HEADS"
  ELSE
    PRINT "The winner is TAILS"
  ENDIF
ENDPROC winner
```

```
RUN       every run will have exact same results
HEADS       specific results depend upon computer
HEADS       and the version of COMAL used
TAILS
HEADS
TAILS
HEADS
HEADS
The winner is HEADS
```

SEE ALSO: RND

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

READs a value from a DATA statement. One or more variables can have values assigned to them from DATA statements via a READ statement. The data must be of the same type as the variable (i.e., string or numeric). When the last data item is READ by the program, the system variable, EOD, is set to be equal to TRUE (a value of 1). Commas (,), colons (:), and semicolons (;) may be part of a string constant. String constants must be enclosed in quote marks ("). A quote mark can be made part of a string constant by using two consecutive quote marks (i.e., "abc""def" will be read as abc"def).

NOTE

Data is considered LOCAL inside a CLOSED procedure in version 2.00.

SYNTAX

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

EXAMPLES

```
READ room,total
READ score,player$
```

SAMPLE PROGRAM

```
DIM month$(1:12) OF 3
DATA "JAN","FEB","MAR","APR","MAY","JUN"
DATA "JUL","AUG","SEP","OCT","NOV","DEC"
FOR temp:=1 TO 12 DO READ month$(temp)
REPEAT
  INPUT "WHICH MONTH NUMBER (0 TO STOP): " : num;
  CASE num OF
  WHEN 0
    PRINT "STOP"
  WHEN 1,2,3,4,5,6,7,8,9,10,11,12
    PRINT month$(num)
  OTHERWISE
    PRINT "OOOPS."
    PRINT "I DON'T KNOW A MONTH WITH THAT NUMBER"
  ENDCASE
UNTIL num=0
PRINT "ALL DONE"

RUN
WHICH MONTH NUMBER (0 TO STOP): 5 MAY
WHICH MONTH NUMBER (0 TO STOP): 1 JAN
WHICH MONTH NUMBER (0 TO STOP): 25 OOOPS.
I DON'T KNOW A MONTH WITH THAT NUMBER
WHICH MONTH NUMBER (0 TO STOP): 12 DEC
WHICH MONTH NUMBER (0 TO STOP): 0 STOP
ALL DONE
```

ADDITIONAL SAMPLES SEE: DATA, EOD, RESTORE, SELECT OUTPUT, SGN
SEE ALSO: APPEND, CLOSE, DATA, EOD, EOF, FILE, OPEN, RANDOM, WRITE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

READs data from a disk or tape sequential file that previously was OPENed as a READ file. The data read must be of the same type as the variable it is assigned to (i.e., string or numeric). More than one item may be read with one READ statement, each item separated by a comma.

NOTES

(1) Version 2.00 allows one statement to specify that an entire array is to have its values assigned from a sequential file. Simply use the array name without its parentheses section, i.e.,

```
DIM table(3,3)
OPEN FILE 2,"0:TABLE'VALUES",READ
READ FILE 2: table
CLOSE FILE 2
(The third line reads an entire array from disk.)
```

(2) READ FILE is used to read files created with WRITE FILE statements and is incompatible with files created with PRINT FILE, since they are ASCII format instead of binary.
(3) See Appendix C and Appendix L for more information about sequential files.

SYNTAX

```
READ FILE <filenum> : <variable>{,<variable>}
```

EXAMPLES

```
READ FILE 4 : name$
READ FILE infile : score,date,player$
READ FILE 3 : item$(10,num)
```

SAMPLE PROGRAM

```
DIM filename$ OF 20, item$ OF 40
infile:=4
filename$:="VISITOR'READ"
OPEN FILE infile,filename$,READ
READ FILE infile : item$
PRINT item$
CLOSE FILE infile
PRINT "ALL DONE"

RUN
  (system opens disk file # 4 named VISITOR'READ for input)
COMAL USERS GROUP     varies depending on contents of file
  (system closes file number 4)
ALL DONE
```

ADDITIONAL SAMPLE SEE: EOF
SEE ALSO: APPEND, CLOSE, DATA, EOD, EOF, FILE, GET$, INPUT, OPEN, WRITE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

READs data from a disk RANDOM or direct access file that previously was OPENed as a RANDOM type file and correctly specifying the fixed record length. Random access files reserve a fixed length for each record. The actual record may be shorter, but the same amount of space is used. Any record can be read at any time by specifying its record number. Any number of bytes (or characters) may be skipped at the beginning of the record as specified by the ⟨offset⟩.

NOTES

(1) READ is not compatible with files created by PRINT FILE statements as they are ASCII files, and READ expects a binary file.
(2) If you attempt to read a record not yet written by a previous WRITE FILE statement, you will either get garbage (nonsense) or a possible error.
(3) If you attempt to read a record past the last created record, you will get a disk error.
(4) In version 2.00 if you omit the ⟨record number⟩ you will read the next field in the last accessed record, or if that record is finished, you will read the next record.

SYNTAX

```
READ FILE <filenum>[,<record number>[,<offset>]]: <varlist>

 <record number> is a positive <numeric expression>
   whose value is a valid record number
 <offset> is a positive <numeric expression>;
   if omitted, no bytes will be skipped
 <varlist> is <variable>{,<variable>}
```

EXAMPLES

```
READ FILE 3,5: name$
READ FILE infile,rec'no: item$,price
```

SAMPLE PROGRAM

```
DIM text$ OF 80
OPEN FILE 5,"RANDOM'READ",RANDOM 80
PRINT "READ SOME RANDOM TEXT RECORDS"
REPEAT
  INPUT "WHAT RECORD NUMBER (0 TO STOP): " : number
  IF number THEN
    READ FILE 5,number: text$
    PRINT text$
  ENDIF
UNTIL number=0
CLOSE
PRINT "ALL DONE"

RUN
  (file 5 named RANDOM'READ is opened for random access)
READ SOME RANDOM TEXT RECORDS
WHAT RECORD NUMBER (0 TO STOP): 5
  (record number 5 is read from file 5)
THIS IS THE FIFTH RECORD
WHAT RECORD NUMBER (0 TO STOP): 9
  (record number 9 is read from file 5)
THIS IS THE NINTH RECORD
WHAT RECORD NUMBER (0 TO STOP): 0
  (file 5 is closed)
ALL DONE
```

ADDITIONAL SAMPLE SEE: RANDOM
SEE ALSO: CLOSE, FILE, GET$, INPUT, OPEN, RANDOM, WRITE

CATEGORY: Type of OPEN
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Specifies that the disk or tape file being OPENed is a sequential input file that will be read. Records from the file can be read with either a READ FILE or INPUT FILE statement. Or you may get one character at a time using GET$ (version 2.00 only) or procedure DISK'GET from Appendix D. For more information about sequential files, see Appendix C. You may open the screen as a file and read characters directly from it with INPUT FILE and GET$ statements. The screen is device 3, and requires UNIT 3 as part of the OPEN statement (i.e., version 0.14: OPEN FILE 7,"",UNIT 3,READ — version 2.00: OPEN FILE 7,"DS:",READ). The word FILE is optional and if omitted will be supplied by the system.

SYNTAX

```
OPEN [FILE] <filenum>,<filename>,READ
```

EXAMPLES

```
OPEN FILE 4,"SCORES",READ
OPEN FILE infile,filename$,READ
```

SAMPLE PROGRAM

```
DIM item$ OF 40
OPEN FILE 4,"VISITOR'INPUT",READ
READ FILE 4 : item$
PRINT item$
CLOSE
PRINT "ALL DONE"

RUN
 (system opens disk file # 4 named VISITOR'INPUT for input)
COMAL USERS GROUP     varies depending on contents of file
 (system closes file number 4)
ALL DONE
```

ADDITIONAL SAMPLES SEE: EOF, GET$
SEE ALSO: APPEND, CLOSE, DATA, EOF, FILE, OPEN, RANDOM, WRITE

CATEGORY: Type of parameter
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Specifies that the parameter will be an alias for the matching variable or array in the calling statement (it is passed by reference rather than by value). This saves memory space while still allowing a different variable name. The initial value is shared and the original is changed along with the alias as the procedure or function is executed. See PROC or FUNC for more details on parameters. In version 2.00 arrays may be passed to the procedure or function without the use of REF, but more memory will then be used. Arrays used as a parameter need not be dimensioned within the procedure. To specify in the PROC or FUNC statement that a parameter is an array, include a set of parentheses after the array name, like SORT(). If it is a multidimension array, include the same number of commas as would be used in its DIMension statement, i.e., SORT(,,) for a three dimensional array. See Appendix A for a description of the procedure structure.

SYNTAX

```
PROC <proc name>[(<parameter list>)] [EXTERNAL <filename>]
  or
PROC <proc name>[(<parameter list>)] [CLOSED]
  or
FUNC <func name>[(<parameter list>)] [EXTERNAL <filename>]
  or
FUNC <func name>[(<parameter list>)] [CLOSED]

 <proc name> is an <identifier>
 <func name> is an <identifier>
 <parameter list> is optional and represented by:
   [REF ]<variable name>{,[REF ]<variable name>}
```

EXAMPLES

```
PROC scores(REF staff$)
PROC compress(REF prog$,type) CLOSED
```

SAMPLE PROGRAM

```
a:=ORD("A"); z:=ORD("Z")
DIM junk$(3) OF 1
FOR temp:=1 TO 3 DO junk$(temp):=CHR$(RND(a,z))//random
REPEAT
  INPUT "WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): " : which
  CASE which OF
  WHEN 1,2,3
    EXEC see(which,junk$)
  OTHERWISE
    PRINT "NO SUCH JUNK"
  ENDCASE
UNTIL NOT which
PRINT "ALL DONE"
//
PROC see(num,REF a$()) CLOSED
  PRINT "JUNK NUMBER";num;"IS";a$(num)
ENDPROC see
```

```
RUN
WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): 2
JUNK NUMBER 2 IS C
WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): 4
NO SUCH JUNK
WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): 3
JUNK NUMBER 3 IS M
WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): 1
JUNK NUMBER 1 IS Y
WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): 3
JUNK NUMBER 3 IS M
WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): 0
NO SUCH JUNK
ALL DONE
```

USED IN PROCEDURES: GET'CHAR, LOWER'TO'UPPER, SCANKEY
SEE ALSO: CLOSED, ENDFUNC, ENDPROC, EXEC, FUNC, IMPORT, PROC

CATEGORY: Statement / Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

RENAMEs a disk file. Takes an existing disk file and gives it a new name.

NOTES

(1) RENAME is not supported by version 0.14. The keyword PASS can be used to do the same thing.
(2) Both files must be on the same disk unit.

SYNTAX

```
RENAME <old file name>,<new file name>

 <old file name> is a <filename>
 <new file name> is a <filename>
```

EXAMPLES

```
RENAME "temp","my'program"
RENAME "show.l","final'show.l"
```

SAMPLE EXERCISE

```
DIM oldname$ OF 20, newname$ OF 20
PAGE // clear screen
PRINT "Rename a file"
INPUT "Current file name: ": oldname$
INPUT "New name for file: ": newname$
RENAME oldname$,newname$
PRINT "ALL DONE"

RUN
  (screen clears)
Rename a file
Current file name: TEST
New name for file: FINAL
  (file TEST is renamed to FINAL)
ALL DONE
```

SEE ALSO: COPY, DELETE, PASS

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Renumbers the program currently in the computer. Valid line numbers for a COMAL program are 1 through 9999. If renumbering any line yields a line number above 9999 version 2.00 will print an error message. Version 0.14 will renumber it with some lines numbered over the maximum of 9999. The lines over 9999 cannot be listed with the LIST or EDIT command, but will not be deleted. Renumber the program again with lower line numbers and the lines will reappear. Version 2.00 also allows renumbering only the last portion of the program, starting at whatever line you specify. All lines before the starting line you specified will remain unchanged.

SYNTAX

```
RENUM [[<sourcestart>;]<target start>][,<target increment>]

 <sourcestart> is a number from 1-9999
   lines from this number to program end are renumbered
   if omitted, all lines are renumbered
 <target start> is a number from 1 - 9999
   to be used as the first line number;
   if omitted, the default value is 10
 <target increment> is a number from 1 - 9999
   to be used as the increment between lines;
   if omitted, the default value is 10
```

EXAMPLES

```
COMMAND           RESULT
RENUM             renumbers to 10, 20, 30, ...
RENUM ,5          renumbers to 10, 15, 20, ...
RENUM 9000,2      renumbers to 9000, 9002, 9004, ...
RENUM 500         renumbers to 500, 510, 520, ...
RENUM 100;1000  * renumbers lines 100-9999
                       as 1000, 1010, 1020, ...
```

* Version 2.00 only.

SAMPLE EXERCISE

```
10 PRINT "FIRST"
20 PRINT "SECOND"
30 PRINT "THIRD"

RENUM 9000

LIST
9000 PRINT "FIRST"
9010 PRINT "SECOND"
9020 PRINT "THIRD"
```

ADDITIONAL SAMPLE SEE: MERGE
SEE ALSO: AUTO, DEL, DISPLAY, EDIT, FIND, LIST

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Begins the REPEAT structure. Statements inside the structure are continually executed until the condition after the UNTIL evaluates to TRUE (a value not equal to 0). Since the statements are executed before the condition is evaluated, they will always be executed at least one time. A REPEAT loop may be used to read data from files or DATA statements. See Appendix A for a description of the REPEAT structure.

SYNTAX

```
REPEAT
```

EXAMPLE

```
REPEAT
```

SAMPLE PROGRAM

```
number:=RND(1,10)
REPEAT
  INPUT "GUESS MY NUMBER (FROM 1 TO 10): " : guess
  IF guess>number THEN PRINT "TOO HIGH"
  IF guess<number THEN PRINT "TOO LOW"
UNTIL guess=number
PRINT "RIGHT ON"

RUN
GUESS MY NUMBER (FROM 1 TO 10): 5
TOO HIGH
GUESS MY NUMBER (FROM 1 TO 10): 2
TOO LOW
GUESS MY NUMBER (FROM 1 TO 10): 3
RIGHT ON
```

ADDITIONAL SAMPLES SEE: AND, EOD, IN, SELECT OUTPUT
USED IN PROCEDURES: FETCH, GET'CHAR, GET'VALID, SHIFT
SEE ALSO: UNTIL

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Part of the ERROR HANDLER structure. If an ⟨error code⟩ is included, REPORT will cause an error with that value, otherwise the previous error is regenerated. This error can be handled by the ERROR HANDLER structure. If REPORT is issued while in a trapped section, it will go into the HANDLER section. If REPORT is issued while in a HANDLER section, then it will put you into an outer HANDLER if one exists, or back to the system. If REPORT is issued while not in an ERROR HANDLER structure, then the error is reported to the system. For more information on the ERROR HANDLER structure see Appendix A.

NOTE

SYNTAX

```
REPORT [<error code>[,<error text>]]

 <error code> is a <numeric expression>
   whose value is 0-32767
 <error text> is a <string expression>
```

EXAMPLES

```
REPORT
REPORT 900
REPORT user'error'num
```

SAMPLE PROGRAM

```
DIM name$ OF 20
nonsense:=500 // user error, not following directions
TRAP
  PAGE // clear screen
  TRAP
    INPUT "Name: ": name$
    IF name$="" THEN REPORT nonsense // generate an error
    INPUT "Age: ": age#
  HANDLER
    PRINT
    IF ERR=206 THEN // input error
      REPORT nonsense
    ELSE
      REPORT         this is a system error
    ENDIF
  ENDTRAP
  PRINT "Welcome to COMAL";name$
HANDLER
  IF ERR=nonsense THEN
    PRINT "You are not paying attention to directions"
  ELSE
    REPORT            let the system handle other errors
  ENDIF
ENDTRAP
END "ALL DONE"

RUN
Name:          just hit return here
You are not paying attention to directions
ALL DONE

RUN
Name: Mud
Age: Mud
You are not paying attention to directions
ALL DONE
```

```
RUN
Name: Simon
Age: 15
Welcome to COMAL Simon
ALL DONE
```

ADDITIONAL SAMPLES SEE: ERR, ERRTEXT$
SEE ALSO: ENDTRAP, ERR, ERRFILE, ERRTEXT$, HANDLER, TRAP

CATEGORY: Command / Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Allows data in the DATA statements to be reused. The pointer to the next data item is reset back to the first data item in the program, unless a ⟨label⟩ is specified, in which case the pointer will aim at the first data item following the specified label in the program (an error will result if the label is not immediately followed by a DATA statement).

NOTE

Version 0.14 does not allow a ⟨label⟩ to be used with RESTORE. It can only be used to RESTORE to the first data item.

SYNTAX

```
RESTORE [<label name>]

 <label name> is an <identifier>
```

EXAMPLES

```
RESTORE
RESTORE months
```

SAMPLE PROGRAM

```
DATA 1,3,2,1,4          this is the combination
PRINT "TRY TO OPEN THE SAFE -"
PRINT "ENTER ALL THE DIGITS CORRECTLY"
digit:=0                initialize
REPEAT
  digit:+1
  READ code             correct code for this digit
  IF digit=1 THEN PRINT "--------------------------------"
  PRINT digit;
  INPUT "- ENTER 1,2,3,4 (0 TO STOP): " : guess;
  IF guess=code THEN
    PRINT "RIGHT"
  ELSE
    PRINT "OOOPS"
    RESTORE             reinitialize
    digit:=0
  ENDIF
UNTIL EOD OR guess=0
IF EOD THEN PRINT "CONGRATULATIONS - YOU OPENED THE SAFE"
PRINT "ALL DONE"

RUN
TRY TO OPEN THE SAFE -
ENTER ALL THE DIGITS CORRECTLY
```

```
--------------------------------
1 - ENTER 1,2,3,4 (0 TO STOP): 1 RIGHT
2 - ENTER 1,2,3,4 (0 TO STOP): 2 OOOPS
--------------------------------
1 - ENTER 1,2,3,4 (0 TO STOP): 1 RIGHT
2 - ENTER 1,2,3,4 (0 TO STOP): 3 RIGHT
3 - ENTER 1,2,3,4 (0 TO STOP): 1 OOOPS
--------------------------------
1 - ENTER 1,2,3,4 (0 TO STOP): 1 RIGHT
2 - ENTER 1,2,3,4 (0 TO STOP): 3 RIGHT
3 - ENTER 1,2,3,4 (0 TO STOP): 2 RIGHT
4 - ENTER 1,2,3,4 (0 TO STOP): 1 RIGHT
5 - ENTER 1,2,3,4 (0 TO STOP): 4 RIGHT
CONGRATULATIONS - YOU OPENED THE SAFE
ALL DONE
```

ADDITIONAL SAMPLES SEE: DATA, EOD
SEE ALSO: DATA, EOD, LABEL, READ

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Assigns a value to the function and returns control back to the calling statement. It may also be used in a procedure to terminate the procedure as it skips to the ENDPROC statement and returns control back to its calling statement, but such use is rare and not recommended. When a function encounters a RETURN statement, the specified value after the keyword RETURN is taken as the value of the function, and the rest of the function is not executed. The function may be numeric, integer, or string. The RETURN statement can be part of the one-line IF structure.

NOTE

String functions are not supported by version 0.14.

SYNTAX

```
RETURN [<value>]

  <value> is an <expression>;
    it is omitted when used within a procedure
    it must be the same type as the function name
     (i.e., numeric, integer, or string)
```

EXAMPLES

```
RETURN 5
RETURN total DIV 100
```

SAMPLE PROGRAM

```
DIM type$(0:1) OF 4
type$(TRUE):="EVEN"; type$(FALSE):="ODD"
REPEAT
  INPUT "WHAT NUMBER (0 TO STOP): " : number;
  PRINT type$(even(number))      function EVEN is called
UNTIL number=0
PRINT "ALL DONE"
//
FUNC even(num)
  IF num MOD 2 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC even

RUN
WHAT NUMBER (0 TO STOP): 5 ODD
WHAT NUMBER (0 TO STOP): 8 EVEN
WHAT NUMBER (0 TO STOP): 0 EVEN
ALL DONE
```

ADDITIONAL SAMPLES SEE: FUNC, HANDLER
USED IN FUNCTIONS: EVEN, FILE'EXISTS, ROUND
SEE ALSO: CLOSED, EXTERNAL, FUNC, IMPORT, PROC, REF

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Returns a random number greater than or equal to 0 and less than 1 (0 <= =number<1). Or, if a start and end range is supplied, it will return a random integer within that range (inclusive). To be assured of truly random numbers, it is recommended to first execute the following statement before beginning:

Version 0.14 on PET or CBM:

```
dummy:=RND(-256*256*PEEK(141)+ 256*PEEK(142)+ PEEK(143))
```

Version 0.14 on Commodore 64:

```
dummy:=RND(-256*256*PEEK(160)+ 256*PEEK(161)+ PEEK(162))
```

Version 2.00:

```
RANDOMIZE
```

Then, to maintain the completely random sequence simply use the RND(n1,n2) when an integer between two numbers is needed, or if a decimal number between 0 and 1 is required, always use RND(1), since RND(0) and RND(-n) will not use the truly random sequence (version 2.00 does not have to worry about this, since it has no parameter for the random decimal number between 0 and 1).

NOTE

Version 2.00 changed the one-parameter RND statement to be without a parameter, and added the keyword RANDOMIZE to seed the random number generator.

SYNTAX

(between 0 and 1)

```
RND(<parameter>)            version 0.14
    or
RND                         version 2.00
```

<parameter> is a <numeric expression> used as the "seed"
use 0 to seed generator with time and return a number
use a negative number to seed a pseudo random sequence
and a parameter of 1 from then on
version 2.00 uses 1 by default and requires no parameter

(integer within a range)

```
RND(<start number>,<end number>)
```

<start number> and <end number> are <numeric expressions>
the integer returned will be between them (inclusive)
it uses the next number from the random sequence
previously seeded

EXAMPLES

```
Decimal number between 0 and 1:
RND(1)              version 0.14
RND                 version 2.00

Integer between two specified numbers:
RND(1,6)
RND(1,52)
```

SAMPLE PROGRAM

```
DIM roll$ OF 1
// RANDOMIZE //version 2.00 only may include this statement
PRINT "ROLL TWO DICE UNTIL YOU ROLL A SEVEN"
PRINT "-----------------------------------"
REPEAT
  INPUT "HIT [RETURN] TO ROLL THE DICE" : roll$
  dice1:=RND(1,6)
  dice2:=RND(1,6)
  PRINT "--- THE DICE ROLL WAS";dice1;dice2
UNTIL dice1+dice2=7
PRINT "THATS IT --- YOU JUST ROLLED A SEVEN"
PRINT "ALL DONE"
RUN
ROLL TWO DICE UNTIL YOU ROLL A SEVEN
-----------------------------------
HIT [RETURN] TO ROLL THE DICE
--- THE DICE ROLL WAS 2 6
HIT [RETURN] TO ROLL THE DICE
--- THE DICE ROLL WAS 1 4
HIT [RETURN] TO ROLL THE DICE
--- THE DICE ROLL WAS 3 3
HIT [RETURN] TO ROLL THE DICE
--- THE DICE ROLL WAS 2 5
THATS IT --- YOU JUST ROLLED A SEVEN
ALL DONE
```

ADDITIONAL SAMPLES SEE: ENDIF, ENDWHILE, IN, INTERRUPT, POKE, RANDOMIZE, REF, REPEAT, THEN, WHILE
SEE ALSO: INT, RANDOMIZE

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

The program currently in the computer is executed beginning with the first line. All variables are cleared prior to execution. Version 2.00 extends the RUN command to allow the program to be RUN to come from disk or tape.

NOTES

(1) Specifying a ⟨filename⟩ to be automatically LOADed and then RUN from tape or disk is not supported by version 0.14.
(2) In version 2.00, pressing the SHIFTed RUN key while the cursor is in the first position of any line will automatically issue a RUN "*" from the current unit. If the default unit is "cs:" then the LOAD and RUN will be from tape. Otherwise it will LOAD and RUN the first program from the current drive (default on startup is drive 0).

SYNTAX

```
RUN [<filename>]
```

EXAMPLE

```
RUN
RUN "DEMO3"                version 2.00 only
```

SAMPLE EXERCISE

```
10 PRINT "YOU GOT THIS LINE TO PRINT --- SO ..."
20 PRINT "... YOU KNOW HOW TO USE THE WORD RUN"

RUN
YOU GOT THIS LINE TO PRINT --- SO ...
... YOU KNOW HOW TO USE THE WORD RUN
```

ADDITIONAL SAMPLES SEE: MOST KEYWORDS
SEE ALSO: CHAIN, LOAD, SAVE, VERIFY

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Stores the program currently in the computer onto disk or tape in compressed (binary) form. Remark statements are not deleted. Later, the program can be retrieved via a LOAD, RUN, or CHAIN command.

NOTES

(1) SAVE to tape is not supported by version 0.14.
(2) The compressed forms of the programs as SAVEd will not be compatible between versions. To "transfer" a program to another version, first LIST it to disk, then ENTER it under the other version (see ENTER).
(3) In version 2.00 the SAVE command also SAVEs any packages LINKed to the program, together with the COMAL program. Then any future LOAD will automatically LOAD both the package and the program from the same file.
(4) SAVE is also used in version 2.00 to store an external procedure or function on disk. See Appendix A for details.

IMPORTANT NOTE

LIST to tape or disk is very useful. If you LIST a program to tape or disk, type NEW, and then ENTER it back again, you will clean up the COMAL NAME TABLE as well as get more free memory. SAVE and LOAD keeps the old NAME table. Each time you make a spelling mistake (such as LIT instead of LIST) that name goes into the table. Version 0.14 has only 255 names allowed, so if you make many spelling mistakes, you may get a TOO MANY NAMES error. Then you must perform this name table cleanup operation.

SYNTAX

```
SAVE <program name>

 <program name> is a <filename> and cannot be a variable
```

EXAMPLES

```
SAVE "TEMP1"
SAVE "1:SPECIAL"
```

SAMPLE EXERCISE

```
10 PRINT "THIS IS ONLY A SAMPLE"

CAT
0"SAMPLE DISK "      SD 2A
3   "LOADER"           PRG
661 BLOCKS FREE.

SAVE "SAMPLE'ONLY"
CAT
0"SAMPLE DISK "      SD 2A
3   "LOADER"           PRG
1   "SAMPLE'ONLY"      PRG
660 BLOCKS FREE.
```

ADDITIONAL SAMPLE SEE: MAIN
SEE ALSO: LOAD, RUN, VERIFY

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Does a prepass of the program currently in the computer, SCANning for correct program structures. Once a SCAN is done on the program, any procedure or function can be called from direct mode.

NOTES

(1) SCAN is not supported by version 0.14.
(2) An error message is generated for the first structure error found. Once this error is corrected, the SCAN command can be used again to continue SCANning the program structures.
(3) The RUN command always automatically performs a SCAN of the program before actually running it.

SYNTAX

```
SCAN
```

EXAMPLES

```
SCAN
```

SAMPLE EXERCISE (structure errors caught)

```
AUTO
0010 count:=0
0020 REPEAT
0030 count:+1
0040 PRINT "Count now is:";count
0050         hit STOP key here

SCAN
at 0040: "UNTIL" missing
0050 ENDWHILE                    add this line

SCAN
at 0050: "UNTIL" expected, not "ENDWHILE"
0050 UNTIL count>3               change to this

DISPLAY
count:=0
REPEAT
  count:+1
  PRINT "Count now is:";count
UNTIL count>2

SCAN                no structure errors found now
```

ADDITIONAL SAMPLE SEE: DISCARD
SEE ALSO: RUN

CATEGORY: Command / Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Allows you to select (choose) the output location. Recognized locations include "DS:" for Data Screen and "LP:" for Line Printer. Version 2.00 also recognizes a valid filename as a destination for the output. All printed output from a running program is directed to the output location selected via SELECT OUTPUT. While in interactive mode, "DS:" is the initial location for all output. After a LIST command, output returns to the screen (LIST issues a SELECT OUTPUT "DS:" when finished). The word OUTPUT is optional and will be supplied by COMAL if omitted.

Version 2.00 extends SELECT OUTPUT to a more general usage, allowing disk files to be used for printed output, as well as both the screen and printer.

NOTES

(1) INPUT prompts and error messages are directed to the screen even if a SELECT OUTPUT "LP:" has been issued.
(2) A disk or tape file as a SELECT OUTPUT location is not supported by version 0.14.

SYNTAX

```
SELECT [OUTPUT] <type>
 <type> is a <filename> representing the output location
   <filename> can be tape or disk file name or one of these
   predefined locations:  "DS:" for Data Screen
                          "LP:" for Line Printer
```

EXAMPLES

```
SELECT OUTPUT "DS:"
SELECT OUTPUT op$
SELECT OUTPUT "0:ADDRESSES"      version 2.00 only
```

SAMPLE PROGRAM

```
DIM name$ OF 20, reply$ OF 1
DATA "JIM","SUE","FRED","RHIANON","CHARLY","STEVE"
PRINT "REPLY 'Y' TO PRINT NAME ON PRINTER"
REPEAT
  READ name$
  INPUT name$ ; reply$
  IF reply$="Y" THEN
    SELECT OUTPUT "LP:"          output to printer
    PRINT name$
  ENDIF
  SELECT OUTPUT "DS:"            output to screen
UNTIL EOD
SELECT OUTPUT "DS:"              output back to screen
PASS "I0"                          reset C64 serial IEEE
PRINT "ALL DONE"

RUN
REPLY 'Y' TO PRINT NAME ON PRINTER
JIMN
SUEY       (SUE is printed on printer)
FREDY      (FRED is printed on printer)
RHIANONY   (RHIANON is printed on printer)
CHARLYN
STEVEN
ALL DONE
```

ADDITIONAL SAMPLES SEE: CAT, COPY, PAGE
SEE ALSO: PRINT

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns a -1 if the number specified is negative. Returns a 0 if the number specified is 0. Returns a 1 if the number specified is positive. This is one way to convert any number into one of three values.

NOTE

This is useful to convert a range of nonnegative values to a TRUE or FALSE value.

SYNTAX

```
SGN(<numeric expression>)
```

EXAMPLES

```
SGN(-32)
SGN(number)
```

SAMPLE PROGRAM

```
DIM sign$(-1:1) OF 10
DATA "NEGATIVE","ZERO","POSITIVE"
FOR temp:=-1 TO 1 DO READ sign$(temp)
REPEAT
  INPUT "NUMBER (0 TO STOP): " : number
  PRINT "THE SGN IS:";SGN(number);sign$(SGN(number))
UNTIL number=0
PRINT "ALL DONE"

RUN
NUMBER (0 TO STOP): -43
THE SGN IS: -1 NEGATIVE
NUMBER (0 TO STOP): 235
THE SGN IS: 1 POSITIVE
NUMBER (0 TO STOP): -5026.41
THE SGN IS: -1 NEGATIVE
NUMBER (0 TO STOP): 0
THE SGN IS:  0 ZERO
ALL DONE
```

SEE ALSO: ABS, VAL

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the sine of the specified number in radians. One radian equals approximately 57 degrees. The following formulae may be used for radian/degree conversion:

degrees = radians * (180/π) radians = degrees * (π/180)
degrees = radians * 57.2957795 radians = degrees * .0174532925

SYNTAX

```
SIN(<numeric expression>)
```

EXAMPLES

```
SIN(num)
SIN(22)
```

SAMPLE EXERCISE

```
REPEAT
  INPUT "ENTER ANGLE IN RADIANS (0=STOP): " : num
  IF num THEN PRINT "THE SINE OF";num;"IS";SIN(num)
UNTIL num=0
PRINT "ALL DONE"

RUN
ENTER ANGLE IN RADIANS (0=STOP): 5
THE SINE OF 5 IS -.958924274
ENTER ANGLE IN RADIANS (0=STOP): 25
THE SINE OF 25 IS -.132351746
ENTER ANGLE IN RADIANS (0=STOP): 0
ALL DONE
```

SEE ALSO: ATN, COS, TAN

CATEGORY: Command
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Prints the amount of free memory (in bytes) left in the system. Bytes used for the program and its variables are not considered free. Version 2.00 returns the size of the DATA and PROGRAM as well as the amount of FREE memory.

NOTES

(1) SIZE cannot be used as part of a program, only as a direct command.
(2) The number of free bytes as returned by SIZE cannot be assigned to a variable.
(3) The expanded SIZE information is not supported by version 0.14.

SYNTAX

```
SIZE
```

EXAMPLE

```
SIZE
```

SAMPLE EXERCISE (version 0.14)

```
SIZE

3569 BYTES FREE.
```

SAMPLE EXERCISE (version 2.00)

```
SIZE

PROG   DATA   FREE
00000  00000  30500

10 A:=10
SIZE

PROG   DATA   FREE
00008  00007  30485

RUN
END AT 0010
SIZE

PROG   DATA   FREE
00008  00012  30480
```

SEE ALSO: DIM

CATEGORY: Function
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the number of spaces specified. These spaces may be printed or may be used as part of an assignment statement.

NOTES

(1) SPC$ is not supported by version 0.14.
(2) Specifying a negative number will result in an error.

SYNTAX

```
SPC$(<number of spaces>)

 <number of spaces> is a non-negative <numeric expression>
```

EXAMPLES

```
SPC$(6)
SPC$(skip)
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "HOW MUCH SPACE BETWEEN WORDS (0 TO STOP): ": space
  PRINT "         1    1    2    2    3    3"
  PRINT "----5----0----5----0----5----0----5----"
  PRINT "YES",SPC$(space),"NO",SPC$(space),"MAYBE"
UNTIL NOT space
PRINT "ALL DONE"

RUN
HOW MUCH SPACE BETWEEN WORDS (0 TO STOP): 5
         1    1    2    2    3    3
----5----0----5----0----5----0----5----
YES     NO     MAYBE
HOW MUCH SPACE BETWEEN WORDS (0 TO STOP): 9
         1    1    2    2    3    3
---- 5----0----5----0----5----0----5----
YES          NO          MAYBE
HOW MUCH SPACE BETWEEN WORDS (0 TO STOP): 13
         1    1    2    2    3    3
----5----0----5----0----5----0----5----
YES              NO              MAYBE
HOW MUCH SPACE BETWEEN WORDS (0 TO STOP): 0
         1    1    2    2    3    3
----5----0----5----0----5----0----5----
YESNOMAYBE
ALL DONE
```

ADDITIONAL SAMPLES SEE: INTERRUPT, TIME
SEE ALSO: PRINT, STR$, TAB, ZONE

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the square root of the number specified. Specifying a negative number will result in an error.

SYNTAX

```
SQR(<number>)

 <number> is a non-negative <numeric expression>
```

EXAMPLES

```
SQR(23)
SQR(num)
```

SAMPLE PROGRAM

```
number'total:=0; sqr'total:=0; count:=0      initialize
REPEAT
  INPUT "NUMBER (0 TO STOP): " : number
  IF number>0 THEN
    number'total:+number            add NUMBER to TOTAL
    count:+1                        increment COUNT
    PRINT "ITS SQUARE ROOT IS";SQR(number)
    sqr'total:+SQR(number)        add square root to total
  ENDIF
UNTIL number=0
PRINT count;"NUMBERS WERE ENTERED"
PRINT "THEY TOTALED";number'total
PRINT "THE TOTAL OF THEIR SQUARE ROOTS IS";sqr'total
PRINT "THE SQUARE ROOT OF THEIR TOTAL IS";SQR(number'total)
PRINT "SQUARE ROOT OF THEIR AVG IS";SQR(number'total/count)
PRINT "ALL DONE"

RUN
NUMBER (0 TO STOP): 5
ITS SQUARE ROOT IS 2.3606798
NUMBER (0 TO STOP): 25
ITS SQUARE ROOT IS 5
NUMBER (0 TO STOP): 0
2 NUMBERS WERE ENTERED
THEY TOTALED 30
THE TOTAL OF THEIR SQUARE ROOTS IS 7.23606798
THE SQUARE ROOT OF THEIR TOTAL IS 5.47722558
SQUARE ROOT OF THEIR AVG IS 3.87298335
ALL DONE
```

SEE ALSO: EXP

CATEGORY: Command / Function
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

Displays the disk status and resets the disk error indicator. If a file number is specified, the status of the last disk operation with that file will be returned. STATUS$ is similar to DS$ in Commodore BASIC. Version 2.00 automatically checks the disk status after LOAD, SAVE, LIST, ENTER, CAT, CHAIN, and VERIFY and if the status is not OK then an error message is printed.

NOTES

(1) The statement STATUS is converted to PRINT STATUS$ by the system.
(2) STATUS$ may be used as a string function. It may be used in an assignment statement or may be compared to another string.
(3) Disk Error messages are included in Appendix G.
(4) Version 2.00 should use the ERROR HANDLER structure rather than STATUS.

SYNTAX

```
STATUS$
```

EXAMPLES

```
a$:=STATUS$(3:10)          version 2.00 only
PRINT STATUS$
```

SAMPLE PROGRAM

```
PRINT "THE CURRENT DISK STATUS IS";
PRINT STATUS$
OPEN FILE 6,"0:WRONGNAME",READ
PRINT "THE DISK STATUS NOW IS"
PRINT STATUS$
CLOSE
PRINT "ALL DONE"

RUN
THE CURRENT DISK STATUS IS 00, OK,00,00
THE DISK STATUS NOW IS
62, FILE NOT FOUND,00,00
ALL DONE
```

SEE ALSO: EOF, OPEN, VERIFY

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

As part of the FOR structure, STEP sets the amount that the ⟨control variable⟩ is incremented after each time through the loop. The STEP value is 1 if not specified otherwise. STEP can be positive or negative, real or integer (i.e., -2.5 is a valid step amount). See Appendix A for a description of the FOR structure.

SYNTAX

(One line)

```
FOR <var>:=<start> TO <end> [STEP <step>] DO <statement>
```

(Multi-line)

```
FOR <controlvar>:=<start> TO <end> [STEP <step>] [DO]
```

```
 <var> is a <control variable>
 <control variable> is a <numeric variable name>
 <start> is a <numeric expression>
 <end> is a <numeric expression>
 <step> is a <numeric expression>;
   if omitted, the default value is 1
```

EXAMPLES

```
FOR count:=4 TO max STEP 5 DO
FOR card:=52 TO 1 STEP -1 DO
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "WHAT NUMBER TO COUNT BY (0 TO STOP): " : numb
  IF numb THEN
    FOR count:=numb TO numb*6 STEP numb DO
      PRINT count;
    ENDFOR count
    PRINT "HOW WAS THAT!"
  ENDIF
UNTIL numb=0
PRINT "ALL DONE"

RUN
WHAT NUMBER TO COUNT BY (0 TO STOP): 6
6 12 18 24 30 36 HOW WAS THAT!
WHAT NUMBER TO COUNT BY (0 TO STOP): -1
-1 -2 -3 -4 -5 -6 HOW WAS THAT!
WHAT NUMBER TO COUNT BY (0 TO STOP): .25
.25 .5 .75 1 1.25 1.5 HOW WAS THAT!
WHAT NUMBER TO COUNT BY (0 TO STOP): 0
ALL DONE
```

ADDITIONAL SAMPLE SEE: PROC
SEE ALSO: DO, ENDFOR, FOR, TO

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Terminates (halts) program execution and returns to interactive mode. A STOP statement may occur at any point in the program and there may be more than one STOP statement in a program. STOP and END both halt program execution. Program execution may be continued with the next statement in the program by using the CON command. Variables remain intact and may displayed and changed before continuing. Version 2.00 allows you to print your own message when the STOP is encountered. Unless the optional string message is used, this message is displayed when a STOP is encountered (0030 represents the line number where the STOP was encountered):

```
STOP AT 0030
```

If the program is stopped while it is executing an external procedure or function, the default STOP message will indicate the line number executing at the time in *that* external section. Commands such as LIST, SAVE, and FIND will be in effect LOCALly to this external section only. Thus you can edit the section and save it to disk if necessary. Then to get back to the main program simply issue the command MAIN.

NOTES

(1) ⟨message⟩ is not supported by version 0.14.
(2) External procedures are not supported by version 0.14.

SYNTAX

```
STOP [<message>]

  <message> is a <string expression>
```

EXAMPLES

```
STOP
STOP "TEMPORARY STOP HERE AT LINE 545"
```

SAMPLE EXERCISE

```
10 temp:=5
20 PRINT "TEMP IS NOW";temp
30 STOP
40 PRINT "YOU CHANGED TEMP TO";temp

RUN
TEMP IS NOW 5

STOP AT 0030

TEMP:=76

CON
YOU CHANGED TEMP TO 76
```

ADDITIONAL SAMPLE SEE: CON
SEE ALSO: CHAIN, CON, END, RUN

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [-] VERS 2.00 [*]

Converts a ⟨numeric expression⟩ into its string equivalent. Thus the number 567 becomes the string "567". Its companion function is VAL, which will take a string and convert it to its numeric equivalent. It works with decimals and negative numbers (i.e., -3.58 becomes "-3.58") as well as exponential notation.

NOTE

SYNTAX

```
STR$(<number>)

 <number> is <numeric expression>
```

EXAMPLES

```
STR$(-3.58)          converts it to "-3.58"
STR$(2+5)            converts it to "7"
STR$(score)
```

SAMPLE PROGRAM

```
PRINT "INPUT NUMBERS TO CONVERT TO A STRING"
REPEAT
  INPUT "NUMBER (0 TO STOP): " : number;
  PRINT STR$(number)
UNTIL number=0
PRINT "ALL DONE"

RUN
INPUT NUMBERS TO CONVERT TO A STRING
NUMBER (0 TO STOP): -56.23 -56.23
NUMBER (0 TO STOP): +56.23 56.23
NUMBER (0 TO STOP): 5ABC 5
NUMBER (0 TO STOP): -54.56E-5 -5.456E-04
NUMBER (0 TO STOP): 0 0
ALL DONE
```

ADDITIONAL SAMPLES SEE: DELETE, GET$
SEE ALSO: CHR$, ORD, SPC$, VAL

CATEGORY: Statement / Command
KERNAL: [NO] VERS 0.14 [*] VERS 2.00 [*]

Transfers control to an assembly language routine beginning at the specified ⟨memory address⟩. SYS may be used to execute routines you supply and place in the memory locations as needed, or may be used to jump into ROM operating system routines. SYS tends to be very machine dependent, i.e., not very portable between various COMAL computers.

SYNTAX

```
SYS <memory address>

 <memory address> is a <numeric expression>
   whose value is from 0-65535 (a valid memory address)
```

EXAMPLES

```
SYS jump
SYS 64790
```

SAMPLE EXERCISE

```
SYS 64790                    4.0 ROM CBM/PET models

RUN
  (screen clears and the computer resets
    back to BASIC mode of operation)
```

USED IN FUNCTION: DISK'GET
SEE ALSO: EXEC, INTERRUPT, POKE

CATEGORY: System function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Prints spaces up to the column specified. If the position is already past the one specified, it goes to the specified position on the next line. TAB is always part of a PRINT statement. An error results if a nonpositive column number is specified.

NOTE

TAB prints spaces up to the specified column, not cursor rights as in Commodore BASIC.

SYNTAX

```
TAB(<column number>)

  <column number> is a positive <numeric expression>
```

EXAMPLES

```
TAB(24)
TAB(col)
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "WHAT COLUMN (0 TO STOP): " : col
  IF col THEN
    PRINT "         1    1    2    2    3    3    "
    PRINT "1---5----0----5----0----5----0----5----"
    PRINT TAB(col),"*"
  ENDIF
UNTIL col=0
PRINT "ALL DONE"

RUN
WHAT COLUMN (0 TO STOP): 5
         1    1    2    2    3    3
1---5----0----5----0----5----0----5----
    *
WHAT COLUMN (0 TO STOP): 9
         1    1    2    2    3    3
1---5----0----5----0----5----0----5----
        *
WHAT COLUMN (0 TO STOP): 2
         1    1    2    2    3    3
1---5----0----5----0----5----0----5----
 *
WHAT COLUMN (0 TO STOP): 37
         1    1    2    2    3    3
1---5----0----5----0----5----0----5----
                                   *
WHAT COLUMN (0 TO STOP): 0
ALL DONE
```

SEE ALSO: CURSOR, PRINT, USING, ZONE

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Returns the tangent of the specified number in radians. One radian equals approximately 57 degrees. The following formulae may be used for radian/degree conversion:

degrees = radians * (180/π) radians = degrees * (π/180)
degrees = radians * 57.2957795 radians = degrees * .0174532925

SYNTAX

```
TAN(<numeric expression>)
```

EXAMPLES

```
TAN(24)
TAN(num)
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "ENTER ANGLE IN RADIANS (0=STOP): " : num
  IF num THEN PRINT "THE TANGENT OF";num;"IS";TAN(num)
UNTIL num=0
PRINT "ALL DONE"

RUN
ENTER ANGLE IN RADIANS (0=STOP): 5
THE TANGENT OF 5 IS -3.380515
ENTER ANGLE IN RADIANS (0=STOP): 25
THE TANGENT OF 25 IS -.133526403
ENTER ANGLE IN RADIANS (0=STOP): 0
ALL DONE
```

SEE ALSO: ATN, COS, SIN

CATEGORY: Special
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Part of the IF structure. Statements following the THEN are only executed if the IF condition evaluates to TRUE (a value not equal to 0). If the condition evaluates to FALSE (a value of 0) the statements are skipped. It may be part of an IF or ELIF statement. The keyword THEN is optional, and if omitted will be supplied by the system. See Appendix A for a description of the IF structure.

THEN may also be used to extend the AND operator, styled after ADA. If the first ⟨condition⟩ is FALSE, then the second ⟨condition⟩ is not evaluated, since the result of the operation will be FALSE regardless of whether or not the second ⟨condition⟩ is FALSE. In addition to providing faster program execution, this also can prevent evaluation errors in the second ⟨condition⟩ by allowing the first condition to be a test for valid values.

SYNTAX

```
        (multi-line IF structure)

IF <condition> [THEN]
  <statements>

        (single line IF)

IF <condition> THEN <statement>

 <condition> is a <numeric expression>

        (conditional AND)

<condition> AND THEN <condition>
```

EXAMPLE

```
IF guess=num THEN PRINT "WINNER"
WHILE number<>0 AND THEN total/number>10 DO
```

SAMPLE PROGRAM

```
DIM type$ OF 1
total:=0
PRINT "HIT + FOR ADD, - FOR SUBTRACT, X TO EXIT"
REPEAT
  numb1:=RND(1,9);numb2:=RND(1,9)
  PRINT numb1;
  INPUT "": type$;       the null prompt avoids a "?" prompt
  IF "."+type$+"." IN ".+.-." THEN
    PRINT numb2;"=";
    IF type$="+" THEN
      PRINT numb1+numb2
    ELSE
      PRINT numb1-numb2
    ENDIF
  ELIF type$<>"X" THEN
    PRINT "ENTER + OR -, OR X TO EXIT"
  ENDIF
UNTIL type$="X"
PRINT "ALL DONE"

RUN
HIT + FOR ADD, - FOR SUBTRACT, X TO EXIT
5 + 2 = 7
1 + 3 = 4
8 - 9 = -1
4 0 ENTER + OR -, OR X TO EXIT
2 X ALL DONE
```

ADDITIONAL SAMPLES SEE: EXP, EXTERNAL, MOD, PEEK, RANDOM
USED IN PROCEDURES: FETCH, LOWER'TO'UPPER, SCANKEY
SEE ALSO: AND, ELIF, ELSE, ENDIF, IF

CATEGORY: Function / Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Returns the TIME of day from the real time clock in jiffies (60 jiffies in 1 second). The TIME is returned as an integer ranging from 0 to 5184000. When the time exceeds 5184000 (5,184,000 jiffies equal 1 day) it automatically is reset to 0. To set the TIME, include a ⟨numeric expression⟩ after the keyword TIME (i.e., TIME 0).

NOTE

TIME is not supported by version 0.14. See function JIFFIES in Appendix D for a user-defined function that is similar.

SYNTAX

```
TIME                          used as a function
  or
TIME <set time>               used as a statement

  <set time> is a non-negative <numeric expression>
    whose value is from 0-5184000
```

EXAMPLES

```
TIME
TIME temp
TIME 6000
```

SAMPLE PROGRAM

```
PAGE
REPEAT
  CURSOR 12,35
  sec:=TIME DIV 60
  PRINT sec;"SECONDS";SPC$(7)
UNTIL FALSE                    forever

RUN
  (screen clears)
489230 SECONDS          (printed in center of screen)
  (time is updated continually in same location)
```

SEE ALSO: ZONE

CATEGORY: Special
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Separates the ⟨start⟩ from the ⟨end⟩ in the FOR structure. It indicates that the loop will begin with the ⟨start⟩ value, incrementing each pass by the amount specified by the ⟨step⟩ value until the ⟨end⟩ value is exceeded. See Appendix A for a description of the FOR structure. TO also specifies that a LIST is going TO a file.

NOTE

Version 0.14 does not support the use of TO with LIST.

SYNTAX

```
FOR <var>:=<start> TO <end> [STEP <step>] DO <statement>
  or
FOR <controlvar>:=<start> TO <end> [STEP <step>] [DO]
  or
LIST [<range>] TO <filename>[,<device>]

 <var> is a <controlvariable>
 <controlvariable> is a <numeric variable name>
 <start> is a <numeric expression>
 <end> is a <numeric expression>
 <step> is a <numeric expression>;
   if omitted, the default value is 1
 <range> is represented by
   <procname> or
   <funcname> or
   [<start line>][-][<end line>]
     <start line> is a number from 1-9999
     <end line> is a number from 1-9999
```

EXAMPLES

```
FOR spaces:=1 TO 40 DO PRINT " ",
FOR temp:=count TO max STEP 5 DO
LIST TO "0:MYPROGRAM.L"
```

SAMPLE PROGRAM

```
INPUT "HOW MANY SCORES: " : number'scores
total:=0
FOR temp:=1 TO number'scores DO
  INPUT "SCORE: " : score
  total:+score                        add SCORE to TOTAL
ENDFOR temp
PRINT "TOTAL WAS";total;"AVERAGE WAS";total/number'scores

RUN
HOW MANY SCORES: 3
SCORE: 80
SCORE: 75
SCORE: 91
TOTAL WAS 246 AVERAGE WAS 82
```

ADDITIONAL SAMPLES SEE: EXEC, OR, ORD, PEEK, READ
USED IN PROCEDURES: CURSOR, LOWER'TO'UPPER
SEE ALSO: COPY, DO, ENDFOR, FOR, RENAME, STEP

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [+] VERS 2.00 [*]

Allows a COMAL program to disable the STOP key. Use TRAP statements to change the effect of the STOP key. TRAP ESC+ means to enable the STOP key, which is how the system starts. While the STOP key is enabled, hitting the STOP key will stop the program. To disable the STOP key, the statement TRAP ESC- is used. Now when the STOP key is pressed the program is not stopped, but the value of the system variable ESC is set to TRUE (a value of 1). A program can test ESC to watch for the pressing of the STOP key. ESC is set to FALSE (a value of 0) whenever the STOP key is not depressed.

TRAP also can mark the beginning of the error handler structure. The statements between the keyword TRAP and HANDLER will be considered TRAPped, and if any error occurs while executing them, the statements following the keyword HANDLER will be executed. See Appendix A for a description of the TRAP structure.

NOTES

(1) While the STOP key is enabled (TRAP ESC+ in effect) the value of ESC will always be FALSE (a value of 0).
(2) In version 0.14 while the STOP key is disabled (TRAP ESC- in effect) the value of ESC will be TRUE (a value of 1) *only* while the STOP key is depressed. Once the key is let up the value of ESC returns to FALSE (a value of 0). In version 2.00, once the value of ESC is set to TRUE, it remains TRUE until the running program checks its value, at which time it is evaluated again.
(3) TRAP as part of the error handler structure is not supported by version 0.14.

SYNTAX

```
TRAP                    start of error handler structure
 or
TRAP ESC<type>          disable / enable STOP key

 <type> is one of two symbols, + or -
   +       enable the STOP key
   -       disable the STOP key
```

EXAMPLES

```
STATEMENT          RESULT
TRAP ESC+          enable the STOP key
TRAP ESC-          disable the STOP key
TRAP               start of error handler structure
```

SAMPLE PROGRAM

```
TRAP ESC-                    disable the STOP key
PRINT "HIT THE STOP KEY PLEASE"
REPEAT                       waste time until STOP is hit
UNTIL ESC                 ESC equals FALSE until STOP is hit
PRINT "THANK YOU"
TRAP ESC+                    enable the STOP key again

RUN
  (STOP key is disabled)
HIT THE STOP KEY PLEASE
  (nothing happens until you hit the STOP key)
THANK YOU
  (STOP key is enabled again)
```

ADDITIONAL SAMPLES SEE: ENDTRAP, ESC, HANDLER, KEY$, REPORT
USED IN FUNCTION: FILE'EXISTS
SEE ALSO: ENDTRAP, ERR, ERRFILE, ERRTEXT$, ESC, REPORT, SETEXEC

CATEGORY: System constant
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

A predefined constant that always is equal to 1 (it can't be changed). It may be used as a numeric expression (i.e., TYPE$(TRUE) has the same meaning as TYPE$(1)).

SYNTAX

```
TRUE
```

EXAMPLE

```
TRUE
```

SAMPLE PROGRAM

```
done:=FALSE
total:=0
REPEAT
  INPUT "NUMBER (0 TO STOP): " : number
  total:+number                          add NUMBER to TOTAL
  IF number=0 THEN done:=TRUE
UNTIL done
PRINT "TOTAL WAS";total

RUN
NUMBER (0 TO STOP): 4
NUMBER (0 TO STOP): 8
NUMBER (0 TO STOP): 9
NUMBER (0 TO STOP): 1
NUMBER (0 TO STOP): 0
TOTAL WAS 22
```

ADDITIONAL SAMPLES SEE: ENDWHILE, FALSE, NOT, OF, RETURN
USED IN PROCEDURE: FETCH
SEE ALSO: FALSE

CATEGORY: System function / Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Specifies the current default unit to be used for the file names that do not specify a unit. It is optional and defaults to "0:" for disk drive 0 upon power up.

In version 2.00 the unit and secondary address information are part of the file name, and not specified separately. See Appendix N for more information about this.

NOTE

Version 0.14 does not support the UNIT$ function. It does have limited UNIT specifications as shown on the OPEN keyword page and in Appendix N.

SYNTAX

```
UNIT <unit specifier>      sets the current unit
 or
UNIT$                      returns the current unit

 <unit specifier> is a <string expression>
  evaluating to one of the following:
  kb:           keyboard
  ds:           data screen
  lp:           line printer
  sp:           serial port
  cs:           cassette
  u<device>:    device number
  <drive>:      disk drive represented by:

    <device> is a number from 0-31
    <drive> is represented by:
      [@]<drive#>:
        <drive#> is a number from 0-15
        @ is optional, means the file should be overwritten
          if it already exists on the disk
```

EXAMPLES

```
PRINT UNIT$                    prints the current unit
IF UNIT$<>"ds:" THEN
UNIT "cs:"                     set default unit to cassette
```

UNIT$

SAMPLE EXERCISE

```
UNIT "cs:"              set cassette as default
PRINT UNIT$             print the default unit
cs:
LOAD "*"                load the next program on tape
RUN                     program now runs
```

SEE ALSO: ENTER,OPEN

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Marks the end of the REPEAT structure and presents a ⟨condition⟩ for continued execution of the REPEAT loop. The statements in the loop will continue to execute until the ⟨condition⟩ evaluates to TRUE (a value not equal to 0). See Appendix A for a description of the REPEAT structure.

SYNTAX

```
UNTIL <condition>

 <condition> is a <numeric expression>
```

EXAMPLES

```
UNTIL EOD
UNTIL item$="DONE"
```

SAMPLE PROGRAM

```
total:=0
REPEAT
  INPUT "GO PAST 100 - ENTER NUMBER: " : number
  total:+number
UNTIL total>100
PRINT "YOU JUST BROKE 100"
PRINT "YOUR TOTAL WAS";total

RUN
GO PAST 100 - ENTER NUMBER: 24
GO PAST 100 - ENTER NUMBER: 43
GO PAST 100 - ENTER NUMBER: 3
GO PAST 100 - ENTER NUMBER: 19
GO PAST 100 - ENTER NUMBER: 33
YOU JUST BROKE 100
YOUR TOTAL WAS 122
```

ADDITIONAL SAMPLES SEE: AND, ENDIF, EOD, IN, SELECT OUTPUT
USED IN PROCEDURES: FETCH, GET'CHAR, GET'VALID, SHIFT
SEE ALSO: REPEAT

CATEGORY: Statement
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Combines a specified package with the currently running program. The package can be external. In addition to setting up pointers to the machine language package, it automatically places the procedure and function names in the package into the COMAL program's name table. This makes it very easy to use the machine language. See Appendix K and Appendix P for more information about machine language and USE.

NOTE

SYNTAX

```
USE <package name>

 <package name> is an <identifier>
```

EXAMPLES

```
USE graphics
USE class'pack
```

SAMPLE PROGRAM

```
USE bitpac
PRINT "This bit package allows bitwise AND and OR"
PRINT "128 OR 56 EQUALS";LOR(128,56)
END "End of quick demonstration of USE"

LINK "bitpac.p"       assumes package file named "bitpac.p"
RUN
This package allows bitwise AND and OR
128 OR 56 EQUALS 184
End of quick demonstration of USE
```

ADDITIONAL SAMPLE SEE: DISCARD
SEE ALSO: LINK

USING

CATEGORY: Special
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Allows formatted output. A ⟨string expression⟩ identifies the format to use. Within this ⟨string expression⟩ # reserves a position for each possible digit of the numeric variables' value. Period (.) is placed at the correct decimal location (and must have at least one # on both sides). Minus (-) may be used prior to the first # to reserve space for the minus sign (it is a floating minus sign). Zeroes (0) are padded where needed to the right of the decimal. Blanks on the left of the decimal are left as blanks. All other characters (except #. -) are printed as supplied. If the number has more digits than reserved, a "*" is printed in place of each reserved digit. Add the optional AT section of the PRINT statement, and the printing can begin at any spot on the screen.

NOTE

USING is not supported by Version 0.14 for the PET/CBM.

SYNTAX

```
PRINT [AT <row>,<col>:] USING <format string> : <varlist>

 <row> is a <numeric expression> whose value is 0-25
   (0 means current row)
 <col> is a <numeric expression> whose value is 0-80
   (0-40 on 40 column screens)
   (0 means current row)
 <format string> is a <string expression>
   # reserves a digit place
   . specifies the location of the decimal point
   - floating minus sign is an option
 <varlist> can be one or more numeric variables separated
   by commas
```

EXAMPLES

```
PRINT USING "$###.##" : cost
PRINT USING money$ : price
```

USING

SAMPLE PROGRAM 1

```
DIM money$ OF 40
money$:="IN DOLLARS UP TO 999.99 IT IS: $###.##"
REPEAT
  INPUT "ENTER AMOUNT (0 TO STOP): " : amount
  PRINT USING money$ : amount
UNTIL amount=0
PRINT "ALL DONE"

RUN
ENTER AMOUNT (0 TO STOP): 25
IN DOLLARS UP TO 999.99 IT IS: $ 25.00
ENTER AMOUNT (0 TO STOP): 4.5
IN DOLLARS UP TO 999.99 IT IS: $  4.50
ENTER AMOUNT (0 TO STOP): 985621
IN DOLLARS UP TO 999.99 IT IS: $******
ENTER AMOUNT (0 TO STOP): 0
IN DOLLARS UP TO 999.99 IT IS: $  0.00
ALL DONE
```

USING

SAMPLE PROGRAM 2

```
DIM company$ of 20, item$ of 20
WHILE NOT EOD DO
  READ company$,amount,item$,num
  company$(1:20):=company$ // pad spaces to right
  item$(1:20):=item$ // pad spaces to right
  PRINT USING company$+" $###.## "+item$+" ###": amount,num
ENDWHILE
PRINT "All done."
DATA "Swifty Sales",45.13,"Tool Box",6
DATA "Joes Diner",20,"Deluxe Tool Set",1
DATA "Super Saver Inc.",474.1,"Nails & Bolts",846

RUN
Swifty Sales        $ 45.13  Tool Box            6
Joes Diner          $ 20.00  Deluxe Tool Set      1
Super Saver Inc.    $474.10  Nails & Bolts      846
All done.
```

SEE ALSO: AT, PRINT, TAB, ZONE

CATEGORY: Function
KERNAL: [YES] VERS 0.14 [-] VERS 2.00 [*]

Returns the numeric value of a string of digits. It is possible to use string variables in your INPUT statements, and with VAL, convert them to numeric after validating them if need be. The VAL function will accept the ten digits (1234567890), positive and negative signs (+ -), decimal point (.), and exponential notation (unshifted E). Real numbers such as "-4.265" can be used, as well as exponential notation (i.e., 1.04E-04). If a nonvalid character is encountered in the string, it and the rest of the string will be ignored, unless it is the first character, in which case an error will result.

NOTES

(1) VAL is not supported by version 0.14. It can be simulated by the function VALUE listed in Appendix D.
(2) In addition to the ten digits (0,1,2,3,4,5,6,7,8,9), VAL will accept a positive or negative sign (+ or -), a decimal point (.), and the exponential notation (unshifted E).
(3) Version 2.00 considers it to be an error to try to take the value of a nonnumeric string. This is because an error handler is available, allowing the program to take whatever actions are necessary to handle the situation in each individual case.
(4) Version 2.00 allows the string to be a hexadecimal number, such as "$FF" or a binary number such as "%10001010". See Appendix M for more information.

SYNTAX

```
VAL(<number string>)

 <number string> is a <string expression>
```

EXAMPLES

```
VAL(amount$)
VAL("1.5E20")              becomes 1.5E+20
VAL("-3.5")                becomes -3.5
VAL("ABC")                 yields a function argument error
VAL("")       null string -yields a function argument error
VAL("$FF")                 becomes 255
```

SAMPLE PROGRAM

```
DIM text$ OF 20
PRINT "ENTER SOME NUMBERS TO TEST THE VAL FUNCTION"
REPEAT
  INPUT "NUMBER (0 TO STOP): " : text$;
  PRINT VAL(text$)
UNTIL text$="0"
PRINT "ALL DONE"

RUN
ENTER SOME NUMBERS TO TEST THE VAL FUNCTION
NUMBER (0 TO STOP): +123 123
NUMBER (0 TO STOP): 1.5E20 1.5E+20
NUMBER (0 TO STOP): -3.5 -3.5
NUMBER (0 TO STOP): ABC
AT 0050: NUMERIC CONSTANT EXPECTED
```

SEE ALSO: CHR$, INT, ORD, STR$

CATEGORY: Command
KERNAL: [NO] VERS 0.14 [-] VERS 2.00 [*]

Verifies that the file specified is identical to that in the computer. It is useful to verify that a program has been correctly SAVEd to disk or tape.

NOTES

(1) VERIFY is not supported by version 0.14.
(2) VERIFY cannot be used in conjunction with LISTing files to tape or disk.

SYNTAX

```
VERIFY <filename>
```

EXAMPLES

```
VERIFY name$
VERIFY "1:TEST"
```

SAMPLE EXERCISE

```
10 // MINE

SAVE "MY'PROGRAM"

VERIFY "MY'PROGRAM"        verify error will occur if not
                           an exact match
NEW

10 // YOURS

SAVE "YOUR'PROGRAM"

VERIFY "MY'PROGRAM"
VERIFY ERROR
  (we expected the error since line 10 was different)
```

SEE ALSO: LOAD, SAVE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Provides a specific case within the CASE structure. One or more valid values are listed after the WHEN separated by commas. These values must be of the same type as the CASE ⟨control expression⟩ (i.e., string or numeric). If any of these values match the current value of the CASE ⟨control expression⟩, that WHEN is TRUE and its following statements are executed. WHEN can also be used in version 2.00 as part of a conditional EXIT statement. See Appendix A for a description of the CASE and LOOP structures.

NOTE

EXIT WHEN is not supported by version 0.14.

SYNTAX

```
WHEN <list of values>
  <statements>
   and
EXIT WHEN <condition>

 <list of values> is series of either <numeric expressions>
    or <string expressions>
   if more than one is used, a comma is placed between them
   each must be of same type as CASE <control expression>
 <condition> is a <numeric expression>
```

EXAMPLE 1

```
WHEN "H","HELP","?"
  EXEC instructions
```

EXAMPLE 2

```
WHEN 5,6,7
  PRINT "YOU WIN"
```

EXAMPLE 3

```
EXIT WHEN error>0
```

SAMPLE PROGRAM

```
DIM choice$ OF 1
REPEAT
  INPUT "ENTER A VOWEL (X TO EXIT): " : choice$;
  CASE choice$ OF
  WHEN "A","E","I","O","U","a","e","i","o","u"
    PRINT "RIGHT"
  WHEN "X","x"
    PRINT "EXIT"
  OTHERWISE
    PRINT "NOPE"
  ENDCASE
UNTIL choice$="X" or choice$="x"
PRINT "ALL DONE"

RUN
ENTER A VOWEL (X TO EXIT): P NOPE
ENTER A VOWEL (X TO EXIT): A RIGHT
ENTER A VOWEL (X TO EXIT): X EXIT
ALL DONE
```

ADDITIONAL SAMPLES SEE: CASE, CHAIN, ENDCASE, MOD, OTHERWISE, READ
USED IN PROCEDURE: FETCH
SEE ALSO: CASE, ENDCASE, OF, OTHERWISE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Marks the start of the WHILE structure. The statements within the WHILE loop are executed only if the ⟨comparison⟩ is TRUE (a value not equal to 0). Thus it is possible for the ⟨statements⟩ to be skipped and not executed at all. See Appendix A for a description of the WHILE structure.

SYNTAX

```
      (one line)

WHILE <condition> DO <statement>

      (multi-line)

WHILE <condition> [DO]
  <statements>

  <condition> is an <expression>
```

EXAMPLES

```
WHILE reply$="" DO EXEC getproc
WHILE ok DO
WHILE NOT EOF(2) DO
```

SAMPLE PROGRAM

```
yes:=0; errors:=0; maxerrors:=1          initialize
WHILE errors<maxerrors DO
  numb1:=RND(1,98);numb2:=RND(numb1,99)      numb1 is larger
  PRINT "WHAT IS";numb2;"MINUS";numb1;
  INPUT ": " : answer;
  IF answer=numb2-numb1 THEN
    yes:+1                                 increment right count
    PRINT "YES"
  ELSE
    PRINT "NO"
    errors:+1
  ENDIF
ENDWHILE
PRINT "OOPS -";errors;"ERROR(S) AFTER";yes;"RIGHT"
PRINT "ALL DONE"

RUN
WHAT IS 45 MINUS 38 : 7 YES
WHAT IS 81 MINUS 50 : 31 YES
WHAT IS 33 MINUS 8 : 41 NO
OOPS - 1 ERROR(S) AFTER 2 RIGHT
ALL DONE
```

ADDITIONAL SAMPLES SEE: ENDWHILE, EOD, EOF, GET$, INPUT, NOT, WRITE
SEE ALSO: DO, ENDWHILE

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Specifies that data is to be output to the file specified. The file must previously have been opened with an OPEN statement using type WRITE or APPEND. It is written as a sequential binary file. Make sure to CLOSE the file when you are finished writing to it. A disk file must be closed properly before it can be RENAMEd, COPYed, BACKUPped (DUPLICATEd), or SCRATCHed. Attempting to SCRATCH an open file may result in disk errors. COLLECT (VALIDATE) will remove all improperly closed disk files. See your CBM disk manual for more information on these commands. Examples of how to use these commands from COMAL are presented with the keyword PASS and in Appendix O.

NOTES

(1) A file created by WRITE FILE must be read with READ FILE statements (INPUT FILE cannot be used since it expects an ASCII file, not binary as used by WRITE).
(2) See Appendix C and Appendix L for more information about sequential files.
(3) In version 2.00 the value of each element of an entire array may be written to a file using ony one statement. Just use the array name without the parentheses section:

```
DIM table(3,3)
FOR x:=1 TO 3 DO
  FOR y:=1 TO 3 DO
    TABLE(x,y):=x*y
  ENDFOR y
ENDFOR x
OPEN FILE 3,"0:TABLE'VALUES",WRITE
WRITE FILE 3: table  <-this line writes the entire array to disk
CLOSE FILE 3
```

SYNTAX

```
WRITE FILE <filenum> : <variable list>

  <variable list> is one or more variables
    numeric or string, separated by commas
```

EXAMPLES

```
WRITE FILE 2 : text$
WRITE FILE choice : name$,address$,city$,state$,zip
```

SAMPLE PROGRAM

```
DIM name$ OF 40
OPEN FILE 2,"WINNERS3",WRITE
PRINT "WE WILL NOW WRITE 3 WINNER NAMES TO DISK"
FOR temp:=1 TO 3 DO
  INPUT "WINNERS NAME: " : name$;
  WRITE FILE 2 : name$
  PRINT "*"
ENDFOR temp
CLOSE FILE 2
PRINT "ALL DONE"

RUN
  (file number 2 named WINNERS3 is opened for output)
WE WILL NOW WRITE 3 WINNER NAMES TO DISK
WINNERS NAME: STEVE *      the * appears after the name
WINNERS NAME: GEORGE *     is written to the file
WINNERS NAME: MARY *
 (each name was written to the file - it now is closed)
ALL DONE
```

ADDITIONAL SAMPLE SEE: CLOSE
SEE ALSO: CLOSE, FILE, OPEN, PRINT, READ

CATEGORY: Statement
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Specifies that data is to be output to the file specified. With a random access file, you can write to any record you wish at any time by specifying the record number. Random access files reserve a fixed length for each record. The actual record may be shorter, but the same amount of space is used. Random file manipulation takes more care and planning than manipulation of a sequential file. If you are new to programming, you may wish to master sequential file manipulation first. See Appendix L for more information about random WRITE files.

NOTE

In version 2.00 if you do not specify the record number, you will continue writing the same record started in a previous WRITE FILE statement.

SYNTAX

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

  <record number> is a positive <numeric expression>
   it must evaluate to a valid record number
   if omitted in version 2.00, the same record is continued
     unless the previous record was ended,
     then the next record is started
  <offset> is a positive <numeric expression>;
    if omitted, no bytes will be skipped
  <variable list> is one or more variables,
    numeric or string, separated by commas
```

EXAMPLES

```
WRITE FILE 2,5: test$
WRITE FILE choice,item: item'name$,price
```

SAMPLE PROGRAM 1

```
DIM text$ OF 80
OPEN FILE 7,"RANDOM'READ",RANDOM 80
PRINT "WRITE SOME RANDOM TEXT RECORDS"
REPEAT
  INPUT "WHAT RECORD NUMBER (0 TO STOP): " : number
  IF number THEN
    INPUT "TEXT:" : text$
    WRITE FILE 7,number: text$
  ENDIF
UNTIL number=0
CLOSE
PRINT "ALL DONE"

RUN
  (file 7 named RANDOM'READ is opened for random access)
WRITE SOME RANDOM TEXT RECORDS
WHAT RECORD NUMBER (0 TO STOP): 5
TEXT:THIS IS THE FIFTH RECORD
  (THIS IS THE FIFTH RECORD is written to file 7, record 5)
WHAT RECORD NUMBER (0 TO STOP): 9
TEXT:THIS IS THE NINTH RECORD
  (THIS IS THE NINTH RECORD is written to file 7, record 9)
WHAT RECORD NUMBER (0 TO STOP): 0
  (file 7 is closed)
ALL DONE
```

SAMPLE PROGRAM 2 (version 2.00 only)

```
DIM text$ of 20
OPEN FILE 2,"ran",RANDOM 50
WRITE FILE 2,20: 1/4 // first element
WRITE FILE 2: "mystring" // second element of record 20
WRITE FILE 2: 10+2 // third element of record 20
READ FILE 2,20: num // read element 1 of record 20
READ FILE 2: text$,sum // read elements 2 & 3 of record 20
CLOSE FILE 2
PRINT num;text$;sum
PRINT "All done."

RUN
 (Random file number 2 is opened)
 (Writes 3 fields into record 20)
 (Reads values for 3 variables from record 20)
.25 mystring 12
All done.
```

ADDITIONAL SAMPLE SEE: RANDOM
SEE ALSO: CLOSE, CREATE, FILE, OPEN, PRINT, READ

CATEGORY: Type of an OPEN
KERNAL: [YES] VERS 0.14 [+] VERS 2.00 [*]

Specifies that the file being opened is for OUTPUT. Data can be written to it using WRITE FILE or PRINT FILE statements. When used with tape or disk, a WRITE type of file will be a sequential file that can be read with either READ FILE or INPUT FILE statements. The word FILE is optional and if omitted will be supplied by the system.

NOTES

(1) Writing to tape is not supported by version 0.14.
(2) You may open a file to a printer by specifying its device number as UNIT ⟨device⟩ (i.e., UNIT 4) in version 0.14, or by specifying a file name of "lp:" in version 2.00. PRINT FILE statements to that file are then directed to the printer.
(3) See Appendix C for more information about a sequential file.

SYNTAX

```
OPEN [FILE] <filenum>,<filename>,WRITE
```

EXAMPLES

```
OPEN FILE 2,"0:TESTFILE",WRITE
OPEN FILE infile,filename$,WRITE
```

SAMPLE PROGRAM

```
DIM account$ OF 20
count:=0
PRINT "WE WILL NOW WRITE AN ACCOUNT NAME FILE"
PRINT "IT COULD BE USED LATER TO EXPAND CODED ACCOUNTS"
OPEN FILE 2,"0:ACCOUNT LIST",WRITE
WHILE NOT EOD DO
  READ account$
  count:+1
  WRITE FILE 2 : account$
  PRINT "ACCOUNT NUMBER";count;"-";account$
ENDWHILE
DATA "OFFICE SUPPLIES","POSTAGE AND SHIPPING"
DATA "PROFESSIONAL FEES","ADVERTISING"
CLOSE FILE 2
PRINT "ALL DONE"

RUN
WE WILL NOW WRITE AN ACCOUNT NAME FILE
IT COULD BE USED LATER TO EXPAND CODED ACCOUNTS
  (sequential disk file # 2 named ACCOUNT LIST is opened)
  (system writes OFFICE SUPPLIES to disk file number 2)
ACCOUNT NUMBER 1 - OFFICE SUPPLIES
  (system writes POSTAGE AND SHIPPING to disk file # 2)
ACCOUNT NUMBER 2 - POSTAGE AND SHIPPING
  (system writes PROFESSIONAL FEES to disk file number 2)
ACCOUNT NUMBER 3 - PROFESSIONAL FEES
  (system writes ADVERTISING to disk file number 2)
ACCOUNT NUMBER 4 - ADVERTISING
  (system closed file number 2)
ALL DONE
```

SEE ALSO: CLOSE, FILE, OPEN, READ

CATEGORY: Function / Statement
KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Sets the screen tab position interval. The default value upon power up, RUN, or NEW is 0 (a tab stop at every location). When a comma separates two items being printed, the second begins at the next tab position after the first item is printed. The first zone position is the beginning of the line (position 1). The next zone position is the previous zone position plus the value of ZONE. When the end of the line is reached, the next tab position is the first print position on the next line. ZONE is always global and its value applies even to CLOSED procedures and functions without the need of an IMPORT statement. The ZONE setting applies to the printer or while SELECT OUTPUT "LP:" is in effect as well as to any WRITE files. ZONE is reset to its default value of 0 when a NEW command is issued. ZONE cannot be a negative value. If it is set to a negative number, an error (ZONE VALUE INCORRECT) will result when a PRINT statement attempts to use the ZONE reference.

SYNTAX

```
ZONE                        used as a function
  or
ZONE <tab interval>         used as a statement

  <tab interval> is a non-negative <numeric expression>
    if omitted ZONE will be used as a function and return
     the current zone value.
```

EXAMPLES

```
ZONE 5
oldzone:=ZONE
```

SAMPLE PROGRAM 1

```
ZONE 4
startzone:=ZONE          remember current ZONE setting
PRINT "THE CURRENT ZONE SETTING IS";startzone
PRINT "NOW WE WILL SHOW YOU SOME INCREASING ZONE SETTINGS"
FOR temp:=2 TO 9 DO
  ZONE temp
  PRINT temp,"B","C","D"
ENDFOR temp
ZONE startzone           return zone to original setting
PRINT "THE ZONE SETTING IS NOW SET BACK TO";ZONE
PRINT "ALL DONE"

RUN
THE CURRENT ZONE SETTING IS 4
NOW WE WILL SHOW YOU SOME INCREASING ZONE SETTINGS
2 B C D
3  B  C  D
4   B   C   D
5    B    C    D
6     B     C     D
7      B      C      D
8       B       C       D
9        B        C        D
THE ZONE SETTING IS NOW SET BACK TO 4
ALL DONE
```

SAMPLE PROGRAM 2

```
ZONE 11
PRINT "THE TOP THREE PLAYERS IN EACH GAME"
PRINT
PRINT "GAME","FIRST","SECOND","THIRD"
PRINT "----","-----","------","-----"
PRINT 1,"RHIANON","SUE","PAM"
PRINT 2,"PAM","TOM","RICHARD"
PRINT 3,"SUE","TIMOTHY","RHIANON"
PRINT 4,"RICHARD","RHIANON","TIMOTHY"

RUN
THE TOP THREE PLAYERS IN EACH GAME

GAME        FIRST       SECOND      THIRD
----        -----       ------      -----
1           RHIANON     SUE         PAM
2           PAM         TOM         RICHARD
3           SUE         TIMOTHY     RHIANON
4           RICHARD     RHIANON     TIMOTHY
```

ADDITIONAL SAMPLES SEE: NOT, OR, ORD
USED IN PROCEDURES: CURSOR, FETCH, TAKE'IN
SEE ALSO: INPUT, PRINT, TAB, USING

APPENDIX A
THE COMAL STRUCTURES

The power of COMAL lies in its multiline structures, each made up of several specific KEYWORDS, and is emphasized by the automatic indenting of the block of statements within it. For more details on the structure, please refer to each of its KEYWORDS. The structures are summarized in this appendix for your quick reference. They are grouped as follows:

CONDITIONAL STATEMENT EXECUTION

```
IF ... THEN ... ELIF ... ELSE ... ENDIF
CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE
```

REPETITION

```
REPEAT ... UNTIL
WHILE ... DO ... ENDWHILE
FOR ... TO ... STEP ... DO ... ENDFOR
LOOP ... EXIT / EXIT WHEN ... ENDLOOP   version 2.00
```

MODULES

```
FUNC .. REF .. CLOSED .. EXTERNAL ... IMPORT ... RETURN ... ENDFUNC
PROC .. REF .. CLOSED .. EXTERNAL ... IMPORT ... RETURN ... ENDPROC
TRAP .. HANDLER .. ERR .. ERRFILE .. ERRTEXT$ .. REPORT .. ENDTRAP
```

IF Structure

The IF structure has five basic forms, depending upon the use of ELSE and ELIF. They are as follows:

(a) IF ... THEN ... (one line IF)
(b) IF ... THEN ... ENDIF
(c) IF ... THEN ... ELSE ... ENDIF
(d) IF ... THEN ... ELIF ... ENDIF
(e) IF ... THEN ... ELIF ... ELSE ... ENDIF

An IF structure is a decision structure. Some condition is evaluated to be either TRUE or FALSE. IF it is TRUE, one set of statements is executed. If it is FALSE and an ELSE section is included, its statements are executed. In addition, the keyword ELIF allows another condition to be evaluated if the previous condition fails. The five combinations of these KEYWORDS are summarized below:

(a) IF ... THEN ... one line IF

Often you will either want something or not. Nothing complicated, just a simple TRUE or FALSE. IF the condition evaluates to TRUE (a value not equal to 0), the statement following the THEN is executed. Otherwise, nothing happens, and program execution continues with the next statement. NOTE that ENDIF must not be used.

STRUCTURE FORM

```
IF <condition> THEN <statement>
```

EXAMPLE

```
IF lines=Ø THEN EXEC initialize
```

(b) IF ... THEN ... ENDIF

This is similar to the simple one-line IF above, but several lines of statements may be executed if the condition evaluates to TRUE (a value not equal to 0). However, IF the condition evaluates to FALSE (a value of 0), all these statements are skipped, and program execution continues after the ENDIF.

STRUCTURE FORM

```
IF <condition> THEN
  <statements>
ENDIF
```

EXAMPLE

```
IF errors=Ø THEN
  PRINT "SO FAR SO GOOD"
  PRINT "TRY ONE MORE"
ENDIF
```

(c) IF ... THEN ... ELSE ... ENDIF

Here, the ELSE section provides statements to be executed if the condition evaluates to FALSE (a value of 0). The statements prior to the ELSE, after the THEN, are executed if the condition evaluates to TRUE (a value not equal to

0). Either one set of statements or the other is executed, but never both. After the chosen set is executed, program execution continues following the ENDIF.

STRUCTURE FORM

```
IF <condition> THEN
  <true statements set>
ELSE
  <false statements set>
ENDIF
```

EXAMPLE

```
IF score>high THEN
  high:=score
  EXEC new'high
ELSE
  EXEC report'it
ENDIF
```

(d) IF ... THEN ... ELIF ... ENDIF

Here, the ELIF presents another condition to check if the previous condition evaluates to FALSE (a value of 0). As many ELIFs as needed can be combined into one IF structure. Each is checked only if the previous one evaluates to FALSE. If a condition evaluates to TRUE (a value not equal to 0), the statements immediately following that conditions THEN are executed. All other conditions and statements in the structure are then skipped, and program execution continues after the ENDIF. If none of the conditions evaluate to TRUE, then nothing happens and program execution continues after the ENDIF.

STRUCTURE FORM

```
IF <condition-1> THEN
  <statements-1>
ELIF <condition-2> THEN
  <statements-2>
{ELIF <condition-X> THEN
  <statements-X>}
ENDIF
```

EXAMPLE

```
IF reply$ IN "AEIOU" THEN
  EXEC vowel
ELIF REPLY$="Y" THEN
  EXEC y
ELIF REPLY$="HELP" THEN
  EXEC instructions
ENDIF
```

(e) IF ... THEN ... ELIF ... ELSE ... ENDIF

This structure behaves just like the previous IF structure (d), with the addition of the ELSE section. Now, if none of the ELIF conditions evaluate to TRUE (a value not equal to 0), the ELSE section statements are executed and then program execution continues following the ENDIF. These statements are skipped if any of the conditions evaluate to TRUE.

STRUCTURE FORM

```
IF <condition-1> THEN
  <statements-1>
ELIF <condition-2> THEN
  <statements-2>
{ELIF <condition-X> THEN
  <statements-X>}
ELSE
  <statements-else>
ENDIF
```

EXAMPLE

```
IF trys>9 THEN
  EXEC finished
ELIF errors>0 THEN
  EXEC missed
ELSE
  PRINT "TRY ONE MORE"
  EXEC more
ENDIF
```

CASE Structure

The CASE structure allows a multiple-choice decision to be made. The first line of this structure is the CASE ... OF. This line presents a value to be used for comparison to each of the specific WHEN cases that follow. As many WHEN sections as needed may be used, each with their own set of statements and condi-

tional execution expression list. After the CASE expression has been evaluated, it is compared with each of the expessions of the first WHEN section. If any of its expressions match, the statements of that WHEN section are executed, and program execution then continues after the ENDCASE, skipping the remainder of the CASE structure. One after another, each WHEN section's conditional expression list is compared to the CASE expression. If a match is found, that WHEN's statements are executed, and the remainder of the CASE structure is skipped, continuing program execution after the ENDCASE. If none of the WHEN section expressions provide a match, the OTHERWISE statements are executed, and program execution then continues after the ENDCASE. If the optional OTHERWISE section has not been included in the structure an error condition occurs.

STRUCTURE FORM

```
CASE <expression> OF
WHEN <expression list>
  <statements>
{WHEN <expression list>
  <statements>}
[OTHERWISE
  <statements>]
ENDCASE
```

EXAMPLE

```
CASE label'line OF
WHEN Ø
  EXEC header
WHEN 1,2,3,4
  PRINT customer$(label'line)
OTHERWISE
  EXEC next'label
ENDCASE
```

REPEAT Structure

This structure allows a block of statements to be executed repeatedly, as long as the final UNTIL condition is FALSE (a value of 0). Once the condition evaluates to TRUE (a value not equal to 0) program execution continues with the statement following the UNTIL statement. Since the statements are executed first, and the condition is checked afterward, the statements will always be executed at least once when the initial REPEAT is encountered.

STRUCTURE FORM

```
REPEAT
  <statements>
UNTIL <condition>
```

EXAMPLE

```
REPEAT
  EXEC another
  EXEC sort
UNTIL errors>Ø
```

WHILE Structure

This structure is similar to the REPEAT structure, in that it presents a block of statements to be conditionally executed over and over. However, its condition occurs in its initial WHILE statement, and it is evaluated prior to executing the statements. If it evaluates to TRUE (a value not equal to 0), the statements are executed, and it is then evaluated again. If it evaluates to FALSE (a value of 0), the block of statements are skipped, and program execution continues following the ENDWHILE statement.

STRUCTURE FORM

```
WHILE <condition> DO
  <statements>
ENDWHILE
```

EXAMPLE

```
WHILE NOT EOF(2) DO
  READ FILE 2: item$
  PRINT item$
ENDWHILE
```

A simple one-line version of the WHILE structure is also available. It must not use the keyword ENDWHILE.

STRUCTURE FORM

```
WHILE <condition> DO <statement>
```

EXAMPLE

```
WHILE done=FALSE DO EXEC ask
```

FOR Structure

The FOR structure again presents a block of statements for repeated execution, except they are to be executed a set number of times and to include a control variable. The initial FOR statement sets up the start value and end limit for the control variable, which is initialized to the start value when the structure is executed. The statements are executed if the control variable's value has not exceeded the end limit. After they are executed, the control variable is incremented by the STEP amount (if STEP is omitted, it is incremented by 1). The control variable is then again checked against the end limit. The statements continue to be executed as long as the limit is not exceeded. Once it is exceeded, the FOR loop is considered finished, and program execution continues after its ENDFOR statement. Since the control variable is compared to the end limit prior to statement execution, it is possible for the loop statements to be skipped entirely. For example, the following loop statements will not be executed, since the end limit is exceeded before the loop starts:

```
starting:=1; ending:=0
FOR x:=starting TO ending DO
  PRINT "*",
ENDFOR x
```

NOTE: CBM COMAL will automatically convert all NEXT statements into ENDFOR statements, which are more compatible with the other loop structure terminating keywords. The keyword NEXT is still allowed, in keeping with the COMAL KERNAL, but will appear in listings as ENDFOR. Version 2.00 will list the word NEXT instead of ENDFOR if you issue the following command:

```
POKE $ 24B,PEEK($ 24B) BITOR % 00000100
```

Version 2.00 treats the FOR loop control variable as a LOCAL variable, preventing possible variable name conflicts. The control variable may be an integer variable which is much faster.

STRUCTURE FORM

```
FOR <controlvar>:=<start> TO <end> [STEP <step>] DO
  <statements>
ENDFOR [<control variable>]
```

EXAMPLE

```
FOR temp:=1 TO max STEP 5 DO
  EXEC ask
  EXEC compare
ENDFOR temp
```

A simple one-line version of the FOR structure is also available. It must not use the ENDFOR.

STRUCTURE FORM

```
FOR <controlvar>:=<start> TO <end> [STEP <step>] DO <statement>
```

EXAMPLE

```
FOR x:=1 TO 40 DO PRINT "-",
```

LOOP STRUCTURE (version 2.00 only)

The LOOP structure presents a block of statements for repeated execution, similar to the REPEAT and WHILE loop; however, its terminating condition occurs in the middle of the loop rather than at the beginning or end. An EXIT or EXIT WHEN statement provides a condition for leaving the LOOP structure. If the condition evaluates to TRUE (a value not equal to 0), program execution continues with the statement following the ENDLOOP statement. If it evaluates to FALSE (a value of 0), execution continues with the next statement. A loop structure should only have one exit or it is not part of structured programming. If there are no statements prior to your EXIT statement, you should be using a WHILE loop. If there are no statements after your EXIT statement, you should be using a REPEAT loop. The loop structure is useful in combination with the ERROR HANDLER structure.

NOTE

The LOOP ... ENDLOOP structure is not supported by version 0.14.

STRUCTURE FORM

```
LOOP
  <statements>
  EXIT WHEN <condition>
  <statements>
ENDLOOP
```

EXAMPLE

```
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: temp$
  EXIT WHEN temp$="*END*"
  pointer:+1
  array$(pointer):=temp$
ENDLOOP
```

Procedure and Function Structure

It is easy to write "modular" programs in COMAL, using procedures and functions, COMAL's specialties. You can call a procedure or function at any time, anywhere in your program, simply by using an EXEC statement or function call. When a running program comes upon an EXEC statement, it executes the specified procedure before continuing on to the next statement. The procedure in COMAL is called by name and allows parameter passing as well as local or global variables. A procedure can, in turn, call another procedure, or can even call itself. Procedures can be nested (one procedure within another) in version 2.00. A function can be numeric, integer, or string. Include a # after the name for an integer function, and a $ after the name of a string function (string functions are not implemented in version 0.14). The value of the function is returned via the RETURN statement. All the variations of procedures discussed below apply equally to a function, except instead of using an EXEC statement, it is called with a function call and a value is returned.

The first line of the procedure structure is the PROC statement. Here you give the procedure its name, specify any parameters if used, and choose whether or not it will be CLOSED with local variables. The PROC statement can become rather complex, which is what provides flexibility and power.

The simplest PROC statement merely includes the keyword PROC and the procedure name. All variables are global, and no parameters are used. It looks like this:

PROC STATEMENT SYNTAX

```
PROC <procedure name>
```

EXAMPLE

```
PROC ask
```

CALLING STATEMENT SYNTAX

```
[EXEC] <procedure name>
```

EXAMPLE

```
EXEC ask
```

You can make all the variables used in the procedure LOCAL simply by including the keyword CLOSED after the ⟨procedure name⟩. A local variable is unknown to the rest of the program. And besides isolating the variables within the CLOSED procedure, you isolate the procedure itself. It does not know about any of the variables, functions, procedures, or labels used in other parts of the program. In order to share some of them, you must use parameters or IMPORT statements. The simple CLOSED PROC statement looks like this:

PROC STATEMENT SYNTAX

```
PROC <procedure name> CLOSED
```

EXAMPLE

```
PROC pause CLOSED
```

CALLING STATEMENT SYNTAX

```
[EXEC] <procedure name>
```

EXAMPLE

```
EXEC pause
```

To assign some variables an initial value, a CLOSED procedure can use parameters. These parameters are considered local even if the procedure is not closed. A procedure does not have to be CLOSED to be able to use parameters. Any procedure can specify parameters, which receive initial values from the calling statement. Multiple parameters may be used, each separated by a comma. With simple value parameter passing, the procedure simply receives the starting values for each of the variables in its parameter list. The calling statement must contain the same number of values as the procedure requires for its parameters. In this case the value passing is not "two ways." A PROC statement with simple value passing parameters looks like this:

PROC STATEMENT SYNTAX

```
PROC <procedure name>(<formal parameter list>)
  or
PROC <procedure name>(<formal parameter list>) CLOSED
```

CALLING STATEMENT SYNTAX

```
[EXEC] <procedure name>(<actual parameter list>)
```

PROC STATEMENT EXAMPLES

```
PROC error'num(n)
  or
PROC mail'label(n$,a$,c$,s$,zip) CLOSED
```

CALLING STATEMENT EXAMPLES

```
EXEC error'num(5)
  or
EXEC MAIL'LABEL(name$,address$,city$,state$,zip)
```

It is very nice to be able to CLOSE a procedure, allowing local variables. But many times there will be some variables that you wish to be in effect throughout the whole program, including in CLOSED procedures. COMAL will allow you to do that, via the IMPORT statement. The IMPORT statement specifies which variables, functions, and procedures will not be "locked out" of the CLOSED procedure. Entire arrays may be global by including the array name without the parentheses section in the IMPORT statement. In version 2.00 it is required that any procedure or function called from within a CLOSED procedure be declared in an IMPORT statement. Procedures and functions are always global in version 0.14.

IMPORT STATEMENT FORM

```
IMPORT <identifier list>          identifiers separated by commas
```

EXAMPLE

```
IMPORT table,players$
```

You may also wish specific parameters could be "two-way" parameters, receiving their initial value from the calling statement and returning a value back to the calling statement as well. In essence, this provides OUTPUT and UNIVERSAL

(INPUT and OUTPUT) parameters, in addition to the previously mentioned INPUT parameters. COMAL provides an easy way to accomplish this. Simply include the word REF (for reference) in front of the parameter variable in the procedure's parameter list. Each variable preceded by REF will be used as an alias, called by reference, for the matching variable in the calling statement's parameter list. Any change in value during procedure execution is reflected on both. However, the REF variable name is still considered local within the procedure, and the same variable name can be used elsewhere in the program without conflict.

A final touch of elegance is available with COMAL. A procedure can be recursive in nature. It can call itself. This is a very powerful feature, but may lead to strange results if used incorrectly.

COMAL allows you the choice of making the keyword EXEC optional. This provides a less cluttered listing which you may prefer. But, if you prefer to see the word EXEC in your calling statements, that option is still available as well. See keyword SETEXEC (one of the special setup keywords listed at the beginning of the keyword section) for more information on this option.

There is no restriction on where in the program you may call a procedure or function. The procedure and function declarations may be at the beginning or at the end of the program, or anywhere in between. But they are not allowed inside any control structure (IF, CASE, LOOP, FOR, REPEAT, WHILE, or TRAP structure).

Version 2.00 allows procedures and functions to be EXTERNAL to the program. The program thus can call procedures and functions that are on disk. All that is needed in the program is the PROC or FUNC header statement, with the proper name, parameters, and external filename specified. All you need to know about an external procedure or function is the number of parameters and their sequence and type. The COMAL system does the rest for you.

It is not complicated to store a procedure or function on disk in a form that can be used as external. Follow these easy steps:

(1) The procedure or function must be declared CLOSED. Do NOT include the keyword EXTERNAL in the header.
(2) The procedure or function may contain other procedure or function declarations (nested procedures and functions). However, it may not contain any IMPORT statements. See below for an explanation of how to use nested external procedures in the same manner as IMPORTing the procedures.
(3) There are no restrictions on parameter passing; however, IMPORT statements are not allowed.
(4) Remarks or blank lines may precede the procedure or function.

(5) Statements may follow the procedure or function. These statements will not be executed when the procedure or function is called externally. However, they are allowed to be included to facilitate future testing of the procedure.
(6) SAVE the procedure to disk. It is suggested to end the file name with a .E to avoid confusion with other types of files.
(7) The EXTERNAL procedure is now ready to be used.

For example, here is how to create an external function to simulate rolling dice. It will return a number that is the total of two random integers between 1 and 6:

```
// revision date: 4-26-83
FUNC dice CLOSED
  number1:=RND(1,6)
  number2:=RND(1,6)
  RETURN number1+number2
ENDFUNC dice
SAVE "0:dice.e"
```

Now, any future program can easily use this function by simply including only a function declaration header statement that specifies EXTERNAL "0:dice.e". The following program is an example:

```
// print the result of throwing 2 dice until a 7 or 11 results
REPEAT
  result:=throwdice
  PRINT result;
UNTIL result=7 OR result=11
//
FUNC throwdice EXTERNAL "0:dice.e"
//
END "DONE"

RUN
5 10 4 10 7
DONE
```

Notice that the actual function name of the external function on the disk is *dice*. However, COMAL allows it to be given an "alias" name in the calling program. In our example it was called *throwdice*. Also note that in practice, only large procedures and functions will be external, due to the time delay in the disk access, unless this access time is of no concern (this time delay is eliminated with the COMBI board, due to its 128K RAM buffer).

NOTE

If you have a procedure that you wish to save as external, but it IMPORTs other procedures, you may either convert the IMPORT statement into external PROC headers for those needing importing (dynamic nesting) or directly nest those procedures inside the original one. The procedure will then work as before. For example, to convert the procedure BOLD'STRING into an external procedure,

the IMPORT statement must be changed as shown below (assuming that the procedure BOLD'CHAR is already an external procedure on the disk):

ORIGINAL VERSION (bold'char procedure IMPORTed):

```
PROC bold'face(b$,c'return) CLOSED
  IMPORT bold'char
  FOR char:=1 TO LEN(b$) DO EXEC bold'char(b$(char))
  IF c'return THEN PRINT
ENDPROC bold'face
```

EXTERNAL VERSION (the bold'char procedure is external):

```
PROC bold'face(b$,c'return) CLOSED
  FOR char:=1 TO LEN(b$) DO EXEC bold'char(b$(char))
  IF c'return THEN PRINT
  //
  PROC bold'char(b$) EXTERNAL "1:bold'char.e"
  //
ENDPROC bold'face
```

EXTERNAL VERSION (the bold'char procedure is directly nested)

```
PROC bold'face(b$,c'return) CLOSED
  FOR char:=1 TO LEN(b$) DO EXEC bold'char(b$(char))
  IF c'return THEN PRINT
  //
  PROC bold'char(b$) CLOSED
    z:=ZONE
    ZONE Ø
    PRINT CHR$(27),"g"
    FOR x:=1 TO 3 DO PRINT b$,
    PRINT CHR$(27),"4",
    PRINT " ",
    ZONE z
  ENDPROC bold'char
  //
ENDPROC bold'face
```

Now you can enjoy the use of an external procedure and function library on disk, without any complicated usage rules, other than the restriction on IMPORT statements. If a specific procedure needs to be updated, it only has to be updated once, on the master PROC library disk. Then all programs using it will automatically be using the latest revision. Without this external procedure capability, you would have to manually update every program that used the procedure needed updating.

If you are using the COMBI board with 128K RAM, external procedures can be loaded into the extra RAM with an LOADEXT "name.e" statement. Thus, you may wish to have a "loader" that loads all your external procedures into the 128K RAM buffer when you power up the computer. If for some reason, you have over 128K of external procedures and functions, only 128K can be retained, and the extra ones will use disk access as on a normal ROM board, Commodore 64 Cartridge, or 8096. By using the 128K RAM in this way, programs using external routines will not be slowed down by disk access (other than the initial loading time).

Another benefit of using external procedures and functions is that your programs will be smaller in size, take up less disk space, and will be compatible with each other. Also, since the external procedures and functions are only in the memory while they are executing, several procedures can use the same memory space, each one discarded as soon as it is finished executing. This is sometimes referred to as virtual memory.

COMAL provides a special feature when dealing with external functions and procedures. If a program is stopped while executing an external segment, the system will allow you to LIST, EDIT, FIND, DEL, SAVE, MERGE, SIZE, CON, AUTO, and RENUM using only that specific segment. This can be convenient. Then to get back to the main program (in case you forgot to SAVE it), simply issue the command MAIN. Once you return to the main program, the external section is inaccessable.

One drawback of using external procedures, is that the COMAL system must retrieve the procedure from disk every time it is called (except on the COMBI board). This is because once an external procedure is finished executing, it is discarded from memory. This may not be a problem in most cases, but you should keep it in mind.

The whole procedure, with all its options, looks like this:

PROCEDURE STRUCTURE FORM

```
PROC <procedure name>[(<parm list>)] [CLOSED] [EXTERNAL <filename>]
  {IMPORT <identifier list>}              not version 0.14
  <statements>
ENDPROC <procedure name>
```

NOTE: CLOSED and EXTERNAL may not both be used at the same time.

CALLING STATEMENT FORM

```
[EXEC] <procedure name>[(<actual parameter values>)]
```

PROCEDURE EXAMPLE

```
PROC report'card(errors) CLOSED
  IMPORT player'name$
  PRINT "Well,";player'name;"- the game is over."
  IF errors>3 THEN
    PRINT errors;"errors is quite a few!"
  ELSE
    PRINT "Good Job! Your errors amounted to:";errors
  ENDIF
ENDPROC report'card
```

CALLING STATEMENT EXAMPLE

```
EXEC report'card(5)        or      report'card(5)
```

FUNCTION STRUCTURE FORM

```
FUNC <function name>[(<parm list>)] [CLOSED] [EXTERNAL <filename>]
  {IMPORT <identifier list>}                    not version Ø.14
  <statements>
  RETURN <value>
ENDFUNC <function name>
```

NOTE: CLOSED and EXTERNAL may not both be used at the same time.

FUNCTION CALL FORM

```
<function name>[(<actual parameter values>)]      do not use EXEC
```

FUNCTION EXAMPLE

```
FUNC odd(integer)
  RETURN (integer MOD 2)
ENDFUNC odd
```

FUNCTION CALL EXAMPLE

```
type:=odd(age)
```

ERROR HANDLER STRUCTURE (version 2.00 only)

COMAL includes an advanced structured error handling system. The basic structure is similar to the IF ... ELSE .... ENDIF structure. The COMAL error handling structure is appropriately called TRAP ... HANDLER ... ENDTRAP. It looks like this:

STRUCTURE FORM

```
TRAP
  +---------------+
  !  statements   !
  ! to be trapped !
  +---------------+
HANDLER
  +---------------+
  !  statements   !
  ! executed only !
  ! in case of an !
  !    error      !
  +---------------+
ENDTRAP
```

EXAMPLE 1

```
DIM filename$ of 20
INPUT "Name of file:": filename$
TRAP
  OPEN FILE 28,filename$,READ
  CLOSE FILE 28
  PRINT "The file exists."
HANDLER
  CLOSE FILE 28
  PRINT "The file does not exist."
ENDTRAP
PRINT "All done."
```

The statements that should be trapped in case of an error are placed in the block between the TRAP and HANDLER keywords. These are the "trapped" statements. They are executed in the normal manner. If they all execute without error, program execution continues after the ENDTRAP statement, skipping the

statements in the HANDLER section. If an error occurs while executing the "trapped" statements, execution immediately is transfered to the first statement in the HANDLER section of statements (the remainder of the "trapped" statements are skipped). The two possible outcomes are shown by the sample run of the example program listed above:

```
CAT
 Ø "EXAMPLE DISK     " ID 2A
 5     "BITPAC.OBJ"        SEQ
 3     "BEFORE"            PRG
 5     "AFTER"             PRG
 647 BLOCKS FREE.

RUN          (supply a file name that exists)
Name of file:BEFORE
The file exists.
All done.
RUN          (supply a file name that doesn't exist)
Name of file:NOTHING
The file does not exist.
All done.
```

This illustrates how the HANDLER structure works. COMAL allows nested HANDLERS, just as other structures can be nested. The two statement blocks will be automatically indented to emphasize the structure. The HANDLER also includes three system variables: ERR, ERRFILE, and ERRTEXT$, and the statement REPORT.

When an error occurs within a "trapped" section, three system variables are assigned the appropriate values:

```
ERR is the error number
ERRFILE is the file number in use (if any)
ERRTEXT$ is the error message string
```

The statements in the HANDLER section may refer to these three variables in order to determine the appropriate action to take. It is important to note the difference between ERRTEXT$ and STATUS$. STATUS$ can be read only once, and it is then cleared. ERRTEXT$ remains unchanged as long as you are within the current HANDLER, no matter how many times you read it. The following examples illustrate the use of each:

EXAMPLE 2 (illustrates ERRTEXT$)

```
EXAMPLE 2  (illustrates ERRTEXT$)

DIM message$ of 80
message$:="You are now beginning the test of ERRTEXT$"
LOOP
  TRAP
    PAGE // clear screen
    PRINT AT 10,1: message$
    INPUT AT 12,1: "Enter a number (0 to stop): ": number
    EXIT WHEN number=0
    message$:="Your last number was "+str$(number)
  HANDLER
    message$:=ERRTEXT$
  ENDTRAP
ENDLOOP
END "All done."

RUN
  (the screen clears)
You are now beginning the test of ERRTEXT$

Enter a number (0 to stop): 589
  (the screen clears)
Your last number was 589

Enter a number (0 to stop): 21.8
  (the screen clears)
Your last number was 21.8

Enter a number (0 to stop): ABC
  (the screen clears)
input error

Enter a number (0 to stop): 0

All done.
```

EXAMPLE 3 (illustrates ERR)

```
PAGE // clear screen
PRINT AT 10,1: "You are now beginning the test of ERR"
LOOP
  TRAP
    INPUT AT 12,1: "Enter a number (0 to stop): ": number
    EXIT WHEN number=0
    PAGE // clear screen
    PRINT AT 10,1: "Your last number was";number
  HANDLER
    PAGE
    CASE ERR OF
    WHEN 206
      PRINT AT 10,1: "A number is expected."
    WHEN 2
      PRINT AT 10,1: "The number is too large."
    OTHERWISE
      PRINT AT 10,1: "Another type of error."
    ENDCASE
  ENDTRAP
ENDLOOP
END "All done."

RUN
  (screen clears)
You are now beginning the test of ERR

Enter a number (0 to stop): 75.3
  (screen clears)
Your last number was 75.3

Enter a number (0 to stop): 99e9999
  (screen clears)
The number is too large.

Enter a number (0 to stop): 0

All done.
```

EXAMPLE 4 (illustrates ERRFILE and ERRTEXT$)

```
DIM in'name$ OF 20, out'name$ of 20
in'file:=2; out'file:=3
PAGE
PRINT AT 10,1: "Copy a sequential file"
LOOP
  INPUT AT 12,1: "Source File Name: ": in'name$
  INPUT AT 14,1: "Target File Name: ": out'name$
```

```
    PRINT AT 10,1: SPC$(79)
    TRAP
      OPEN FILE in'file,in'name$,READ
      OPEN FILE out'file,out'name$,WRITE
      EXIT // both files opened without error
    HANDLER // error in open statements
      CLOSE
      CASE ERRFILE OF
      WHEN in'file
        PRINT AT 10,1: in'name$;":";error$
      WHEN out'file
        PRINT AT 10,1: out'name$;":";error$
      OTHERWISE
        PRINT AT 10,1: error$
      ENDCASE
    ENDTRAP
  ENDLOOP
  TRAP
    WHILE NOT EOF(in'file) DO
      PRINT FILE out'file: GET$(in'file,5000), // copy 5K at a time
    ENDWHILE
  HANDLER
    IF ERRFILE=in'file THEN
      PRINT AT 16,1: "Error in input file: "+in'name$
    ELIF ERRFILE=out'file THEN
      PRINT AT 16,1: "Error in output file: "+out'name$
    ELSE
      PRINT AT 16,1: "Non I/O error occurred"
    ENDIF
  ENDTRAP
  CLOSE
  PAGE // clear screen
  END "Copy program finished"

  FUNC error$ CLOSED // error message without the numbers
    DIM e$ of 40
    e$:=ERRTEXT$(4:LEN(ERRTEXT$))
    RETURN e$(1:("," IN e$)-1)
  ENDFUNC error$

  RUN
  Copy a sequential file

  Source File Name: temp.1

  Target File Name: final.1
    (if an error occurs, the error message replaces the top line)
    (screen clears when finished)
  Copy program finished
```

REPORT — GIVES FLEXIBILITY TO THE ERROR HANDLER

The REPORT statement is an optional part of the ERROR HANDLER structure. It can be used to generate a "user error number" to be passed to an outer HANDLER structure. REPORT ⟨number⟩ will generate an error with the number specified (this number may also be a predefined system error number). If a number is not included, COMAL will generate the previous error once again. In either case, the REPORT statement does the following:

(1) If not within an ERROR HANDLER the error is reported to the system.
(2) If within a trapped section (top half of an ERROR HANDLER structure) the REPORT statement causes a jump into the HANDLER section of that structure.
(3) If within a HANDLER section (bottom half of an ERROR HANDLER structure) the REPORT statement causes and "exit" from that ERROR HANDLER structure into an outer HANDLER structure. If no outer HANDLER exists, then it reports to the system.

The REPORT statement is useful when you would like to trap only certain errors and ignore the others. You can check if an error is of the type you wish to catch, and if not, issue the statement REPORT. For example, error number 206 is generated when a number is expected from an INPUT statement, and some nonnumeric text is entered. The following HANDLER will catch only this error and will assign the value of 0 to the number input. All other errors (such as error#2, overflow) are REPORTed to the system:

EXAMPLE 5

```
PAGE // clear screen
TRAP
  INPUT AT 10,1: "Number please: ": number
HANDLER
  PRINT
  IF ERR=206 THEN
    number:=0
  ELSE
    REPORT
  ENDIF
ENDTRAP
PRINT AT 12,1: "Your number was";number
```

REPORT also allows you to generate your own error numbers to be reported to an outer HANDLER. Thus you can create your own error number classification system.

The following example illustrates LOCAL (dynamically nested) HANDLER structures, as well the use of a user-defined error number.

EXAMPLE 6

```
PAGE // clear screen
user'error:=300 // this is the number for our generated error

FOR temp:=-2 TO 50 STEP 15 DO
  TRAP
    PRINT USING "-##! = ": temp,
    PRINT fac(temp)
  HANDLER
    IF ERR=user'error THEN
      PRINT "undefined"
    ELSE
      REPORT
    ENDIF
  ENDTRAP
ENDFOR temp

FUNC fac(number) CLOSED
  IMPORT user'error
  IF number<0 THEN
    REPORT user'error
  ELSE
    TRAP
      result:=1
      FOR temp:=2 TO number DO result:=result*temp
      RETURN result
    HANDLER // number is too big resulting in overflow
      RETURN 1e+38 // a large number
    ENDTRAP
  ENDIF
ENDFUNC fac

RUN
-2! = undefined
13! = 6.2270208e+09
28! = 3.04888345e+29
43! = 1e+38
```

APPENDIX B
STRING HANDLING

String handling in COMAL is a simple matter. It is, however, different than the method used by Commodore BASIC (however, COMAL version 2.00 can simulate the BASIC method of string handling using three user defined string functions. COMAL allows strings as well as string arrays and string functions.

A string must be dimensioned before it is used. Once it is dimensioned, it cannot be dimensioned again. To dimension a string you must specify the string variable name along with the maximum number of characters to allow for:

```
DIM a$ OF 10
```

Now A$ may be used in the program and can hold up to 10 characters. If you assign more than 10 characters to A$, it will ignore all characters after the first 10 (i.e., "abcdefghijkl" assigned to A$ would become "abcdefghij"). This feature is not available in Commodore BASIC, and you may wish to take advantage of it. If you are asking for a reply, and are only concerned with a one character answer, simply use a string variable dimensioned for a maximum of one character:

```
DIM c$ OF 1
```

Now if "YES", "YEP", "YOU BET", or "YIPPEE" are assigned to C$, C$ would become equal to "Y". You may assign a value to a string variable with an INPUT, READ, or LET statement.

You can add something on to the end of a string variable (called concatenating). Assume that A$ equals "ABC". To add an "X" to the end of A$ you would write:

```
a$ :=a$ + "X"
```

Version 2.00 accepts the above method, and offers an additional way:

```
a$ :+ "X"
```

Either way, A$ becomes "ABCX". It is easy to choose any substring you wish from an existing string. Use the following guide:

```
<string variable>$(<start character>[:<end character>])

  <string variable> is the name of the string variable
  <start character> is the first character of the substring
  <end character> is the last character of the substring
    if omitted, only the start character is used
```

Now, assume A$ has a value of "ABCDEFG". To specify the "DEF" in A$ use the following:

```
a$ (4:6)                    start with character 4 and end with 6
```

A substring can be used just like a string. You can assign it to another string variable or you can even change it to something else, without affecting the rest of the string (you can't do that in CBM BASIC). For example, assume A$ is assigned the value of "ABCDEFG". Now let's change the "D" to an "X":

```
a$(4):="X"
   or                       A$ then equals "ABCXEFG"
a$(4:4):="X"
```

If the actual string value being assigned to a substring is shorter in length than the substring length, spaces are added to the right of the actual string value to make it the same length as the substring length. Thus, using the same A$ from above, we could insert three spaces beginning at the third position like this:

```
a$(3:5):=""
   or                       A$ then equals "AB   FG"
a$(3:5):=" "
```

You may do the same things with elements of a string array. It must be dimensioned before it is used. For a description of a string array, see keyword DIM. To specify an element in the string array you would use the array name followed by the array index value(s). For example, we can have the names of three players stored in array PLAYER$. It would be dimensioned like:

```
DIM player$ (3) OF 20
```

The name of player 2 would be indicated like this:

```
player$ (2)
```

Now, to access a substring of an element in a string array, use this guideline:

```
<array name>$(<array index part>) (<substring part>)

  <array name> is the name of the string array
  <array index part> specifies which element to use:
    <index number>{,<next index number>}
  <substring part> is just like for a string represented by:
    <start character>[:<end character>]
      <start character> is the first character of the substring
      <end character> is the last character of the substring
        if omitted, only the start character is used
```

So, if we have a two-dimension string array named B$, and we wish to change the first three characters in the first element of the second dimension to "XYZ", we would use the following:

```
b$ (2,1) (1:3):="XYZ"
```

The above is interpreted like this: B$ is the the array name. It has two dimensions, so the 2,1 are used as the index to dimension 2, element 1. The substring is specified by the 1:3, meaning start with the first character and end with the third character. Thus if previously B$(2,1) had the value of "ABCDEFG", it would now be equal to "XYZDEFG".

Of course, the current value of a substring of an element in a string array can be assigned to another variable, or used just like a string. For example, to use the first character of previously mentioned B$(2,1) as a CASE control expression use the following:

```
CASE b$(2,1)(1:1) OF
  or                          evaluates to CASE "X" OF
CASE b$(2,1)(1) OF
```

If the length of the actual string value being assigned to the substring of an array element is less than the length of the substring, spaces will be added to the right of it. For example, to change the first ten characters of element 5 in array D$, we would use this: D$(5)(1:10):="".

String functions may also be used, including substrings of a string function, using the following guideline:

```
<string function name>[(<actual parameters>)][(<substring part>)]

  <string function name> is <identifier>$
  <actual parameters> is optional, represented by:
    [REF ]<value>{,[REF ]<value>}
  <substring part> is just like for a string, represented by:
    <start character>[:<end character>]
      <start character> is the first character of the substring
      <end character> is the last character of the substring;
        if omitted, only the start character is used
```

SPECIAL NOTE FOR VERSION 2.00 — ENHANCED STRING OPERATIONS

Substrings can now be taken of any string operand, thus anywhere in this appendix that "string variable" is used, you may use a string function, string constant, or string expression. For example:

```
<variable>(<substring part>)
<constant>(<substring part>)
<system function>(<substring part>)
<user function>(<substring part>)
(<string expression>)(<substring part>)

"Commodore"(start'pos:end'pos)    substring of constant
STR$(1000+number)(2:4)            substring of system function
                                  forces leading zeros
(first$+"x"+last$)(1:marker)      substring of expession
```

A recursive string function example:

```
FUNC hex$(n) CLOSED
  IF n<16 THEN
    RETURN "0123456789abcdef"(n+1)
  ELSE
    RETURN hex$(n DIV 16)+hex$(n MOD 16)
  ENDIF
ENDFUNC hex$
```

A substring of a substring would look like:

```
((a$(1:n))(f:20))(2:4)
```

Commodore BASIC String Handling to COMAL Conversion

You may wish to convert a program written in Commodore BASIC to run in COMAL. BASIC uses three keywords to specify substrings: LEFT$, RIGHT$, and MID$.

```
  Commodore BASIC:
LEFT$(<string expression>,<number>)
  <string expression> is a:
    string constant
    string variable or
    string array element
  <number> is the number of consecutive characters to use
    beginning with character 1

   Commodore BASIC:
RIGHT$(<string expression>,<number>)
  <string expression> is a:
    string constant
    string variable or
    string array element
  <number> is the number of consecutive characters to use
    ending with the last character
```

```
   Commodore BASIC:
MID$(<string expression>,<start>[,<number>])
  <string expression> is a:
    string constant
    string variable or
    string array element
  <start> is the character to start with
  <number> is the number of consecutive characters to use
    if omitted, the remainder of the string is used
```

It is easy to convert BASIC string handling to work with COMAL version 2.00. Simply include three user defined string functions. With these three functions you can virtually duplicate BASIC string handling. However, it is recommended that you use these three string functions only for this purpose. When writing a COMAL program, use the COMAL methods of string handling, which are far superior.

APPENDIX C
SEQUENTIAL FILE DIFFERENCES

CBM COMAL uses two different methods of storing records in a sequential file. One method uses a predefined delimiter between records. Another is to precede each record with a count of how many characters are in that record.

WRITE FILE and READ FILE

These files are referred to as BINARY FILES. A string record created by a WRITE FILE statement is preceded by a two byte character count. The record may be any length, and since there is no delimiter involved, may include any of the ASCII character set. The two byte character count is represented in this manner: multiply the first byte times 256 and add the second byte. This can be written as: (⟨byte 1⟩*256) +⟨byte 2⟩. Numeric real data is always written as a five byte binary coded record, and integer data as a two byte binary coded record, no matter what the numeric value is. A COMAL READ FILE statement is used to read a record created by a WRITE FILE statement.

PRINT FILE and INPUT FILE

These files are referred to as ASCII FILES. A record created by a PRINT FILE statement is followed by a delimiter. A COMAL INPUT FILE statement is used to read a record created by a PRINT FILE statment. The delimiter used by COMAL is CHR$(13)+CHR$(10) (a carriage return, linefeed) in version 0.14 and just CHR$(13) in version 2.00. Both string records and numeric records use this method. (A COMAL INPUT FILE statement will also read a Commodore BASIC file, which uses a CHR$(13) as its delimiter). Numeric data is written to the file just as it is written to a printer. This is significant if you wish to read a numeric record written by Commodore BASIC. COMAL represents a numeric value just as it is, thus a 5 is represented as 5. However, Commodore BASIC precedes each number with one byte for the sign (- for negative, ⟨space⟩ for nonnegative) and ends each number with a cursor right. Therefore, a 5 is represented as ⟨space⟩5⟨cursor right⟩. Thus for COMAL to read a Commodore BASIC numeric file, a short conversion routine would have to be used. However, COMAL can read a Commodore BASIC text file directly with INPUT FILE statements.

GET$

COMAL GET$ function calls can read any sequential file, any number of bytes at a time. Care must be taken, however, since it cannot distinguish a delimiter or record count byte from an ASCII character in a record, nor can it decode the five byte numeric record created by WRITE FILE. GET$ can be useful if carefully planned. For instance, you can read the first character of a record created by a PRINT FILE statement, analyze or compare it, and then use an INPUT FILE statement to read the remainder of the record. This cannot be done with a string record created by a WRITE FILE statement, since it depends on a character count preceding each record. Your program could, however, read the character count, and then read that many characters as the next record, one character at a time, using GET$ statements. GET$ may also be used to read the directory of a disk one byte at a time.

APPENDIX D
SOME USEFUL SAMPLE PROCEDURES AND FUNCTIONS

TO USE THESE PROCEDURES AND FUNCTIONS

COMAL makes it very easy to program in modules. You can store your procedures and functions on tape or disk. Later, when you need one of them, you simply ENTER or MERGE it, automatically merging it with your current program. Having a library of commonly used procedures and functions will save you lots of time when writing new programs, plus they will help make the programs compatible with each other. To give you a start with your procedure library, this appendix lists and explains many useful procedures. Use the ones that you find useful. Modify them to suit your particular needs. Once you have it typed in, remember to LIST it to tape or disk. Put a .L at the end of the file name to remind you later to use ENTER or MERGE (not LOAD) to retrieve it.

To avoid line number problems in version 0.14 while merging a procedure into your program, always begin your procedures with line 9000. Then as you merge them into the program, first RENUMber the program, bringing the line numbers down below 9000. This is unnecessary in version 2.00 since the default MERGE command will automatically renumber the procedure at the end of the current program.

To avoid variable conflicts between your procedure and the rest of the program, you may wish to use special naming conventions (like starting any procedure variable with a Z or such), or better yet, simply declare all your procedures CLOSED, as most of the sample procedures in this appendix are. Variables in a CLOSED procedure will not conflict with variables by the same name in the main program.

Now, here is an example of how quickly you can write a complete program using procedures from your library. This example will use procedures TAKE'IN, FETCH, GET'VALID, and GET'CHAR, which are assumed to be on the disk in drive 0 of unit 8 (a standard CBM disk drive).

First, clear out any program now in the computer:

```
NEW
```

Now, enter the main program section. Use AUTO to provide line numbers for you.

```
AUTO

0010 // CREATE A NEw EMPLOYEE / CLASSIFICATION FILE
0020 DIM NAME$ OF 20, CLASS$ OF 20
0030 OPEN 2,"0:EMPLOYEE",WRITE
0040 REPEAT
0050 EXEC TAKE'IN("EMPLOYEE NAME:","A",NAME$,20)
0060 IF NAME$>"" THEN
0070 REPEAT
0080 EXEC TAKE'IN("CLASSIFICATION:","B",CLASS$,20)
0090 UNTIL CLASS$>""
0100 WRITE FILE 2: NAME$,CLASS$
0110 ENDIF
0120 UNTIL NAME$=""
0130 CLOSE
0140 // PROCEDURES FOLLOW
0150          version 2.00 hit STOP key :: version 0.14 hit RETURN
```

To see it with the structures indented, do a LIST:

```
LIST

0010 // CREATE A NEW EMPLOYEE / CLASSIFICATION FILE
0020 DIM name$ OF 20, class$ OF 20
0030 OPEN FILE 2,"0:EMPLOYEE",WRITE
0040 REPEAT
0050   EXEC TAKE'IN("EMPLOYEE NAME:","A",name$,20)
0060   IF name$>"" THEN
0070     REPEAT
0080       EXEC take'in("CLASSIFICATION:","B",class$,20)
0090     UNTIL class$>""
0100     WRITE FILE 2: name$, class$
0110   ENDIF
0120 UNTIL name$=""
0130 CLOSE
0140 // PROCEDURES FOLLOW
```

Now, let's merge in the procedures from our procedure library.

```
VERSION Ø.14                    VERSION 2.ØØ
-----------------------------   ----------------------------------
ENTER "TAKE'IN.L"               MERGE "TAKE'IN.L"
RENUM                           MERGE "FETCH.L"
ENTER "FETCH.L"                 MERGE "GET'VALID.L"
RENUM                           MERGE "GET'CHAR.L"
ENTER "GET'VALID.L"
RENUM
ENTER "GET'CHAR.L"
RENUM
```

Now, list the complete program:

```
0010 // create a new employee / classification file
0020 DIM name$ OF 20, class$ OF 20
0030 OPEN FILE 2,"0:employee",WRITE
0040 REPEAT
0050   take'in("employee name:","a",name$,20)
0060   IF name$>"" THEN
0070     REPEAT
0080       take'in("classification:","b",class$,20)
0090     UNTIL class$>""
0100     WRITE FILE 2: name$,class$
0110   ENDIF
0120 UNTIL name$=""
0130 CLOSE
0140 //procedures follow
0150 //
0160 PROC take'in(prompt$,valid$,REF reply$,max) CLOSED
0170   IMPORT fetch,get'valid,get'char // not used in version 0.14
0180   z:=ZONE
0190   ZONE 0
0200   PRINT prompt$,
0210   PRINT "<", //                        left side
0220   FOR x:=1 TO max DO PRINT " ", //     blank out input area
0230   PRINT ">", //                        right side
0240   FOR x:=1 TO max+1 DO PRINT "<LEFT>", // cursor left
0250   fetch(reply$,valid$,max)
0260   PRINT //                           carriage return
0270   ZONE z
0280 ENDPROC take'in
0290 //
```

```
0300 PROC fetch(REF a$,v$,max) CLOSED
0310   IMPORT get'valid,get'char // not needed in version 0.14
0320   DIM valid$ OF 40, b$ OF 1
0330   // if v$ = "a" then the alphabet is used (plus space)
0340   // if v$ = "d" then the digits are used
0350   // if v$ = "b" then both alphabet and digits are used (plus space)
0360   // otherwise valid$ is set to the value as sent
0370   // note: >>> carriage return and delete key are added to valid$
0380   z:=ZONE
0390   ZONE 0
0400   a$:=""
0410   CASE v$ OF
0420   WHEN "a"
0430     valid$:="abcdefghijklmnopqrstuvwxyz "
0440   WHEN "d"
0450     valid$:="0123456789"
0460   WHEN "b"
0470     valid$:="abcdefghijklmnopqrstuvwxyz 0123456789"
0480   OTHERWISE
0490     valid$:=v$
0500   ENDCASE
0510   done:=FALSE; num:=0
0520   REPEAT
0530     get'valid(b$,valid$+CHR$(13)+CHR$(20))
0540     CASE b$ OF
0550     WHEN CHR$(13) // carriage return
0560       done:=TRUE
0570     WHEN CHR$(20) // delete key
0580       IF num THEN // only do if already have something
0590         num:-1 //    minus one for number in string
0600         PRINT "<LEFT> <LEFT>", // cursor left space cursor left
0610         a$:=a$(1:num)
0620       ENDIF
0630     OTHERWISE //    all other valid characters
0640       IF num<max THEN // don't go past maximum
0650         a$:=a$+b$ //  add character to the stringreturn needed
0660         num:+1 //     add 1 to the count of characters
0670         PRINT b$, //  print the character
0680       ENDIF
0690     ENDCASE
0700   UNTIL done
0710   ZONE z
0720 ENDPROC fetch
0730 //
```

```
0740 PROC get'valid(REF c$,valid$) CLOSED
0750   IMPORT get'char // not used in version 0.14
0760   REPEAT
0770     get'char(c$)
0780   UNTIL c$ IN valid$
0790 ENDPROC get'valid
0800 //
0810 PROC get'char(REF c$) CLOSED // version 2.00 & c64 0.14
0820   REPEAT
0830     c$:=KEY$
0840   UNTIL c$="" //                    clear buffer
0850   REPEAT
0860     c$:=KEY$
0870   UNTIL c$>"" //                    wait for key
0880 ENDPROC get'char
```

Remember to SAVE the program BEFORE trying it, in case something goes wrong (power failure, etc.).

```
SAVE "EMPLOYEE'ENTRY"
```

Within a couple of minutes, we have a complete program that will create a file of employees and their classification using keyboard input. It won't accept complete garbage like a number as part of a name, or symbols such as #$*. It shows the user how much space is allowed for the name and classification and allows mistakes to be deleted as you go. Now here is a typical run of this program:

```
RUN
  (file 2 named EMPLOYEE is opened for output)
EMPLOYEE NAME:<SAM                    >
CLASSIFICATION:<CLERK 2                >
  (the records SAM and CLERK 2 are written to file 2)
EMPLOYEE NAME:<SUZIE                  >
CLASSIFICATION:<RECEPTIONIST           >
  (the records SUZIE and RECEPTIONIST are written to file 2)
EMPLOYEE NAME:<                       >    just hit RETURN here
  (file 2 is closed)
```

FUNCTION NAME: BUT'FIRST$
NOTE: Version 2.00 only

This function will return all but the first character of the string passed to it. If the string is less than two characters long, the null string ("") is returned.

SYNTAX FOR CALLING STATEMENT

```
but'first$(<intext>)

   <intext> is a <string expression>
```

EXAMPLES OF CALLING STATEMENT

```
PRINT but'first$(temp$)
text$:=but'first$(text$)
```

FUNCTION LISTING

```
//
FUNC but'first$(text$) CLOSED
  IF LEN(text$)>1 THEN
    RETURN text$(2:LEN(tex+
  ELSE
    RETURN ""
  ENDIF
ENDFUNC but'first$
```

FUNCTION NAME: BUT'LAST$
NOTE: Version 2.00 only

This function will return all but the last character of the string passed to it. If the string is less than two characters long, the null string ("") is returned.

SYNTAX FOR CALLING STATEMENT

```
but'last$(<intext>)

   <intext> is a <string expression>
```

EXAMPLES OF CALLING STATEMENT

```
PRINT but'last$(temp$)
text$:=but'last$(text$)
```

FUNCTION LISTING

```
//
FUNC but'last$(text$) CLOSED
  IF LEN(text$)>1 THEN
    RETURN text$(1:LEN(text$)-1)
  ELSE
    RETURN ""
  ENDIF
ENDFUNC but'last$
```

PROCEDURE NAME: CREATE
NOTE: Version 0.14 only. Version 2.00 use the CREATE statement.

This procedure creates a new random file, just like the version 2.00 CREATE statement.

SYNTAX FOR CALLING STATEMENT

```
EXEC create(<filename>,<number records>,<record length>)

  <filename> is an INPUT PARAMETER
  <number of records> is a <numeric expression>, INPUT PARAMETER
    the limit of the number of records depends upon the disk unit
  <record length> is a <numeric expression>, INPUT PARAMETER
    its value is from 1-254
```

EXAMPLES OF CALLING STATEMENT

```
EXEC create("TEST.DAT",40,10)
EXEC create(name$,total,rec'len)
```

PROCEDURE LISTING

```
// courtesy of UniComal Denmark
PROC create(name$,last'rec,rec'len) CLOSED
  z:=ZONE                  remember the old zone
  ZONE Ø                   no extra spaces after a comma
  OPEN FILE 79,name$,RANDOM rec'len
  PRINT FILE 79,last'rec: CHR$(255),
  CLOSE FILE 79
  ZONE z                   return the zone to the original
ENDPROC create
```

PROCEDURE NAME: CURSOR
NOTE: Version 0.14 only. Version 2.00 use CURSOR statement.

This procedure will position the cursor at the specified ⟨line⟩ and ⟨column⟩. The POKE statement is included to make sure that "quote mode" is off (quote mode is entered after an odd number of quote marks, prior to a carriage return). While quote mode is on, cursor movements are not executed. Their "symbol" is printed instead.

NOTES

(1) The line is specified first and the column (position on the line) second.
(2) The screen lines are referred to as lines 1-25, with the top line as line 1.
(3) The columns (positions) on each line are referred to as columns 1-40 on 40 column screens, and as columns 1-80 on 80 column screens, with the first position as column 1.

SYNTAX FOR CALLING STATEMENT

```
EXEC cursor(<line>,<column>)

  <line> is a <numeric expression>, INPUT PARAMETER
    the new cursor line
  <column> is a <numeric expression>, INPUT PARAMETER
    the new cursor position on the line
```

EXAMPLES OF CALLING STATEMENT

```
EXEC cursor(20,2)          puts the cursor on position 2 of line 20
EXEC cursor(row,col)
```

PROCEDURE LISTING

```
//
PROC cursor(line,col) CLOSED
  z:=ZONE                          remember the current ZONE setting
  ZONE Ø                           set ZONE to tab in every column
  quotemode:=2Ø5                   Commodore 64 use 212
  POKE quotemode,Ø                 verify quote mode is off
  PRINT "[HOME]",                            HOME cursor
  FOR l:=1 TO line-1 DO PRINT "[DOWN]",   cursor DOWN
  FOR c:=1 TO col-1 DO PRINT "[RIGHT]",   cursor RIGHT
  ZONE z                           reset ZONE to original setting
ENDPROC cursor
```

FUNCTION NAME: DISK'GET
REQUIRES PROCEDURE: DISK'GET'INIT
NOTE: This function is needed only with version 0.14. Version 2.00 use GET$instead. (Commodore 64 modifications by David Stidolph).

This function reads one character at a time from a previously opened disk file. It relies on a short machine language routine written by Steve Kortendick. This routine is loaded by procedure DISK'GET'INIT. Since the access to the disk file is by our own machine language program, COMAL does not know when to update the system variable EOF at the end of the file. Therefore, we use the variable FILE'END for the same purpose, assigning it a value of 1 (TRUE) when the file end is reached.

SYNTAX FOR THE FUNCTION CALL

```
disk'get(<file number>,<file end>)

  <file number> is a <numeric expression>, INPUT PARAMETER
    the file number of the file to get characters from
  <file end> is a <numeric variable>, OUTPUT PARAMETER
    set to TRUE (a value of 1) when end of file is reached
    set to FALSE (a value of Ø) while not at the end of file
```

EXAMPLES OF THE FUNCTION CALL

```
b$=CHR$(disk'get(infile,file'end))
dc=disk'get(1,eofl)
```

FUNCTION LISTING (PET/CBM Version)

```
FUNC disk'get(file'num,REF file'end) CLOSED
  POKE 636,file'num
  SYS 635                          execute machine language routine
  file'end=PEEK(15Ø)               set to TRUE at end of file
  RETURN PEEK(634)                 ASCII value of the character
ENDFUNC disk'get
```

FUNCTION LISTING (Commodore 64 Version)

```
FUNC disk'get(file'num,REF file'end) CLOSED
  POKE 2026,file'num
  SYS 2025                    execute machine language routine
  file'end=PEEK(144)          set to TRUE at end of file
  RETURN PEEK(2024)           ASCII value of the character
ENDFUNC disk'get
```

PROCEDURE NAME: DISK'GET'INIT

NOTE: Not used with version 2.00 which uses GET$ instead. (Commodore 64 modifications by David Stidolph.)

This procedure puts a machine language routine into the beginning of the cassette buffer. This routine, written by Steve Kortendick, will get one character at a time from a disk file. Since the access to the disk is through our own machine language program, COMAL does not update the system variable EOF at the end of the file. Therefore, we use the variable FILE'END for the same purpose, assigning it a value of 1 (TRUE) when the file end is reached. The DISK'GET'INIT procedure needs to be called only once, before executing any DISK'GET functions, since the machine language program remains in the cassette buffer while the program runs.

SYNTAX FOR CALLING STATEMENT

```
EXEC disk'get'init
```

EXAMPLE OF CALLING STATEMENT

```
EXEC disk'get'init
```

PROCEDURE LISTING (PET / CBM version)

```
PROC disk'get'init CLOSED
  FOR location=634 TO 649          cassette buffer locations
    READ value
    POKE location,value            put the code into the buffer
  ENDFOR location
  DATA 0,162,0,32,198,255,32,207
  DATA 255,141,122,2,32,204,255,96
ENDPROC disk'get'init
```

PROCEDURE LISTING (Commodore 64 version)

```
PROC disk'get'init CLOSED
  FOR location=2024 TO 2039          after screen memory buffer
    READ value
    POKE location,value              put the code into the buffer
  ENDFOR location
  DATA 0,162,0,32,198,255,32,207
  DATA 255,141,232,7,32,204,255,96
ENDPROC disk'get'init
```

PROCEDURE NAME: DISK'GET'SKIP
REQUIRES PROCEDURE: DISK'GET'INIT
REQUIRES FUNCTION: DISK'GET
NOTE: Needed only with version 0.14. Version 2.00 use GET$ instead.

This procedure uses procedure DISK'GET as a function to skip the specified number of characters in a previously opened file. Since access to the disk is through our own machine language program, COMAL does not update the system variable EOF at the end of the file. Therefore, we use the variable FILE'END for the same purpose, assigning it a value of 1 (TRUE) when the file end is reached. This procedure is useful when you are directly reading a disk's directory.

SYNTAX FOR CALLING STATEMENT

```
EXEC disk'get'skip(<number to skip>,<file number>,<file end>)

  <number to skip> is a <numeric expression>, INPUT PARAMETER
    the number of characters from the file to be skipped
  <file number> is a <numeric expression>, INPUT PARAMETER
    the file number of the file to skip characters from
  <file end> is a <numeric variable>, OUTPUT PARAMETER
    set to TRUE (a value of 1) when end of file is reached
    set to FALSE (a value of Ø) while not at end of file
```

EXAMPLES OF CALLING STATEMENT

```
EXEC disk'get'skip(9,infile,file'end)       skips nine characters
EXEC disk'get'skip(c,2,eof2)
```

PROCEDURE LISTING

```
PROC disk'get'skip(count,file'num,REF file'end) CLOSED
  FOR x=1 TO count DO y:=disk'get(file'num,file'end)
ENDPROC disk'get'skip
```

PROCEDURE NAME: DISK'GET'STRING
REQUIRES PROCEDURE: DISK'GET'INIT
REQUIRES FUNCTION: DISK'GET
NOTE: Needed only in version 0.14. Version 2.00 use GET$ instead.

This procedure uses procedure DISK'GET as a function to get a string of the specified number of characters from a previously opened file. Since disk access is through our own machine language program, COMAL does not update the system variable EOF at the end of the file. Therefore, we use the variable FILE'END for the same purpose, assigning it a value of 1 (TRUE) when the file end is reached. This procedure is useful when you are directly reading a disk's directory.

SYNTAX FOR CALLING STATEMENT

```
EXEC disk'get'string(<string>,<num char>,<file>,<file end>)

  <string> is a <string variable>, OUTPUT PARAMETER
    the characters acquired from disk are assigned to this variable
  <num char> is a <numeric expression>, INPUT PARAMETER
    the number of characters to get from the file
  <file> is a <numeric expression>, INPUT PARAMETER
    the file number to get the characters from
  <file end> is a <numeric variable name>, OUTPUT PARAMETER
    set to TRUE (a value of 1) when end of file is reached
    set to FALSE (a value of Ø) while not at end of file
```

EXAMPLES OF CALLING STATEMENT

```
EXEC disk'get'string(s$,9,infile,file'end)     gets nine characters
EXEC disk'get'string(name$,c,2,eof2)
```

PROCEDURE LISTING

```
PROC disk'get'string(REF item$,count,file'num,REF file'end) CLOSED
  item$=""                          initialize the string
  FOR x=1 TO count DO item$(x)=CHR$(disk'get(file'num,file'end))
ENDPROC disk'get'string
```

FUNCTION NAME: EVEN

This function returns a value of 1 (TRUE) if the number passed to it is even, or a value of 0 (FALSE) if it is not.

SYNTAX FOR CALLING STATEMENT

```
even(<numeric expression>)
```

EXAMPLES OF CALLING STATEMENT

```
PRINT even(15)
IF even(number) THEN EXEC skip'line
```

FUNCTION LISTING

```
//
FUNC even(number#) CLOSED
  IF number# MOD 2=Ø THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  ENDIF
ENDFUNC even
```

PROCEDURE NAME: FETCH
REQUIRES PROCEDURES: GET'VALID and GET'CHAR

This procedure is used to get a string of valid characters as input from the keyboard. It is designed to be called from the procedure TAKE'IN, but can be called directly if needed.

SYNTAX FOR CALLING STATEMENT

```
EXEC fetch(<reply>,<valid>,<maximum length>)

  <reply> is a <string variable>, OUTPUT PARAMETER
    the characters in the users reply are assigned to this variable
  <valid> is a <string expression>, INPUT PARAMETER
    a string of the characters to be allowed as part of the reply
      <valid>="a" means all letters plus space are allowed
      <valid>="d" means all digits are allowed
      <valid>="b" means letters, digits, and space are allowed
      all other <valid> strings are not altered
        <carriage return> and <delete> are added to all <valid> sets
  <maximum length> is a <numeric expression>, INPUT PARAMETER
    the maximum number of characters allowed in the reply
```

EXAMPLES OF CALLING STATEMENT

```
EXEC fetch(answer$,"A",5)
EXEC fetch(n$,"B",4Ø)
EXEC fetch(y'or'n$,"YN",1)
```

PROCEDURE LISTING

```
//
PROC fetch(REF a$,v$,max) CLOSED
  IMPORT get'valid            not used in version Ø.14
  DIM valid$ OF 4Ø, b$ OF 1
  z:=ZONE                     remember current ZONE
  ZONE Ø                      set ZONE for tab in each column
  a$:=""                      initialize
  CASE v$ OF                  the cases can be changed to your needs
  WHEN "A"
    valid$:="ABCDEFGHIJKLMNOPQRSTUVWXYZ "   include final space
  WHEN "D"
    valid$:="Ø123456789"
  WHEN "B"
    valid$:="ABCDEFGHIJKLMNOPQRSTUVWXYZ Ø123456789"   include space
  OTHERWISE
    valid$:=v$
  ENDCASE
  done:=FALSE; num:=Ø           initialize
  REPEAT
    EXEC get'valid(b$,valid$+CHR$(13)+CHR$(2Ø))    add CR & DELETE
    CASE b$ OF
    WHEN CHR$(13)               carriage return
      done:=TRUE                end of input
    WHEN CHR$(2Ø)               delete
      IF num THEN               do only if already have something
        num:-1                  subtract one from the string length
        PRINT "[LEFT][SPACE][LEFT]",   cursor LEFT,SPACE,cursor LEFT
        a$:=a$(1:num)           reassign string without last character
      ENDIF
    OTHERWISE                   all other valid characters
      IF num<max THEN           don't go past maximum
        a$:=a$+b$               add character to the string
        num:+1                  add 1 to the count of characters
        PRINT b$,               print the character just hit
      ENDIF
    ENDCASE
  UNTIL done
  ZONE z                        reset ZONE to the original setting
ENDPROC fetch
```

FUNCTION NAME: FILE'EXISTS

This function is used to determine whether or not a file already exists on disk. If the file exists, it returns the value 1 (TRUE), if not, it returns the value 0 (FALSE).

SYNTAX FOR CALLING STATEMENT

```
file'exists(<filename>)

  <filename> is a an INPUT PARAMETER
```

EXAMPLES OF CALLING STATEMENT

```
IF file'exists(file'name$) THEN EXEC notify
IF NOT file'exists(name$) THEN
```

FUNCTION LISTING (version 0.14)

```
//
FUNC file'exists(file'name$) CLOSED
  DIM s$ of 2
  OPEN FILE 78,file'name$,READ
  s$:=status$ // only first two characters are assigned
  CLOSE FILE 78
  IF s$:="62" THEN
    RETURN FALSE
  ELSE
    RETURN TRUE
  ENDIF
ENDFUNC file'exists
```

VERSION 2.00 ONLY MAY USE THE FOLLOWING FUNCTION

```
FUNC file'exists(file'name$) CLOSED
  TRAP
    OPEN FILE 78,file'name$,READ
    CLOSE FILE 78
    RETURN TRUE
  HANDLER
    CLOSE FILE 78
    RETURN FALSE
  ENDTRAP
ENDFUNC file'exists
```

PROCEDURE NAME: GET'CHAR

This procedure is used to get one character from the keyboard without the need for hitting the RETURN key. It will wait until a key is hit, and that character is then assigned to the specified variable. If you only want to look at the keyboard once, as with Commodore BASIC's GET, use the procedure SCAN. Note that the keyboard buffer is cleared before beginning to wait for a character. This eliminates any "type ahead" problems. Two different procedures are listed, both with the same results. One is for version 0.14, the other uses some enhanced features of version 2.00.

SYNTAX FOR CALLING STATEMENT

```
EXEC get'char(<character>)

  <character> is a <string variable>, OUTPUT PARAMETER
    the character of the key hit is assigned to this variable
```

EXAMPLES OF CALLING STATEMENT

```
EXEC get'char(a$)
EXEC get'char(reply$)
```

PROCEDURE LISTING (version 0.14 PET/CBM)

```
//
PROC get'char(REF c$) CLOSED
  buffer'count'loc:=158            buffer count - C64 use 198
  buffer'loc:=623                  keyboard buffer - C64 use 631
  POKE buffer'count'loc,0 //       clear keyboard buffer
  REPEAT
  UNTIL PEEK(buffer'count'loc) // wait till count>0
  c$:=CHR$(PEEK(buffer'loc)) //    assign string
  POKE buffer'count'loc,0          reset buffer count to 0
ENDPROC get'char
```

PROCEDURE LISTING (version 0.14 C64 and all versions 2.00)

```
PROC get'char(REF c$) CLOSED
  REPEAT
    c$:=KEY$
  UNTIL c$:=""                        clear keyboard buffer
  REPEAT
    c$:=KEY$
  UNTIL c$>""                         wait till key is hit
ENDPROC get'char
```

FUNCTION NAME: GET'RECORD$
NOTE: Version 2.00 only. Other versions do not support string functions.

This function will return the complete record for the record number specified. It is returned as one long string.

SYNTAX FOR CALLING STATEMENT

```
get'record$(<filenum>,<record#>)

  <filenum> is an INPUT PARAMETER
  <record#> is a <numeric expression>, INPUT PARAMETER
```

EXAMPLES OF CALLING STATEMENT

```
rec$:=get'record$(2,num)
put'record(out,recl,get'record$(inp,rec2)) // moves a record
```

FUNCTION LISTING

```
// courtesy of UniComal Denmark
FUNC get'record$(filenum,record#) CLOSED
  z:=ZONE                 remember original zone
  ZONE Ø                  no extra spaces after a comma
  // set position in the file
  PRINT FILE filenum,record#: "",
  ZONE z                  back to original zone
  RETURN GET$(filenum,254)
ENDFUNC get'record$
```

PROCEDURE NAME: GET'VALID
REQUIRES PROCEDURE: GET'CHAR

This procedure is used to get one character from the keyboard that is considered valid. It will ignore all characters not specified as valid characters. It calls procedure GET'CHAR actually to get the character.

SYNTAX FOR CALLING STATEMENT

```
EXEC get'valid(<reply>,<valid>)

  <reply> is a <string variable>, OUTPUT PARAMETER
    the character of the key hit is assigned to this variable
  <valid> is a <string expression>, INPUT PARAMETER
    the string of all characters to be considered a valid reply
```

EXAMPLES OF CALLING STATEMENT

```
EXEC get'valid(choice$,v$)
EXEC get'valid(loc$,"SP")
```

PROCEDURE LISTING

```
//
PROC get'valid(REF c$,valid$) CLOSED
  IMPORT get'char            not used in version 0.14
  REPEAT
    EXEC get'char(c$)
  UNTIL c$ IN valid$
ENDPROC get'valid
```

FUNCTION NAME: JIFFIES
NOTE: For version 0.14 only. Version 2.00 use the keyword TIME.

This function will return the current number of jiffies (60 jiffies in 1 second) using the Commodore real time clock. Since it is a function, you do not use EXEC when calling it. Simply use it as a numeric function (see the sample calling statements below).

SYNTAX FOR THE FUNCTION CALL

```
jiffies
```

EXAMPLES OF THE FUNCTION CALL

```
PRINT jiffies
seconds:=jiffies DIV 60
IF jiffies>100 THEN EXEC try'it
```

PROCEDURE LISTING

```
//
FUNC jiffies CLOSED
  j:=256*256*PEEK(141)+256*PEEK(142)+PEEK(143) // PET/CBM
  // j:=256*256*PEEK(160)+256*PEEK(161)+PEEK(162) // C64
  RETURN j
ENDFUNC jiffies
```

NOTE

C64 COMAL: Do not type in the line ending with //PET/CBM. Type in the following line, but do not include the leading//.

PROCEDURE NAME: LOWER'TO'UPPER

This procedure converts any lower case letter in the specified string to UPPER case. Any other character is not affected. There are several ways to accomplish this conversion. This method was chosen since it demonstrates string handling and the use of the keyword IN to pick out a specific charater from a string.

SYNTAX FOR CALLING STATEMENT

```
EXEC lower'to'upper(<string>)

  <string> is a <string variable>, UNIVERSAL INPUT/OUTPUT PARAMETER
    any lower case characters in this string will be converted
```

EXAMPLES OF CALLING STATEMENT

```
EXEC lower'to'upper(name$)
EXEC lower'to'upper(text$)
```

PROCEDURE LISTING

```
//
PROC lower'to'upper(REF a$) CLOSED
  DIM z$ OF 26, c$ OF 1
  z$:="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  FOR x:=1 TO LEN(a$) DO
    c$:=a$(x)                          one character at position X
    IF c$>="a" AND c$<="z" THEN        skip all other characters
      c$:=z$(c$ IN "abcdefghijklmnopqrstuvwxyz")   convert to UPPER
      a$(x):=c$              put the upper into the original string
    ENDIF
  ENDFOR x
ENDPROC lower'to'upper
```

VERSION 2.00 MAY USE THE STRING FUNCTION BELOW:

EXAMPLES OF CALLING STATEMENT

```
PRINT upper$(name$)          print name$ in caps
text$:=upper$(text$)         convert a string into upper case
```

FUNCTION LISTING

```
//
FUNC upper$(a$) CLOSED
  DIM z$ OF 26, c$ OF 1
  z$:="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  FOR x:=1 TO LEN(a$) DO
    c$:=a$(x)                          one character at position X
    IF c$>="a" AND c$<="z" THEN        skip all other characters
      c$:=z$(c$ IN "abcdefghijklmnopqrstuvwxyz")   convert to UPPER
      a$(x):=c$                        put the upper into the original string
    ENDIF
  ENDFOR x
  RETURN a$
ENDFUNC upper$
```

PROCEDURE NAME: MOUNT
NOTE: Version 0.14 only. Version 2.00 use the MOUNT statement.

This procedure is similar to the MOUNT statement in version 2.00. It initializes the disk drive you specify.

SYNTAX FOR CALLING STATEMENT

```
EXEC mount(<drive>)

  <drive> is a <numeric expression>, INPUT PARAMETER
    it is the drive to be initialized
```

EXAMPLES OF CALLING STATEMENT

```
EXEC mount(0)
EXEC mount(drive'number)
```

PROCEDURE LISTING

```
//
PROC mount(drive) CLOSED
  PASS "i"+str$(drive)
ENDPROC mount
```

PROCEDURE NAME: POS

This procedure will return the row and column of the location of the cursor. Thus at any time your program can find out where the cursor is.

NOTES

(1) The ⟨line⟩ is specified first, and the ⟨column⟩ (or position in the line) second.
(2) The screen lines are referred to as lines 1-25 with the top line as line 1.
(3) The columns (positions) in the line are referred to as columns 1-40 on 40 column displays and columns 1-80 on 80 column displays.

SYNTAX FOR CALLING STATEMENT

```
EXEC pos(<line>,<column>)

  <line> is a <variable name>, OUTPUT PARAMETER
  <column> is a <variable name>, OUTPUT PARAMETER
```

EXAMPLES OF CALLING STATEMENT

```
EXEC pos(row,col)
EXEC pos(line,cur'position)
```

PROCEDURE LISTING

```
//
PROC pos(REF line,REF col) CLOSED
  line'peek:=216                C64 use 214
  col'peek:=198                 C64 use 211
  line:=PEEK(line'peek)+1
  col:=PEEK(col'peek)+1
ENDPROC pos
```

PROCEDURE NAME: PUT'RECORD

This procedure will write a string into the specified record number in a previously opened random file. This is useful when moving records around a random file.

SYNTAX FOR CALLING STATEMENT

```
put'record(<filenum>,<record#>,<string$>)

  <filenum> is an INPUT PARAMETER
  <record#> is a positive <numeric expression>, INPUT PARAMETER
  <string$> is a <string expression>, INPUT PARAMETER
```

EXAMPLES OF CALLING STATEMENT

```
put'record(2,rec,text$)
put'record(out,recl,get'record$(inp,rec2)) // moves a record
```

PROCEDURE LISTING

```
// courtesy of UniComal Denmark
PROC put'record(filenum,record#,text$) CLOSED
  z:=ZONE                 remember the old zone
  ZONE Ø                  no extra spaces after a comma
  PRINT FILE filenum,record#: text$,
  ZONE z                  back to original zone
ENDPROC put'record
```

PROCEDURE NAME: RANDOMIZE
NOTE: Version 0.14 only. Version 2.00 use the RANDOMIZE statement.

This procedure does the same thing as the RANDOMIZE statement in version 2.00. It allows you to set a certain pseudo random sequence, or a completely random one. This procedure does not return a number, but merely seeds the random number generator.

NOTES

(1) To set a pseudo-random sequence, execute this procedure once at the beginning of your program with a specific number (not zero). Then in the rest of the program use RND(1) for random numbers, or RND(n1,n2) for a range of random integers.
(2) To set a true random sequence, execute this procedure once at the beginning of your program, specifying 0 as the seed. Then in the rest of the program use RND(1) for random numbers, or RND(n1,n2) for a range of random integers.

SYNTAX FOR CALLING STATEMENT

```
EXEC randomize(<seed>)      OR      EXEC randomize

  <seed> is a <numeric expression>, INPUT PARAMETER
```

EXAMPLES OF CALLING STATEMENT

```
EXEC randomize(Ø)         with seed - start true random sequence
EXEC randomize(15)        with seed - start pseudo random sequence
EXEC randomize          without seed - start true random sequence
```

PROCEDURE LISTING (with 1 parameter)

```
//
PROC randomize(seed) CLOSED
  seed:=ABS(seed)
  IF seed=Ø THEN seed:=256*256*PEEK(141)+256*PEEK(142)+PEEK(143)
  // PET/CBM use the above line - C64 use the line below
  // IF seed=Ø THEN seed:=256*256*PEEK(16Ø)+256*PEEK(161)+PEEK(162)
  dummy:=RND(-seed)
ENDPROC randomize
```

PROCEDURE LISTING (without parameters)

```
//
PROC randomize CLOSED
  dummy:=RND(-256*256*PEEK(141)-256*PEEK(142)-PEEK(143)) // PET/CBM
  // dummy:=RND(-256*256*PEEK(16Ø)-256*PEEK(161)-PEEK(162) // C64
ENDPROC randomize
```

FUNCTION NAME: ROUND

This function will round any number to the nearest integer, returning that integer.

SYNTAX FOR CALLING STATEMENT

```
round(<number>)

  <number> is a <numeric expression>, INPUT PARAMETER
```

EXAMPLES OF CALLING STATEMENT

```
PRINT round(number)
cash:=round(money/people)
```

FUNCTION LISTING

```
//
FUNC round(number) CLOSED
  RETURN SGN(number)*INT(ABS(number)+.5)
ENDFUNC round
```

PROCEDURE NAME: SCANKEY
NOTE: Version 0.14 PET/CBM only. Version 2.00 and C64 use KEY$ instead.

This procedure is similar to the GET command in Commodore BASIC. It looks at the keyboard buffer once. If a key has been hit, it takes it. If not, it returns CHR$(0). It is a nice way to check if any keys were pressed while the program was busy doing other things.

SYNTAX FOR CALLING STATEMENT

```
EXEC scankey(<character>)

  <character> is a <string variable>, OUTPUT PARAMETER
    if a key was hit, its character is assigned to this variable
```

EXAMPLES OF CALLING STATEMENT

```
EXEC scankey(move$)
EXEC scankey(a$)
```

PROCEDURE LISTING

```
//
PROC scankey(REF c$) CLOSED
  buffer'count'loc:=158              C64 use 198
  buffer'loc:=623                    C64 use 631
  a:=PEEK(buffer'count'loc)          number of keys hit
  IF a THEN                          at least one key was hit
    POKE buffer'count'loc,a-1        decrement count
    c$:=CHR$(PEEK(buffer'loc-1+a)) last key hit character
  ELSE                               no key was hit
    c$:=CHR$(Ø)                      same as KEY$ would return
  ENDIF
ENDPROC scankey
```

FUNCTION NAME: SCREEN'POS
NOTE: This function is not for the C64 computer versions.

This function can be used to find the exact memory location of the position on the screen specified by the ⟨line⟩ and ⟨column⟩ parameters. The top screen line is referred to as line number 1, and the first position in a line is position number 1. Its main use would be for PEEKing and POKEing screen locations (it does not affect the cursor position). It is set up to be used as a function, so it does not need a calling EXEC statement. Simply code it as a function, specifying a line and column as its parameters.

NOTES

(1) The ⟨line⟩ is specified first, and the ⟨column⟩ (or position in the line) second.
(2) The screen lines are referred to as lines 1-25 with the top line as line 1.
(3) The columns (postions) in the line are referred to as columns 1-40 on 40 column displays and columns 1-80 on 80 column displays.

SYNTAX FOR THE FUNCTION CALL

```
screen'pos(<line>,<column>)

  <line> is a <numeric expression>, INPUT PARAMETER
    the line specified
  <column> is a <numeric expression>, INPUT PARAMETER
    the position on the specified line
```

EXAMPLES OF THE FUNCTION CALL

```
POKE screen'pos(1,1),32
a:=PEEK(screen'pos(row,col))
```

FUNCTION LISTING

```
//
FUNC screen'pos(line,col) CLOSED
  video'start:=32768              start of screen memory
  line'length:=80                 change to 40 for 40 column computer
  s:=(line-1)*line'length+col+(video'start-1)
  RETURN s
ENDFUNC screen'pos
```

FUNCTION NAME: SHIFT

This function may be used anytime you wish to check if the SHIFT key is depressed or not. It will return a value of TRUE (a value of 1) if the SHIFT key is depressed, or a value of FALSE (a value of 0) if the SHIFT key is not depressed. Anytime you use the word SHIFT in an expression, the system will look at the SHIFT key and return the appropriate value. To wait till the SHIFT key is actually depressed use procedure SHIFT'WAIT.

SYNTAX FOR THE FUNCTION CALL

```
shift
```

EXAMPLES OF THE FUNCTION CALL

```
PRINT shift
IF shift THEN EXEC changes
```

FUNCTION LISTING

```
//
FUNC shift CLOSED
  shift'flag:=152 //              C64 use 653
  RETURN PEEK(shift'flag) //      PET/CBM
  // RETURN PEEK(shift'flag)=1 // C64
  // WITH THE C64 PEEK(653) YEILDS THE FOLLOWING INFORMATION:
  // 1=SHIFT Key  2=Commodore Key  4=CONTROL Key  Ø=None of them
ENDFUNC shift
```

PROCEDURE NAME: TAKE'IN
REQUIRES PROCEDURES: FETCH, GET'VALID, and GET'CHAR

This procedure provides an organized method of getting input from the keyboard. It will print a prompt for you, provide a defined blank area for the input, and only accept valid characters. It calls procedure FETCH actually to get the keyboard input.

SYNTAX FOR CALLING STATEMENT

```
EXEC take'in(<prompt>,<valid>,<reply>,<maximum length>)

  <prompt> is a <string expression>, INPUT PARAMETER
    the string that will be printed as the prompt
  <valid> is a <string expression>, INPUT PARAMETER
    the string of characters that will be considered valid
      <valid>="a" means all the letters plus space are allowed
      <valid>="d" means all the digits are allowed
      <valid>="b" means letters, digits and space are allowed
        all other <valid> strings are not changed
        <carriage return> and <DELETE> are added to all valid sets
  <reply> is a <string variable>, OUTPUT PARAMETER
    the characters of the reply are assigned to this variable
  <maximum length> is a <numeric expression>, INPUT PARAMETER
    the maximum number of characters allowed in the reply
```

EXAMPLES OF CALLING STATEMENT

```
EXEC take'in("NAME:","A",name$,20)
EXEC take'in(p$,v$,text$,max)
EXEC take'in("SCORE:","D",score$,5)
```

PROCEDURE LISTING

```
//
PROC take'in(prompt$,valid$,REF reply$,max) CLOSED
  IMPORT fetch                              not used in version Ø.14
  z:=ZONE                                   remember current zone
  ZONE Ø                                    set ZONE for no tabs
  PRINT prompt$,                            print the prompt
  PRINT "<",                                define left of input area
  FOR x:=1 TO max DO PRINT " ",             blank out input area
  PRINT ">",                                define right of input area
  FOR x:=1 TO max+1 DO PRINT "[LEFT]",      cursor left to first spot
  EXEC fetch(reply$,valid$,max)             get the input
  PRINT                                     carriage return
  ZONE z                                    reset ZONE to original
ENDPROC take'in
```

FUNCTION NAME: VALUE
NOTE: Needed only in version 0.14. Version 2.00 use VAL.

This function will take a string of digits and convert it to its numeric equivalent. It will only work with a string of digits that represent an integer. This procedure is courtesy of Borge Christensen of Denmark. Version 2.00 do not need this function since VAL is a built in function.

SYNTAX FOR THE FUNCTION CALL

```
value(<digits>)

  <digits> is a <string expression>, INPUT PARAMETER
    the string of digits to be converted to a numeric value
```

EXAMPLES OF THE FUNCTION CALL

```
age:=value(temp$)
x:=value("462")
```

FUNCTION LISTING

```
//
FUNC value(s$) CLOSED
  length:=LEN(s$)
  ones:=ORD(s$(length))-ORD("Ø")
  IF length:=1 THEN
    RETURN ones
  ELSE
    RETURN ones+value(s$(1:length-1))*1Ø      recursive call
  ENDIF
ENDFUNC value
```

APPENDIX E
OPERATORS

An expression in COMAL may have multiple operations. These operations are performed according to a predefined sequence of eight levels of precedence. Operations are performed with the operator of the highest precedence first. If there are multiple operators of the same level of precedence, they are performed from left to right. Parentheses may be used to override the predefined sequence, as parentheses are the highest level of precedence. The chart below illustrates the levels of precedence in CBM COMAL:

PRECEDENCE	OPERATOR	TYPE	MEANING	EXAMPLE
HIGHEST				
1	( )		PARENTHESES	(A-B)*C
2	^	ARITHMETIC	EXPONENTIATION	A^B
3	*	ARITHMETIC	MULTIPLICATION	A*B
3	/	ARITHMETIC	DIVISION	A/B
3	DIV	ARITHMETIC	INTEGER DIVISION	A DIV B
3	MOD	ARITHMETIC	REMAINDER FROM DIVISION	A MOD B
4	+	ARITHMETIC	ADDITION	A+B
4	-	ARITHMETIC	SUBTRACTION	A-B
4	-	ARITHMETIC	UNARY MINUS	-A
5	BITAND	DYADIC	BITWISE AND	A BITAND B
5	BITOR	DYADIC	BITWISE OR	A BITOR B
5	BITXOR	DYADIC	BITWISE XOR	A BITXOR B
6	=	RELATIONAL	EQUAL	A=B
6	<>	RELATIONAL	NOT EQUAL	A<>B
6	<	RELATIONAL	LESS THAN	A<B
6	>	RELATIONAL	GREATER THAN	A>B
6	<=	RELATIONAL	LESS THAN OR EQUAL	A<=B
6	>=	RELATIONAL	GREATER THAN OR EQUAL	A>=B
6	IN	RELATIONAL	SUBSTRING POSITION	A$ IN B$
7	NOT	BOOLEAN	LOGICAL NEGATION	NOT A
8	AND	BOOLEAN	LOGICAL CONJUNCTION	A AND B
9	OR	BOOLEAN	LOGICAL DISJUNCTION	A OR B
LOWEST				

APPENDIX F
ERROR MESSAGE FILE GENERATOR

```
print "generate comal error messages"
//
print
dim message$ of 255
open file 2,"@0:comalerrors",write
//
while not eod do
 read errno,severity,message$
 print file 2: chr$(errno),chr$(len(message$)),chr$
 (severity),message$,
endwhile
close
end
//
//data format:
//
// <error number>,<severity>,<error message>
//
// <error number>  : an internal number used by
//                   the interpreter.
//
// <severity>      : 0: not severe error.
//                      the stack is unchanged,
//                      and you can 'con'-tinue.
//
//                   1: severe error.
//                      the stack is reset, all
//                      variables become undeclared
,
//                      and you cannot 'con'-tinue.
//
// <error message> : the message which is diplayed
//                   if the error occurs.
//
//
// start-of-data
//
```

```
data 0,0,"format error"
data 1,0,"syntax error"
data 2,0,"type conflict"
data 3,0,"function argument error"
data 4,0,"statement too long or too complicated"
data 5,1,"system error"
data 6,0,"name too long"
data 7,0,"bracket error"
data 8,0,"overflow"
data 9,0,"error in structured statement"
data 10,0,"error in goto statement"
data 11,1,"stack overflow"
data 12,0,"unknown variable"
data 13,1,"procedure param error"
data 14,1,"index/param error"
data 15,0,"substring error"
data 16,0,"command, array, substring, or procedure
error"
data 17,0,"index error"
data 18,0,"illegal no. of indices"
data 19,0,"string assignment error"
data 20,0,"function argument error"
data 21,1,"not implemented"
data 22,0,"zone value incorrect"
data 23,0,"step = 0"
data 24,1,"array redefined"
data 25,1,"dimension error"
data 26,0,"case error"
data 27,0,"end of data"
data 28,0,"file already open"
data 29,0,"file input error"
data 30,0,"end-of-file"
data 31,0,"file not open"
data 32,1,"con not possible"
data 33,1,"error in print using"
data 34,0,"division by zero"
data 35,1,"program not prepassed"
data 36,0,"file not found"
data 37,1," "
data 38,1,"not input file"
data 39,1,"device not present"
data 40,1,"not output file"
data 41,0,"string not dimensioned"
data 42,1,"local variable error"
data 52,0,"too many names"
data 53,1,"function value not returned"
data 54,0,"not a statement"
data 55,0,"not a command or simple statement"
data 56,0,"',' expected"
data 57,0,"number out of range"
data 58,0,"expression expected"
data 59,0,"not implemented"
data 60,0,"operand expected"
data 91,0,"user error #1"
data 92,0,"user error #2"
//
// end-of-data
//
```

```
// generate error message file for 0.14 commodore 64
DIM a$ OF 80
OPEN FILE 1,"@0:comalerrors",WRITE
WHILE NOT EOD DO
  READ errno,lng,sev,a$
  PRINT FILE 1: CHR$(errno),CHR$(lng),CHR$(sev),a$,
ENDWHILE
CLOSE
END
DATA 0,12,0,"format error"
DATA 1,12,0,"syntax error"
DATA 2,13,0,"type conflict"
DATA 3,23,0,"function argument error"
DATA 4,37,0,"statement too long or too complicated"
DATA 5,12,1,"system error"
DATA 6,13,0,"name too long"
DATA 7,13,0,"bracket error"
DATA 8,8,0,"overflow"
DATA 9,29,0,"error in structured statement"
DATA 10,23,0,"error in goto statement"
DATA 11,14,1,"stack overflow"
DATA 12,16,0,"unknown variable"
DATA 13,21,1,"procedure param error"
DATA 14,17,1,"index/param error"
DATA 15,15,0,"substring error"
DATA 16,45,0,"command, array, substring, or procedure error"
DATA 17,11,0,"index error"
DATA 18,22,0,"illegal no. of indices"
DATA 19,23,0,"string assignment error"
DATA 20,23,0,"function argument error"
DATA 21,15,1,"not implemented"
DATA 22,20,0,"zone value incorrect"
DATA 23,8,0,"step = 0"
DATA 24,15,1,"array redefined"
DATA 25,15,1,"dimension error"
```

```
DATA 26,10,0,"case error"
DATA 27,11,0,"end of data"
DATA 28,17,0,"file already open"
DATA 29,16,0,"file input error"
DATA 30,11,0,"end-of-file"
DATA 31,13,0,"file not open"
DATA 32,16,1,"con not possible"
DATA 33,20,1,"error in print using"
DATA 34,16,0,"division by zero"
DATA 35,21,1,"program not prepassed"
DATA 36,14,0,"file not found"
DATA 37,1,1," "
DATA 38,14,1,"not input file"
DATA 39,18,1,"device not present"
DATA 40,15,1,"not output file"
DATA 41,22,0,"string not dimensioned"
DATA 42,20,1,"local variable error"
DATA 52,14,0,"too many names"
DATA 53,27,1,"function value not returned"
DATA 54,15,0,"not a statement"
DATA 55,33,0,"not a command or simple statement"
DATA 56,12,0,"',' expected"
DATA 57,19,0,"number out of range"
DATA 58,19,0,"expression expected"
DATA 59,15,0,"not implemented"
DATA 60,16,0,"operand expected"
DATA 91,13,0,"user error #1"
DATA 92,13,0,"user error #2"
DATA 100,18,1,"graphic not active"
DATA 101,13,0,"illegal color"
DATA 102,24,0,"illegal plot coordinates"
```

APPENDIX G
VERSION 2.00 COMAL ERROR NUMBERS AND THEIR MESSAGES
(Courtesy of UniComal)

Runtime errors, which can be trapped:

```
000:  no error to report
001:  function argument error
002:  overflow
003:  division by zero
004:  substring error
005:  value out of range
006:  step = 0
007:  illegal bound
008:  error in print using
009:  error
010:  index out of range
011:  file name too long
012:  error
013:  verify error
014:  program too big
015:  bad comal code
016:  not comal program file
017:  program made for other comal version
018:  unknown file attribute
```

Runtime errors, which cannot be trapped:

```
051:  system error
052:  out of memory
053:  wrong dimension in parameter
054:  parameter must be an array
055:  too few indices
056:  string assignment error
057:  not implemented
058:  con not possible
059:  program has been modified
060:  too many indices
061:  function value not returned
062:  not a variable
063:  error
064:  error
065:  error
066:  error
067:  parameter lists differ or not closed
068:  no closed proc/func in file
069:  too few parameters
070:  wrong index type
071:  parameter must be a variable
072:  wrong parameter type
073:  non-ram load
074:  checksum error in object file
075:  memory area is protected
076:  too many libraries
077:  not an object file
078:  no matching when
079:  too many parameters
```

Syntax error messages:

```
101:  syntax error
102:  wrong type
103:  statement too long or too complicated
104:  statement only, not command
105:  error
106:  line number range: 1 to 9999
107:  error
108:  procedure/function does not exist
109:  structured statement not allowed here
110:  not a statement
111:  line numbers will exceed 9999
112:  source protected!!!
113:  illegal character
114:  error in constant
115:  error in exponent
```

Input/output error messages (can all be trapped):

```
200:  end of data
201:  end of file
202:  file already open
203:  file not open
204:  not input file
205:  not output file
206:  numeric constant expected
207:  not random access file
208:  device not present
209:  too many files open
210:  read error
211:  write error
212:  short block on tape
213:  long block on tape
214:  checksum error on tape
215:  end of tape
216:  file not found
217:  unknown device
218:  illegal operation
219:  i/o break
```

From 220-274 is disk error messages placed (i.e., disk error number + 200):

```
220:  read error   (block header not found)
221:  read error   (no sync character)
222:  read error   (data block not present)
223:  read error   (checksum error in data block)
224:  read error   (byte decoding error)
225:  write error  (write-verify error)
226:  write protect on
227:  read error   (checksum error in header)
228:  write error  (long data block)
229:  disk id mismatch
230:  syntax error (general syntax)
231:  syntax error (invalid command)
232:  syntax error (long line)
233:  syntax error (invalid file name)
234:  syntax error (no file given)
239:  syntax error (invalid command)
250:  record not present
251:  overflow in record
252:  file too large
260:  write file open
261:  file not open
262:  file not found
263:  file exists
264:  file type mismatch
265:  no block
266:  illegal track and sector
267:  illegal system t or s
270:  no channel   (available)
271:  directory error
272:  disk full
273:  4040: cbm dos v2    (dos mismatch)
      8050: cbm dos v2.5
274:  drive not ready
```

Dynamic syntax error messages:

```
<symbol> not expected
<symbol> missing
<symbol1> expected, not <symbol2>
```

Dynamic structure scan error messages:

```
<statement1> without <statement2>
<statement> missing
<statement1> expected, not <statement2>
<statement> not allowed in control structures
import allowed in closed proc/func only
wrong type of <statement>
wrong name in <statement>
<name>: name already defined
<name>: unknown label
illegal goto
```

Dynamic runtime error messages: (cannot be trapped)

```
<name>: unknown statement or procedure
<name>: not a procedure
<name>: unknown variable
<name>: wrong type
<name>: wrong function type
<name>: not an array nor a function
<name>: not a variable
<name>: unknown array or function
<name>: wrong array type
<name>: import error
<name>: unknown package
<name>: array redefined
<name>: unknown label
<name>: name already defined
<name>: not a label
<name>: string not dimensioned
<name>: not a package
```

APPENDIX H
COMAL DEFINITION — THE COMAL KERNAL

```
<comal program> ::=
            <block>

<block> ::=
           {<declaration statement> |
            <non declaration statement>}

<declaration statement> ::=
            <structured declaration statement> |
            <unstructured declaration statement>

<non declaration statement> ::=
            <structured statement> |
            <unstructured statement>

<structured declaration statement> ::=
            <procedure declaration> |
            <function declaration>

<unstructured declaration statement> ::=
            <dim statement> |
            <data statement>

<structured statement> ::=
            <repetitive statement> |
            <conditional statement>

<unstructured statement> ::=
            <simple statement> <eol> |
            <remark> <newline> |
            <label statement> <eol>

<eol> ::=
            [<remark>] <newline>

<remark> ::=
            //{<displayable character>}

<newline> ::=
            implementation dependent

<displayable character> ::=
            implementation dependent
```

```
<additive operator> ::=
            + | -

<term> ::=
            [<term> <multiplicative operator>] <factor>

<multiplicative operator> ::=
            * |
            / |
            DIV |
            MOD

<factor> ::=
            <operand> [ <factor>]

<operand> ::=
            (<numeric expression>) |
            <constant> |
            <numeric variable> |
            <numeric function call>

<constant> ::=
            <integer> | <real number> |
            TRUE | FALSE

<real number> ::=
            <decimal number> [<exponent>]

<decimal number> ::=
            <integer> [. [<integer>]] |
            .<integer>

<exponent> ::=
            E [<sign>] <integer>

<integer> ::=
            <digit>{<digit>}

<numeric variable> ::=
            <numeric identifier> [(<subscript list>)]

<numeric identifier> ::=
            <integer identifier> |
            <real identifier>

<integer identifier> ::=
            <identifier>#

<real identifier> ::=
            <identifier>

<subscript list> ::=
            <subscript> {,<subscript>}

<subscript> ::=
            <numeric expression>

<numeric function call> ::=
            <numeric identifier> [(<actual parameter list>)]
```

```
<string expression> ::=
            <string operand> {+ <string operand>}

<string operand> ::=
            <string constant> |
            <string variable> |
            <string function call>

<string constant> ::=
            "{<displayable character>}"

<string variable> ::=
            <string identifier> [(<subscript list>)]
            [(<substring specifier>)]

<string identifier> ::=
            <identifier>$

<substring specifier> ::=
            <position> | <from>:<to>

<position> ::=
            <numeric expression>

<from> ::=
            <numeric expression>

<to> ::=
            <numeric expression>

<string function call> ::=
            <string identifier>[(<actual parameter list>)]
            [(<substring specifier>)]

<stop statement> ::=
            STOP

<return statement> ::=
            RETURN [<expression>]

<assignment statement> ::=
            <assignment> {;<assignment>}

<assignment> ::=
            <numeric assignment> |
            <string assignment>

<numeric assignment> ::=
            <numeric variable> := <numeric expression>

<string assignment> ::=
            <string variable> := <string expression>

<input statement> ::=
            INPUT [<string constant>:] <variable list> <print end> |
            INPUT <file designator>: <variable list>

<variable list> ::=
            <variable> {,<variable>}
```

```
<variable> ::=
            <numeric variable> |
            <string variable>

<file designator> ::=
            FILE <channel number> [,<record number>]

<channel number> ::=
            <numeric expression>

<record number> ::=
            <numeric expression>

<goto statement> ::=
            GOTO <label identifier>

<restore statement> ::=
            RESTORE [<label identifier>]

<select statement> ::=
            SELECT <type> <device specifier> [,<dev info>]

<type> ::=
            OUTPUT

<device specifier> ::=
            <string expression>

<open statement> ::=
            OPEN FILE <channel number>,<file name>
            [,<dev info>],<mode>

<dev info> ::=
            implementation dependent device information

<file name> ::=
            <string expression>

<mode> ::=
            READ |
            WRITE |
            APPEND |
            RANDOM <record length> [READONLY]

<record length> ::=
            <numeric expression>

<read statement> ::=
            READ <variable list> |
            READ <file designator>: <variable list>

<write statement> ::=
            WRITE <file designator>: <variable list>

<delete statement> ::=
            DELETE <file name> [,<dev info>]

<close statement> ::=
            CLOSE [FILE <channel number>]
```

```
<print statement> ::=
            PRINT [<output list>] |
            PRINT <file designator>: [<output list>]

<output list> ::=
            <print list> [<print end>]

<print list> ::=
            <print element> {<print separator> <print element>}

<print element> ::=
            <expression> |
            <tab function>

<print end> ::=
            <print separator>

<print separator> ::=
            , | ;

<tab function> ::=
            TAB(<numeric expression>)

<zone statement> ::=
            ZONE <numeric expression>

<print using statement> ::=
            PRINT USING <format info>: <using list> [<print end>] |
            PRINT <file designator>: USING <format info>:
            <using list> [<print end>]

<using list> ::=
            <using element> {,<using element>}

<using element> ::=
            <numeric expression>

<format info> ::=
            <string expression>

<procedure call statement> ::=
            [EXEC] <procedure identifier> [(<actual parameter list>)]

<actual parameter list> ::=
            <actual parameter> {,<actual parameter>}

<actual parameter> ::=
            <expression>

<label statement> ::=
            <label identifier>:

<label identifier> ::=
            <identifier>

<identifier> ::=
            <letter> {<letter> | <digit> | ' }
```

```
<simple statement> ::=
            <stop statement> |
            <return statement> |
            <assignment statement> |
            <input statement> |
            <goto statement> |
            <restore statement> |
            <select statement> |
            <open statement> |
            <read statement> |
            <write statement> |
            <close statement> |
            <delete statement> |
            <print statement> |
            <zone statement> |
            <print using statement> |
            <procedure call statement>

<repetitive statement> ::=
            <while statement> |
            <repeat statement> |
            <for statement>

<conditional statement> ::=
            <if statement> |
            <case statement>

<while statement> ::=
            <short while statement> |
            <long while statement>

<short while statement> ::=
            WHILE <logical expression> DO <simple statement> <eol>

<long while statement> ::=
            WHILE <logical expression> DO <eol>
              <statement list>
            ENDWHILE <eol>


<statement list> ::=
           {<non declaration statement>}

<repeat statement> ::=
            REPEAT <eol>
              <statement list>
            UNTIL <logical expression> <eol>

<for statement> ::=
            <short for statement> |
            <long for statement>

<short for statement> ::=
            FOR <for range> [<step>] DO <simple statement> <eol>

<long for statement> ::=
            FOR <for range> [<step>] DO <eol>
              <statement list>
            NEXT <control variable> <eol>
```

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

<step> ::=
            STEP <step value>

<control variable> ::=
            <numeric identifier>

<initial value> ::=
            <numeric expression>

<final value> ::=
            <numeric expression>

<step value> ::=
            <numeric expression>

<if statement> ::=
            <short if statement> |
            <long if statement>

<short if statement> ::=
            IF <logical expression> THEN <simple statement> <eol>

<long if statement> ::=
            IF <logical expression> THEN <eol>
              <statement list>
            {ELIF <logical expression> THEN <eol>
              <statement list>}
            [ELSE <eol>
              <statement list>]
             ENDIF <eol>

<logical expression> ::=
            <numeric expression>

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

<case selector> ::=
            <expression>

<choice list> ::=
            <numeric expression> {,<numeric expression> } |
            <string expression> {,<string expression>}

<procedure declaration> ::=
            PROC <procedure identifier> <head appendix> <eol>
              <procedure block>
            ENDPROC <procedure identifier> <eol>
```

```
<function declaration> ::=
            FUNC <function identifier> <head appendix> <eol>
              <function block>
            ENDFUNC <function identifier> <eol>

<function block> ::=
            <procedure block>

<procedure block> ::=
            {<import statement>}
            {<unstructured declaration statement> |
            <non declaration statement>}

<head appendix> ::=
           [(<formal parameter list>)] [CLOSED]

<procedure identifier> ::=
            <identifier>

<function identifier> ::=
            <numeric identifier> |
            <string identifier>

<formal parameter list> ::=
            <formal parameter> {,<formal parameter>}

<formal parameter> ::=
            [REF] <variable identifier> |
            REF <variable identifier> <array indicator>

<import statement> ::=
            IMPORT <variable identifier> {,<variable identifier>} <eol>

<variable identifier> ::=
            <numeric identifier> |
            <string identifier>

<array indicator> ::=
            ({,})

<dim statement> ::=
            DIM <declaration> {,<declaration>} <eol>

<declaration> ::=
            <numeric declaration> |
            <string declaration>

<numeric declaration> ::=
            <numeric identifier> (<dimension part>)

<string declaration> ::=
            <string identifier> [(<dimension part>)] OF <length>

<dimension part> ::=
            <range> {,<range>}

<range> ::=
            [<lower bound>:] <upper bound>
```

```
<lower bound> ::=
            <numeric expression>

<upper bound> ::=
            <numeric expression>

<length> ::=
            <numeric expression>

<data statement> ::=
            DATA <value> {,<value>} <eol>

<value> ::=
            [<sign>] <integer> |
            [<sign>] <real number> |
            <string constant> |
            TRUE |
            FALSE

<sign> ::=
            + | -

<expression> ::=
            <string expression> |
            <numeric expression>

<numeric expression> ::=
            [<numeric expression> OR] <logical term>

<logical term> ::=
            [<logical term> AND] <logical factor>

<logical factor> ::=
            [NOT] <relation>

<relation> ::=
            <string relation> |
            <arithmetic relation>

<string relation> ::=
            <string expression> <relational string operator>
            <string expression>

<relational string operator> ::=
            IN | <relational operator>

<arithmetic relation> ::=
            <formula> [<relational operator> <formula>]

<relational operator> ::=
            < | <= | = | >= | > | <>

<formula> ::=
            [<sign>] <arithmetic expression>

<arithmetic expression> ::=
            [<arithmetic expression> <additive operator>] <term>
```

```
<letter> ::=
            implementation dependent

<digit> ::=
            0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

APPENDIX I
RESOURCE LIST

BOOKS and PAPERS

COMAL PROBLEMLOSNING OG PROGRAMMERING (Danish)
COMAL 80, DIE STRUKTURIERTE SPRACHE AUF DER BASIS VON BASIC (German)
BEGINNING COMAL (English)
by Borge Christensen, Published by Ellis Horwood Limited, England

STRUCTURED PROGRAMMING WITH COMAL-80
by Roy Atherton, Published by Ellis Horwood Limited, England

COMAL-80 — ADDING STRUCTURE TO BASIC by Max Bramer
Open University, CAL Research Group Technical Report No. 3, England

MICROCOMPUTERS IN EDUCATION
Chapter 7: COMAL — AN EDUCATIONAL ALTERNATIVE by Borge Christensen
Chapter 8: SOFTWARE STANDARDS IN BASIC AND COMAL by Roy Atherton
Published by Ellis Horwood Limited, England

THE NEW LANGUAGES (PASCAL, COMAL) by Roy Atherton
Curriculum Implications of Micro-Electronics Conference, March 1981

PROGRAMMERING I COMAL 80 by Poul Ostergaard
Published by Teknisk Forlag A-S

FOUNDATIONS IN COMPUTER STUDIES WITH COMAL by John Kelly
Published by The Educational Company of Ireland Limited

MAGAZINES and NEWSLETTERS

COMAL TODAY, published by COMAL Users Group, USA
COMAL Online Information Network, modem line sponsored by COMAL Users Group, USA

COMAL BULLETIN, to be published by Commodore England

ICPUG NEWS, published by ICPUG, England

USER GROUPS

COMAL USERS GROUP, USA
5501 Groveland Ter, Madison, WI 53716, USA

COMAL USERS GROUP, England
Islington Community Computer Centre, Polytechnic of North London, Holloway Road, London N7 8DB England

INDEPENDENT COMMODORE PRODUCTS USER GROUP (ICPUG), England
30 Brancaster Road, Newbury Park, Essex, Ilford IG2 7EP England

Alberta COMAL Users Group, c/o Tom Garraway
Division of Educational Research Services, University of Alberta, Edmunton, Alberta, Canada T6G 2G5

Calgary COMAL Users Group, c/o Robert Allin
47 Strathcona Place SW, Calgary, T3H 1L4 CANADA

COMAL Bruger Gruppe, Mindegade 42, 8700 Horsens, Denmark

WHERE TO BUY COMAL

*** COMAL on disk:
Reston Publishing, 11480 Sunset Hills Road, Reston, VA 22090 (703) 437-8900 or (800) 336-0338
COMAL Users Group, 5501 Groveland Ter, Madison, WI 53716

*** COMAL on PET/CBM plug-in boards:
COMAL Users Group, 5501 Groveland Ter, Madison, WI 53716

*** COMAL C64 cartridge:
Commodore Business Machines Sales Centers

APPENDIX J
TROUBLE SHOOTING HINTS

(1) Are you using file numbers 1, 254, or 255? This could lead to intermittent problems due to conflicts since the system also uses those file numbers.
(2) Is there a procedure or function problem? If it is CLOSED and you are using version 2.00 make sure you use IMPORT for every other procedure or function called from within it.
(3) Do you have a name conflict? Remember that each identifier can only be used for one purpose (excluding packages and CLOSED procedures and functions). Thus if you have a variable called TEST, you cannot also have a procedure with the same name. You also could not have a string variable called TEST$.
(4) All string parameters in functions and procedures are automatically DIMensioned when called, and it is dimensioned for whatever size string is supplied at the start. For example:

```
PROC FETCH(REF A$ ,V$ ,MAX) CLOSED
```

The following statement can be used to call this procedure:

```
EXEC FETCH(REPLY$ ,"YN",1)
```

Now, A$ is used as an alias for the string variable REPLY$. V$ is automatically dimensioned to a maximum length of 2 (like DIM V$ OF 2) and assigned the value "YN". MAX is assigned the value of 1. Now, you would have problems if you tried to add anything to V$ with a statement such as:

```
V$:=V$+ CHR$(13)+ CHR$(20)        add a carriage return and delete
```

After the statement was executed, V$ would still be equal to "YN" since it can only hold two characters. In order to expand V$ you must create a string of a new name, DIMensioning it to the maximum needed, and then assign it a starting value of V$. For example:

```
DIM VALID$ of 40
VALID$ :=V$
```

See Appendix D, procedure FETCH, for an example of this.

(5) Version 0.14 does not recognized SHIFTED alphabet characters. Thus the line might look just fine, but COMAL will not recognize the statement.
(6) Remember that a WRITE file precedes each string with a two byte counter. These two extra bytes must be allowed for in calculating a record length for RANDOM files. See also Appendix L for more information.
(7) Are you using DOS 2.0 ROMs in your 4040 disk unit? This old DOS has some bugs in it including RANDOM file mistakes. If you are using RANDOM files, you should get your DOS upgraded to the current level.
(8) Trouble transferring a program from version 2.00 down to 0.14? Make sure you LIST it to disk in all unshifted letters (i.e., lower case) for keywords and identifiers. Issue this command before listing to disk: POKE $24B,PEEK($24B) BITAND %00111111.
(9) With a 40 column screen, watch out for wrap around lines during a LIST. If you edit the line, or merely hit RETURN on it, the extra spaces used for indentation will be included in the line. Thus to correctly edit a COMAL program with only a 40 column screen you should ALWAYS use EDIT, not LIST. Use LIST only to view the structures.
(10) The SELECT OUTPUT statement changed from versions 0.12 and 1.02 to the current versions 0.14 and 2.00. Previously you selected the data screen as "ds" and the line printer as "lp". Now you *MUST* add a colon after each specifier to be consistent with the current file name system: "ds:" and "lp:". Any program written for versions 0.12 or 1.02 should be checked for this update.

IMPORTANT NOTE

LIST to tape or disk is very useful. If you LIST a program to tape or disk, type NEW, and then ENTER it back again, you will clean up the COMAL NAME TABLE as well as get more free memory. SAVE and LOAD keeps the old NAME table. Each time you make a spelling mistake (such as LIT instead of LIST) that name goes into the table. Version 0.14 has only 255 names allowed, so if you make alot of spelling mistakes, you may get TOO MANY NAMES error. Then you must perform this name table cleanup operation.

APPENDIX K
LINK A MACHINE LANGUAGE PACKAGE TO A COMAL PROGRAM
(version 2.00 only)

COMAL version 2.00 has the capability to LINK one or more machine language packages to a standard COMAL program. In addition, COMAL will SAVE the machine language package together with the COMAL program as one file. Thus all future LOADs of the program will not have to worry about LINKing the packages.

In the following example, we will write a small program that uses a machine language package called "bitpac" and stored on disk by the name "bitpac.obj". Notice that the package itself takes up more space on disk, than when combined with a COMAL program. This is because the package is stored in HEX, while it is in binary when LINKed to a COMAL program.

```
AUTO
0010 // example of bitwise operations
0020 // using hexidecimal numbers
0030 use bitpac // name of package is bitpac
0040 zone 20 // set zone for easy tabbing when printing columns
0050                                  just hit RETURN here
0060 number:=$86 // $86 is hex notation for hex number 86
0070 print "Examples of Bitwise AND, Bitwise OR, and Bitwise XOR"
0080 print
0090 for temp:=$a TO $f DO // hex numbers $0a and $0f
0100 print number;"AND";temp;"is";land(number,temp),
0110 print number;"OR ";temp;"is";lor(number,temp),
0120 print number;"XOR";temp;"is";xor(number,temp)
0130 endfor temp
0140                                  just hit RETURN here
0150 end "Example of Bitwise Operators is Completed"
0160                                  hit STOP key here
```

```
LIST
0010 // example of bitwise operations
0020 // using hexidecimal numbers
0030 USE bit // name of package is bit
0040 ZONE 20 // set zone for easy tabbing when printing columns
0050
0060 number:=$86 // $86 is hex notation for hex number 86
0070 PRINT "Examples of Bitwise AND, Bitwise OR, and Bitwise XOR"
0080 PRINT
0090 FOR temp:=$0a TO $0f DO // hex numbers $0a and $0f
0100   PRINT number;"AND";temp;"is";land(number,temp),
0110   PRINT number;"OR";temp;"is";lor(number,temp),
0120   PRINT number;"XOR";temp;"is";xor(number,temp)
0130 ENDFOR temp
0140
0150 END "Example of Bitwise Operators is Completed"

SAVE "BEFORE"
LINK "BITPAC.OBJ"
SAVE "AFTER"
CAT
 0 "EXAMPLE DISK     " ID 2A
 5    "BITPAC.OBJ"        SEQ
 3    "BEFORE"            PRG
 5    "AFTER"             PRG
 647 BLOCKS FREE.


RUN
Examples of Bitwise AND, Bitwise OR, and Bitwise XOR

134 AND 10 IS 2     134 OR 10 IS 142    134 XOR 10 IS 140
134 AND 11 IS 2     134 OR 11 IS 143    134 XOR 11 IS 141
134 AND 12 IS 4     134 OR 12 IS 142    134 XOR 12 IS 138
134 AND 13 IS 4     134 OR 13 IS 143    134 XOR 13 IS 139
134 AND 14 IS 6     134 OR 14 IS 142    134 XOR 14 IS 136
134 AND 15 IS 6     134 OR 15 IS 143    134 XOR 15 IS 137

Example of Bitwise Operators is Completed
```

APPENDIX L
COMAL DISK FILE USAGE

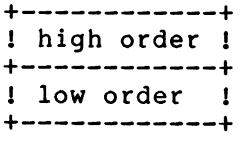
COMAL supports both text input/output and binary input/output data files in normal operation.

BINARY INPUT/OUTPUT FILES

The preferred type of input/output is in binary, because it is compact in size, easier to use, and does not require delimiters between records. There are three different data types: integer, real, and string.

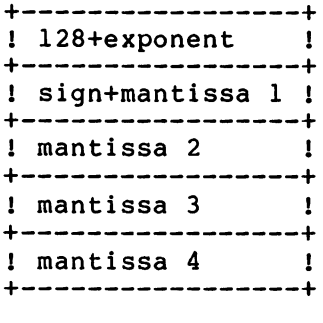
INTEGER TYPE

The integer type is organized as two bytes, regardless of what its value is. It is a signed integer in two's complement:



REAL TYPE

The real type is organized as five bytes, regardless of what its value is. The organization method is called excess 128. These five bytes are arranged as follows:



The sign is the most significant bit of the second byte, as shown above. A zero sign means a positive number, and a one means negative.

The number can be written

```
(1-2*sign)*(2&exponent)*mantissa
```

If the number is zero, then all five bytes are zero. The mantissa is left bit justified so that the most significant bit always is 1, except for the case when the number is zero. Because this bit always is one, it is used to hold the sign.

For example, the real number 3.14159266 would be represented as follows:

```
    decimal            binary
+---------------+   +----------+
!      130      !   ! 10000010 !
+---------------+   +----------+
!       73      !   ! 01001001 !
+---------------+   +----------+
!       15      !   ! 00001111 !
+---------------+   +----------+
!      218      !   ! 11011010 !
+---------------+   +----------+
!      169      !   ! 10101001 !
+---------------+   +----------+
```

STRING TYPE

The string type is organized as the string preceded by a two byte unsigned integer representing the length of the string.

```
+-----------------+
! high order byte !
+-----------------+
! low order byte  !
+-----------------+
! character 1     !
+-----------------+
! character 2     !
.                 .
.                 .
! character n     !
+-----------------+
```

These binary files are written by the WRITE FILE statement, and read by the READ FILE statement. A GET$ function call may also access this type of file, but treats it only as unstructured bytes of data.

TEXT (ASCII) INPUT / OUTPUT FILES

In this method of organization, integer, real, and string types are represented as one record followed by a delimiter. COMAL uses a carriage return and linefeed as its delimiter.

These text files are written by the PRINT FILE statement, as well as by the LIST, and DISPLAY. Text type files are also generated by output after a SELECT OUTPUT statement to a disk or tape file. These files are read by the INPUT FILE statement. A GET$ function call may also access this type of file, but treats it only as unstructured bytes of data.

If the data screen is opened as a file, it is considered a text input/output file. This also applies to the keyboard.

A PRACTICAL EXAMPLE

For this example, we will assume that you have a collection of data containing a man's last name, his age, and his average yearly income. Now, you would like to create a random file to store the data you have collected. The reason a random file is advised instead of a sequential file is that each person then has his own record. A random file is also much faster in data retrieval.

Each person's record is similar to a set of index cards containing information about the person. You can pull out any card at any time and read the information on it.

The first thing we must do to use a random file, is to plan the layout of the file. You need to specify how many people you wish to include in the file. Each person is assigned one record. The number of people in the file can be expanded in the future, if needed, but it is much faster to specify the maximum number from the start. For this example, we will specify 100 as our expected maximum number of records.

Next, we must plan the layout of a record. This is a very important step. Once a layout is used, it cannot be expanded, so plan ahead! In this example we need to include three items in each person's record: his name, age, and income. Each of

these items are often referred to as a field in the record. Now we must design a layout that will hold this information.

The best way to design the layout of the fields in a record is by actually drawing a representation of them on paper. This is the normal method used by professional programmers. Before we can proceed, we must look closely at the types of data we will be storing in each field. A persons name can be stored as a string. We must decide on a maximum length for this string. For our example, we will choose a length of 20. The persons age is a number than can be represented as either a real number or as an integer. We will use an integer field. Finally, his income will be represented as a real floating point number.

So our field specifications for a record so far are:

```
NAME, string, 20 characters maximum
AGE, integer
INCOME, real number
```

The order they are placed in the record is very important, since you can't change it later without recreating the whole file. For our example, we will use the order: NAME, AGE, INCOME.

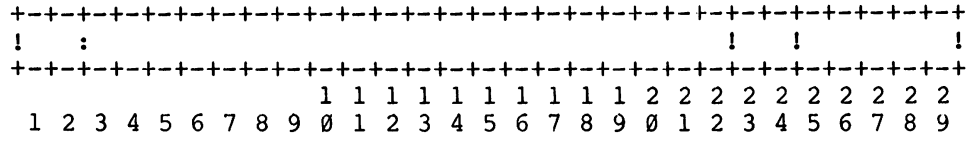
Next we must calculate the actual length of each field in CBM COMAL. To do this we must decide whether to use binary files created by WRITE FILE statements, or ASCII files, created by PRINT FILE statements. We will use binary files for our example, since they are more efficient, faster, and less understood.

Here are the calculations for our maximum record size.

The name will be a maximum of 20 characters, and we must also allow for the 2 byte length counter preceding the actual string. Thus the length allowed for the name is 22. The age is an integer, and all integers are always represented as 2 bytes. The income is a real number and all real numbers are always represented as 5 bytes. Thus our list of specifications now is:

```
NAME, string, 22 bytes
AGE, integer, 2 bytes
INCOME, real, 5 bytes
-----------------------
TOTAL bytes required: 29
```

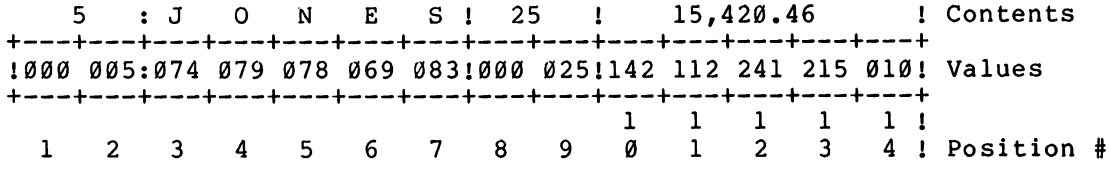
A picture of one record containing these 3 fields would be:



bytes 1-2 are the length counter for the following string
bytes 3-22 are the NAME string (can vary in length up to position 22)
bytes 23-24 are the AGE integer (actual positions depend upon the
bytes 25-29 are the INCOME real number (length of preceding string)

This is a drawing of a maximum size record. Since each name may vary, the actual record size can vary, but the space allocated to each record will always be the maximum.

Use this data for an example record: JONES, age 25, income 15,420.46. This data when stored in a record would look like this:



bytes 1-2 are the length of the string "JONES": 5
bytes 3-7 is the name JONES, represented by ASCII
bytes 8-9 are his age 25, represented in binary
bytes 10-14 are his income 15,420.46, represented in binary

Now we are ready to create the file and write the data into it. For this example, we will name the file "EXAMPLE.DAT".

Version 2.00 uses the following statement to create the file:

```
CREATE "EXAMPLE.DAT",101,29
```

Version 0.14 should use the procedure named CREATE as listed in Appendix D and call it with:

```
EXEC CREATE("EXAMPLE.DAT",101,29)
```

Both will create a random file with 101 records of length 29. One extra record was created because the first record will be used as a "counter" to indicate what the last record number is. This is necessary, because if we attempt to read a record that was not yet written, we will get an error.

Next, the program below can be used to enter the data:

```
DIM name$ OF 20  // this program for version 2.00
record#:=1 // initialize at 1 instead of 0 to save the first record
OPEN FILE 2,"EXAMPLE.DAT",RANDOM 29
LOOP
  INPUT "NAME (0 TO STOP): ": name$
  EXIT WHEN name$="0"
  record#:+1 // increment record counter
  INPUT "AGE: ": age#
  INPUT "INCOME: ": income
  WRITE FILE 2,record#: name$,age#,income
  PRINT // blank line
ENDLOOP
WRITE FILE 2,1: record# // last record number in file
CLOSE
PRINT "ALL DONE"

DIM name$ OF 20  // this program for version 0.14
record#:=1 // initialize at 1 instead of 0 to save the first record
OPEN FILE 2,"EXAMPLE.DAT",RANDOM 29
REPEAT
  INPUT "NAME (0 TO STOP): ": name$
  IF name$<>"0" THEN
    record#:+1 // increment record counter
    INPUT "AGE: ": age#
    INPUT "INCOME: ": income
    WRITE FILE 2,record#: name$,age#,income
    PRINT // blank line
  ENDIF
UNTIL name$="0"
WRITE FILE 2,1: record# // last record number in file
CLOSE
PRINT "ALL DONE"

RUN
  (Random file EXAMPLE.DAT is opened)
NAME (0 TO STOP): JONES
AGE: 25
INCOME: 15420.46
  (writes the data to record number 2)
```

```
NAME (Ø TO STOP): SMITH
AGE: 43
INCOME: 34129.Ø5
  (writes the data to record number 3)

NAME (Ø TO STOP): ANDERSON
AGE: 19
INCOME: 12ØØ3.ØØ
  (writes the data to record number 4)

NAME (Ø TO STOP): Ø
  (writes the number 4 to record number 1)
  (closes file 2)
ALL DONE
```

Now a program to print what is stored in your file so far:

```
DIM name$ OF 2Ø
OPEN FILE 3,"EXAMPLE.DAT",RANDOM 29
READ FILE 3,1: record# // read number indicating last record
FOR record'num:=2 TO record# DO
  READ FILE 3,record'num: name$,age#,income
  PRINT name$;age#;income
ENDFOR record'num
CLOSE

RUN
  (Random file EXAMPLE.DAT is opened)
  (A 4 is read as the last record number)
JONES 25 15420.46
SMITH 43 34129.Ø5
ANDERSON 19 12ØØ3
```

APPENDIX M
HEXADECIMAL AND BINARY NUMBERS
(version 2.00 Only)

A hexadecimal or binary number may be used whenever an integer accepted. To indicate that the number is in hexadecimal notation, it is preceded by a $. For example the decimal number 255 is represented as $ff. To indicate that the number is in binary notation, it is preceded by a %. For example 255 is represented as %11111111. Hexidecimal and binary numbers are always non-negative numbers ranging from decimal 0 to 65535 (hex $00 to $ffff, binary %00000000 to %1111111111111111). The six letters used by the hexidecimal system (abcdef) may be either upper or lower case, however, when listed they will always be in lower case except in graphics mode, where it is upper case (i.e., always listed as unshifted).

Hexidecimal or binary constants may be included in a COMAL program. Hex will always be listed with either two or four digits, preceded by the $. Binary will always be listed with either 8 or 16 digits, preceded by the %. For example, the following lines could be used:

```
10 DATA $0,%110,$ab3,$ffff,$000e
20 hex'number=$e000
30 READ a,b,c,d,e
40 INPUT "number please:":number
50 print hex'number;a;b;c;d;e;number

LIST
0010 DATA $00,%00000110,$0ab3,$ffff,$0e
0020 hex'number:=$e000
0030 READ a,b,c,d,e
0040 INPUT "number please:": number
0050 PRINT hex'number;a;b;c;d;e;number

RUN
number please:$033a
57344 0 6 2739 65535 14 826
```

Hexadecimal or binary numbers are not output by COMAL, since COMAL automatically performs a conversion to binary, as it also does with decimal numbers. All numeric output is in decimal. Thus one hexadecimal or binary number may be added to another and printed, but the output will be a decimal number:

```
PRINT $ff+$a4          or          PRINT $ff-%11111111
419                                Ø
```

COMAL will also allow hexidecimal or binary number strings with the VAL function:

```
PRINT VAL("$ff")       or          PRINT VAL("%1Ø1")
255                                5
```

COMAL can even accept hexidecimal or binary numbers from sequential or random file input. For example, the program below will write three hex numbers to a file and then read them as input:

```
DIM text$ of 3Ø
OPEN FILE 3,"SAMPLE.HEX",WRITE
PRINT FILE 3: "$AFE"+CHR$(13)+"$D1D"+CHR$(13)+"$AC34"
CLOSE
OPEN FILE 5,"SAMPLE.HEX",READ
WHILE NOT EOF(5) DO
  INPUT FILE 5: TEXT$
  PRINT TEXT$;
ENDWHILE
PRINT
PRINT "NOTE THAT THE NUMBERS ARE IN HEX"
CLOSE
OPEN FILE 2,"SAMPLE.HEX",READ
INPUT FILE 2: A,B,C
PRINT A;B;C
CLOSE

RUN
$AFE $D1D $AC34
NOTE THAT THE NUMBERS ARE IN HEX
2814 3357 44Ø84
```

If you would like to print numbers in hexadecimal, you can use the following function:

```
FUNC hex$(n) CLOSED
  DIM hexdigit$ OF 16, res$ OF 4
  hexdigit$:="0123456789abcdef"
  hi:=n DIV 256; lo:=n MOD 256
  res$:=hexdigit$(hi DIV 16+1)
  res$:+hexdigit$(hi MOD 16+1)
  res$:+hexdigit$(lo DIV 16+1)
  res$:+hexdigit$(lo MOD 16+1)
  RETURN "$"+res$
ENDFUN hex$

SCAN
PRINT "decimal";123;"is the same as";hex$(123);"hex"
decimal 123 is the same as $007b hex
```

If you would like to print numbers in binary, you can use the following function:

```
FUNC bin$(n) CLOSED
  DIM res$ of 8
  res$:="00000000"
  bit:=1
  FOR i#:=8 TO 1 STEP -1 DO
    IF n BITAND bit THEN res$(i#):="1"
    bit:+bit
  ENDFOR i#
  RETURN res$
ENDFUNC bin$
```

APPENDIX N
FILE NAMES

Filenames have changed in version 2.00 to be more consistent and compatible with other COMAL interpreters/run-time compilers. Thus this appendix will explain both the new, improved file name system as well as the old system used by version 0.14. Version 2.00 file names will be covered first since they follow the correct conventions.

Version 2.00 File Names

General Syntax:

```
[<unit>:][<name>]{/<attribute>}

 <unit> specifies a device, unit number, or disk drive number:
  accepted devices are: kb          keyboard
                        cs          cassette
                        ds          data screen
                        lp          line printer
                        sp          serial port (RS-232)
                        u<unitno>
                        [@]<drive>
    <unitno> is any device number from Ø-31
@ is optional and only used with disk file names. When included
    the file will be overwritten (updated) if it already exists
 accepted disk drive numbers are: <drive>
    <drive> is any number from Ø-15
     drives on disk unit 8 are Ø and 1
     drives on disk unit 9 are 2 and 3
      through
     drives on disk unit 15 are 14 and 15
  if omitted the default unit is the current UNIT$
   (the current unit is specified by the UNIT$ statement)
  note that a colon (:) is required after any <unit> if used
<name> is a string of characters up to 16 characters long
  (characters should not include a comma, colon, ?, $, or *)
<attribute> is a characteristic of the file
  (characteristics include: a, l, t, s, and d)
```

```
A                ASCII conversion from PET ASCII to ASCII letters
  a-     OFF - do not perform conversion
  a+     ON  - convert upper/lower case characters to ASCII

L                Line feed with each carriage return
  l-     OFF - suppress line feed upon carriage return
  l+     ON  - carriage return also sends line feed

T                COMAL Time Out System
  t-     OFF - use IEEE time out conventions used by CBM BASIC
                 (commonly used with instrument equipment)
  t+     ON  - use IEEE time out conventions used by CBM COMAL
                 (commonly used with CBM disk drives and printers)

S                Secondary address
  s-     OFF - do not use any secondary address
  s+     ON  - use secondary address selected by system
  s<number>    use secondary address specified by <number>
                 <number> is an integer from Ø - 15

D                Disk file
  d-     OFF - the file is not a disk file
  d+     ON  - the file is a disk file
                 (the disk error channel is automatically read
                   when errors occur)
```

NOTE: a / is required preceding each attribute specified.

Default attributes vary with the unit specified as follows:

```
<unit>     <dev> DEFAULT ATTRIBUTES   MODES
kb:          Ø   /a-/l-/t-/s-/d-      READ
cs:          1   /a-/l-/t-/s+/d-      READ,WRITE,APPEND
sp:          2   /a+/l+/t-/s+/d-      READ,WRITE,APPEND
ds:          3   /a-/l-/t-/s-/d-      READ,WRITE,APPEND
lp:          4   /a-/l+/t+/s-/d-      WRITE,APPEND
<drive>:     *   /a-/l-/t+/s+/d+      READ,WRITE,APPEND
u<unitno>: **    /a-/l-/t-/s-/d-      READ,WRITE,APPEND

* means device is 8 + (drive DIV 2)
** means device is specified <unitno>
```

NOTES ON ATTRIBUTES

Timeouts: when /t- is used, if a device has not accepted an I/O operation within 64 milliseconds, then just continue (the computer is not hung up and program execution is not interrupted). When /t+ is used, an error is reported if a device has not accepted an I/O operation within 64 milliseconds. If the device is not a disk drive (or has a /d- attribute), then the error reported will be a READ or WRITE error depending on the operation used. If the device is a disk drive (or has a /d+ attribute) then the disk error channel (secondary address 15) is read and the status number plus 200 is reported to the system.

Secondary Addresses: Depending upon the device, the following are applicable:

```
cs:     Cassette
    Ø = READ file (default READ)
    1 = WRITE file with End Of File mark on close (default WRITE)
    2 = WRITE file with End Of Tape mark on close

lp:     Line printer - only applicable to Commodore models
    Ø = Print data exactly as received (default)
    1 = Print data using previously defined format
    2 = Store formatting information
    3 = Set the number of lines per page
    4 = Enable print diagnostic messages
    5 = Define a programmable character
    6 = Set spacing between lines (not model 2Ø23)
    7 = Set to lower case printing mode (with upgrade ROM)

sp:     Serial port
    1 = Open for READ
    2 = Open for WRITE
    3 = Open for READ and WRITE

Disk drives:
    Ø = Read PRG file (for LOADing programs)
    1 = Write PRG file (for SAVEing programs)
    2-14 = Normal file access (only 2-13 on some models)
    15 = Command channel (with WRITE)
         Error channel (with READ)

ds:     Data screen
    no secondary addresses used

kb:     Keyboard
    no secondary addresses used
```

Disk files: When /d+ is used, additional characters are added to the file name automatically by the system (a user may append these characters to the file name if they wish):

```
READ file:          ",r"
WRITE file:         ",w"
APPEND file:        ",a"
RANDOM file:        ",1,"+chr$(<record length>)
                     <record length> is an integer from 1-254
READ/WRITE file:    ""
```

In addition, ⟨type⟩ characters as specified under ⟨type⟩ (see section below on Disk File Notes) are added in the following way:

```
PRG or P:           ",p"
SEQ or S:           ",s"
```

Attributes may be specified in any order. If an attribute is specified more than once, only the last one will be effective. For example: "lp:/a+/a-" is the same as "lp:/a-".

NOTES ON SERIAL PORT (Version 2.00 only)

The file name used when opening a file for the serial port includes the following information used by the system with that file (characteristics of the serial port — these attributes are in accordance with the specifications of the 6551 chip):

The file name used when opening a file for the serial port includes the following information used by the system with that file (characteristics of the serial port - these attributes are in accordance with the specifications of the 6551 chip):

```
 b<baud>    the baud rate
             <baud> can be 50-19200
             default is b300
 d<num>     the number of data bits
             <num> can be 5-8
             default is d7
 s<num>     the number of stop bits
             <num> can be 0-2
             default is s2
 p<type>    the type of parity
             <type> can be:
              n    none
              e    even
              o    odd
             default is pn
```

```
  Examples:
   "sp:" means 300 baud, 7 data bits, 2 stop bits, no parity
   "sp:b2400" means 2400 baud, 7 data bits, 2 stop bits, no parity
   "sp:b1200d8slpe" means
                    1200 baud, 8 data bits, 1 stop bit even parity
```

NOTES ON TAPE AND DISK FILE NAMES

A disk file name follows the same rules specified by Commodore BASIC (a tape file name is the same as disk file name, but without the ⟨type⟩ specifier). It is a ⟨string expression⟩ represented by the following:

```
<name> is represented by <file name>[,<type>]
 <file name> is up to 16 ASCII characters.
   when used in an OPEN statement, the file type is assumed to be
   sequential (SEQ). To open a file of another type, include the
   <type> as specified below:
 <type> specifies what type of file:
   SEQ or S means sequential file
   PRG or P means program file
   USR or U means user file
   when omitted, SEQ is used
   note that a comma must precede any <type> if used
```

For example, to open a program file (PRG), use a file name like:

```
"MY'PROGRAM,PRG"
```

The ⟨drive⟩ is optional and may be included immediately preceding the disk file name.

DISK FILE NAME EXAMPLES	TAPE FILE NAME EXAMPLES
"0:TEMP'FILE"	"TEMP'FILE"
"@0:SAMPLE"	"SAMPLE"
"FILE'1.L"	"FILE'1.L"
"A123 321B"	"A123 321B"
"PROGNAME,PRG"	

The ⟨file name⟩ may be a string constant enclosed in quotes as shown above, or it may be assigned to a ⟨string variable⟩ or ⟨string array⟩ element, which can then be used in place of the ⟨file name⟩ shown in the SYNTAX (but not with LOAD, SAVE, LIST, ENTER, or MERGE). Below, the variable INFILE$ is used in place of the ⟨file name⟩:

```
INFILE$:="0:CUSTOMER'FILE"
OPEN FILE 2,INFILE$,READ
```

When retrieving or deleting a file, CBM COMAL allows the same wildcard specifications as Commodore BASIC:

```
?       this character can be matched by any ASCII character.
*       the rest of the file name need not be compared.
```

For further information on file names, refer to your Commodore tape drive, disk drive, and computer manuals.

SUGGESTED NAMING CONVENTION

To avoid confusion when viewing a disk directory, the following file naming convention is recommended. A program SAVEd to disk should be a simple file name. Special files should add two or four characters to the end of the name for identification purposes: a period followed by one or three ID letters. The following letters are suggested:

```
<name>.L or .LST     program or segment LISTed to disk
<name>.D or .DSP     program or segment DISPLAYed to disk
<name>.O or .OBJ     object code from assembled file
<name>.S or .SRC     source code for assembler file
<name>.P or .PKG     package
<name>.E or .EXT     external procedure or function
<name>.B or .BAS     a program written in BASIC (not COMAL)
<name>.W or .TXT     a wordprocessor text file
<name>.DAT           a data file
```

EXAMPLES OF FILE NAME USE

```
SELECT OUTPUT "lp:"            PET ASCII output to printer
SELECT OUTPUT "lp:/a+"         ASCII output to printer
SELECT OUTPUT "lp:/t-"         Use with CBM 8024 printer
SELECT OUTPUT "logfile"        Send output to file named "logfile"

OPEN FILE printfile,"lp:66/s3",WRITE   set 66 lines per page (CBM)
OPEN FILE 2,"u8:$0/s0/t+/d+",READ      READ formatted directory
OPEN FILE 3,"u8:#10/s3/t+/d+"          Allocate floppy disk buffer
OPEN FILE 4,"kb:",READ         Open keyboard as a file
OPEN FILE 5,"ds:"              Open data screen for READ/WRITE

DIM wattmeter$ of 3
wattmeter$:="u9:"; w=6
OPEN FILE w,wattmeter$,READ    Open wattmeter instrument as a file

SAVE "cs:myfile/s2"            write program to tape with EOT mark
SAVE "0:copy/all"              /all is not a valid attribute and
                                thus may be used as part of filename

MERGE "routine.1"              MERGE program segment from disk

ENTER "prog.1"                 ENTER PET ASCII file from disk
ENTER "prog.1/a+"              ENTER ASCII file from disk
ENTER "sp:"                    ENTER ASCII file from another computer
                                via the serial port

LINK "graphics.obj"            LINK machine language package

LIST "lp:/a+/l-"               LIST without linefeed to ASCII printer

DISPLAY "lp:/a+"               DISPLAY with linefeed to ASCII printer

UNIT$ "cs:"                    Set cassette as default unit

MOUNT "1:"                     Initialize disk drive 1

PROC terminal(phone$) EXTERNAL "1:terminal.e"
```

EXAMPLES OF HOW COMAL OPENS DISK FILES BY DEFAULT

```
MERGE/ENTER/LINK                    READ, file type=SEQ
LIST/DISPLAY/SELECT OUTPUT          WRITE,file type=SEQ
SAVE                                WRITE,file type=PRG
LOAD/VERIFY/RUN/CHAIN/EXTERNAL      READ, file type=PRG
OPEN FILE <#>,<name>,READ           READ, file type=SEQ
OPEN FILE <#>,<name>,WRITE          WRITE,file type=SEQ
OPEN FILE <#>,<name>,RANDOM <rl>    RANDOM,no file type
CREATE                              RANDOM,no file type
```

VERSION 0.14 NOTES

In this introductory version, the file name was restricted to either a tape or disk file name. Thus only the notes in the above information concerning disk or tape will be applicable.

Also, SELECT OUTPUT statements will accept only two cases: "lp:" or "ds:".

The drive numbers may only be 0 or 1. Alternate disk units may be used by specifying a UNIT number in an OPEN statement (ie, UNIT 9).

Secondary addresses are specified differently. The secondary address is included after the unit number in the OPEN statement (ie, UNIT X,S where X is the unit number and S is the secondary address).

EXAMPLE (Select a Commodore Printer in lower case mode.)

APPENDIX O
COMAL DISK COMMAND

VERSION 0.14 and 2.00	Version 2.00 only

FORMAT A NEW DISK (disk called working'disk, id=w1 on drive 0)

```
pass "n0:working'disk,w1"          pass "n0:working'disk,w1"
```

REFORMAT A NEW DISK (format with same id as before on drive 0)

```
pass "n0:working'disk"             pass "n0:working'disk"
```

INITIALIZE A DISK (initialize drive 0)

```
pass "i0"                          mount "0:"
```

DUPLICATE A DISK (duplicate drive 0 to drive 1)

```
pass "d1=0"                        pass "d1=0"
```

COPY A DISK (copy drive 0 to drive 1)

```
pass "c1:*=0:*"                    copy "0:*","1:*"
```

COPY A DISK FILE (copy "sample" from drive 0 to drive 1)

```
pass "c1:*=0:sample"               copy "0:sample","1:*"
```

RENAME A DISK FILE (rename "temp" to "final" on drive 1)

```
pass "r1:final=temp"               rename "1:temp","final"
```

SCRATCH A DISK FILE (scratch drive 0 files starting with "temp")

```
pass "s0:temp*"                    delete "0:temp*"
```

(scratch all sequential files on drive 0)

```
pass "s0:*=seq"                    delete "0:*=seq"
```

VALIDATE (COLLECT) A DISK (validate drive 0)

```
pass "v0"                          pass "v0"
```

APPENDIX P
CREATING AND USING MACHINE LANGUAGE WITH COMAL

(1) Make sure you have a copy of COMSYMB on disk and set up for your specific COMAL implementation. This involves changing only the TRUE and FALSE assignments at the very beginning of the file to be TRUE for your system and FALSE for the others (only one may be TRUE — the other two MUST be FALSE).

```
CD8096=FALSE
ZRAM=FALSE
COMBI=TRUE     ; <— this is your system (here, the COMBI board)
```

The rest of this file contains the definitions of the over 540 symbols used by the COMAL system (the COMAL memory map). Using these symbols makes writing your assembly language program much simpler and compatible with other assembly language programs. Also, this allows your program to be compatible with future CBM COMAL implementations, since they may change specific locations, but these symbols will remain unchanged.

(2) LOAD and RUN the COMMODORE ASSEMBLER EDITOR

This editor is available from Commodore.

(3) Begin your assembler source code with the following:

```
1000 .lib comsymb
1010 ; comsymb is the name of the file holding the symbols
1015 *=$ 7000 ; this is the start address of machine code
```

(4) Write the rest of the code. For example, to reverse the screen see the sample assembler program listing on the next page.

(5) Save the code on disk:

```
PUT "0:LOR.SRC"
```

(6) Exit the editor:

```
KILL
```

(7) Run the assembler:

```
LOAD "asm65x-iv"
RUN
```

(8) Reply to the prompts:

```
Object file (cr or d:name): 0:LOR.OBJ
Hard copy (cr / y or n)? n
Cross reference (cr / no or y)? no
Source file name? 0:LOR.SRC
```

(9) Your object file is now assembled.
(10) Go back to the COMAL system.
(11) LINK the object code to your COMAL program. See Appendix K for the procedure on doing this.

APPENDIX Q
COMAL KEYWORDS GROUPED BY CATEGORY
(some keywords may appear in more than one category if applicable)

File Open/Close:

APPEND Type of disk file
CLOSE Closes a file
CREATE Creates a random file of a specific size
EXTERNAL Specifies the file name for an external procedure
FILE Special keyword used with file statements
MOUNT Initializes the disk in a specific drive
OPEN Opens a file
PASS Passes a disk command to the disk drive
RANDOM Type of disk file
READ Type of disk file
SELECT OUTPUT May open a disk file as the output location
UNIT$ The default unit to be used
WRITE Type of disk file

Input/Output/Data:

AT Specificies a screen position for INPUT or PRINT
DATA Allows DATA to be included with a program
EOD End Of Data flag
EOF End Of File flag
FILE Special keyword used with File Access statements
GET$ Gets a specified number of characters from a file
INPUT Inputs data from a file or the keyboard
KEY$ Gets a character from the keyboard
LABEL: Marks a line for use with RESTORE
LET Assigns a value to a variable
PRINT Prints data to a file
READ Reads data from a file or data statements
RESTORE Restores the next data item pointer
SELECT OUTPUT Selects where the system output will go
TAB Issues spaces until the tab position is reached

USING Allows formatted data to print
WRITE Writes data to a file
ZONE Establishes zones or tab points

Control Structures:

CASE STRUCTURE:

CASE Beginning of a CASE
ENDCASE End of a CASE
OF Special word at end of first CASE line
OTHERWISE Marks start of the default CASE section
WHEN Marks start of a specific CASE section

ERROR HANDLER STRUCTURE:

ENDTRAP End of the error handler
ERR The error number
ERRFILE The file number in use when the error occured
ERRTEXT$ The error message
HANDLER Marks start of the error handling section
REPORT Used to report an error to HANDLER or outer TRAP
TRAP Beginning of the trapped statements.

FOR STRUCTURE:

DO Special word used with a FOR statement
ENDFOR End of the FOR loop
FOR Start of the FOR loop
STEP Amount to increment the control variable
TO Separates the start value and end value

FUNCTION STRUCTURE

CLOSED Specifies the function variables to be local
ENDFUNC End of the function
EXTERNAL Specifies that the function is in an external file
FUNC Start of function
IMPORT Allows variables to be global within a closed function

REF Specifies that the parameter variable is INPUT/OUTPUT
RETURN Returns the function value

IF STRUCTURE:

ELIF Sets another condition
ELSE Marks the start of statements to use as default
ENDIF End of IF
IF Start of IF condition
THEN Special word used with IF and ELIF

LOOP STRUCTURE:

ENDLOOP End of LOOP
EXIT Exit from LOOP
EXIT WHEN Conditional EXIT from LOOP
LOOP Start of LOOP

PROCEDURE STRUCTURE:

CLOSED Specifies that the variables in the procedure are local
ENDPROC End of procedure
EXEC Execute a procedure
EXTERNAL Specifies that the procedure is in an external file
IMPORT Allows variables to be global within a closed procedure
INTERRUPT A special interrupt driven procedure execute statement
PROC Start of procedure
REF Specifies that the paramenter value is INPUT/OUTPUT
RETURN Return to calling EXEC statement / early end of procedure

REPEAT STRUCTURE:

REPEAT Start of REPEAT loop
UNTIL End of REPEAT loop

WHILE STRUCTURE:

DO Special word used with WHILE
ENDWHILE End of WHILE loop
WHILE Start of WHILE loop

Functions:

ABS Absolute value
ATN Arctangent
CHR$ Character
COS Cosine
EOD End Of Data flag
EOF End Of File flag
ERR Error number
ERRFILE File in use when error occurred
ERRTEXT$ Error message
ESC Stop key depressed flag
EXP Exponent
FALSE 0
GET$ Gets characters from a file
INT Integer
KEY$ Gets a character from keyboard
LEN Length of string
LOG Logarithm
ORD Ordinal value of character (ASCII number)
PEEK Value stored in specified memory location
RND Random number
SGN Sign of number
SIN Sine
SIZE Size of program and free workspace
SPC$ Space character
SQR Square root
STATUS Status of disk
STR$ String from a number
TAN Tangent
TIME Time in jiffies
TRUE 1
UNIT$ Current unit
VAL Numeric value of a string
ZONE Zone spacing

Operators:

AND True if both are true
AND THEN True if both are true - Special rules
BITAND Bitwise AND
BITOR Bitwise OR
BITXOR Bitwise XOR
DIV Integer answer to division
IN Sting search
MOD Modulo - remainder in division
NOT Opposite of TRUE or FALSE condition
OR True if either are true
OR ELSE True if either are true - Special rules

Print Format:

AT Specifies the screen location to start printing at
CURSOR Specifies the screen location to place the cursor
LINEFEED Specifies whether a carriage return includes a linefeed
PAGE Clear the screen or page feed the printer
PRINT Start of print statement
SELECT OUTPUT Select the location where the output is sent
TAB Specify the next print position on a line
USING Specify formating information
ZONE Specify the spacing of zones or system tabs

Disk Commands:

CAT Directory of disk (catalog)
CHAIN Load and run a program from tape or disk
COPY Copy a disk file
CREATE Create a random disk file of specified size
DELETE Delete disk files
DIR Directory of disk
ENTER Load/Merge an ASCII file from tape or disk
LINK Link an Object File onto a COMAL program
LIST STORE a program or segment in ASCII format
LOAD Load a program from tape or disk
MERGE Merge an ASCII format program segment from tape or disk
RENAME Rename a disk file

RUN Load and run a program from tape or disk
SAVE Save a program to tape or disk
SELECT OUTPUT Select a disk file as the output location
STATUS The disk status
VERIFY Verify a program saved to disk or tape

Editing Commands:

AUTO Automatic line numbering
DEL Delete lines
DISPLAY List the program without line numbers
EDIT List a program line without indentations
ENTER Enter a program segment from disk or tape
FIND Find all occurances of a specific string
LIST List a program or program segment
MAIN Return to main program from external section
MERGE Merge a program segment from tape or disk with current one
NEW Erase whole program from memory
RENUM Renumber the program lines
SCAN Scan program for structure errors
SETEXEC Specify whether or not EXEC will be listed
SETMSG Specify whether error messages will be text or numbers
SIZE Check on the size of the program and free workspace

Other Commands:

BASIC Return to BASIC
CON Continue program execution
DIM Dimension space for strings and arrays
DISCARD Discard all packages
RUN Run a program
USE Use a package

Other Statements:

END End of program
NULL Do nothing (no op)
POKE Put a value into a memory location
RANDOMIZE Seed the random number generator
STOP Stop program execution

SYS Jump to a specific address
TRAP ESC Disable / Enable the STOP key

APPENDIX R
SPECIAL KEYWORDS

KEYWORD: NEXT
CATEGORY: Statement
COMAL KERNAL: [YES] VERS 0.14 [*] VERS 2.00 [*]

Terminates a multiline FOR loop structure. NEXT is not used with a one-line FOR statement. The ⟨control variable⟩ used with the FOR must correspond with the one used with its matching NEXT. You may leave the ⟨control variable⟩ out however, and the COMAL interpreter will supply it for you. See Appendix A for a description of the FOR structure. NEXT is now converted to ENDFOR, to be consistent with the other loop structures terminators.

NOTE

To have version 2.00 list the keyword NEXT instead of ENDFOR, issue the following command: POKE $24b,PEEK($24b) BITOR %00000100.

SYNTAX

```
NEXT [<control variable>]

 <control variable> is a <numeric variable name>
   it matches the <control variable> in its matching FOR statement
   if omitted, it wil be supplied by the system
```

EXAMPLES

```
NEXT
NEXT temp
```

SAMPLE PROGRAM

```
FOR x=1 TO 3
  FOR y=3 TO 4
    PRINT x;"PLUS";y;"IS";x+y
  NEXT y              converts to ENDFOR y
NEXT x                converts to ENDFOR x
PRINT "ALL DONE"

RUN
1 PLUS 3 IS 4
1 PLUS 4 IS 5
2 PLUS 3 IS 5
2 PLUS 4 IS 6
3 PLUS 3 IS 6
3 PLUS 4 IS 7
ALL DONE
```

ADDITIONAL SAMPLES SEE: OR, ORD, PEEK, REM
USED IN PROCEDURES: DISK'GET'INIT, LOWER'TO'UPPER
SEE ALSO: DO, ENDFOR, FOR, STEP, TO

APPENDIX S
BASIC KEYWORDS IMPLEMENTED DIFFERENTLY IN COMAL

ASC — implemented as ORD
BACKUP — implemented via PASS and COPY
CLR — not needed in COMAL
CMD — implemented via SELECT OUTPUT
COLLECT — implemented via PASS
CONCAT — implemented via PASS
CONT — implemented as CON
DEF — implemented via FUNC
DLOAD — implemented as LOAD
DS — implemented via STATUS
DS$ — implemented via STATUS
DSAVE — implemented as SAVE
FN — implemented via FUNC
FRE — implemented as SIZE
GET — implemented as KEY$ and GET$
GOSUB — implemented as EXEC
HEADER — implemented via PASS
INPUT# — implemented as INPUT FILE
LEFT$ — implemented via better substring definition

MID$ — implemented via better substring definition

ON — implemented via CASE
POS — see POS procedure in APPENDIX D
PRINT# — implemented as PRINT FILE
REM — implemented as //
RIGHT$ — implemented via better substring definition

SCRATCH — implemented as DELETE
SPC — implemented as SPC$
ST — implemented via STATUS

TI — implemented as TIME
(also see function JIFFIES in APPENDIX D)
TI$ — implemented via TIME
USR — implemented via better machine language links via packages and LINK
WAIT — implemented via WHILE loop

INDEX

THE COMAL HANDBOOK 2/E
Len Lindsay

If you have a Commodore 64® computer—this is your complete introduction to COMAL!

At last—a book that explains this new structured form of BASIC. Invented in Denmark, this language is for use on the Commodore 64 and 6502 computers. An easy-to-follow reference book—complete with a cross-reference, offers you an excellent source of over 100 COMAL programs and procedures.

You'll appreciate these special features:

- SYNTAX is shown together with several examples
- SAMPLE PROGRAM is shown for each keyword with a sample RUN to show you how it is used in action
- All other related keywords are cross-referenced

The CBM COMAL interpreter (2.0 version), all the sample procedures from THE COMAL HANDBOOK, the error message file, and selected programs from the book are included in a 5¼ floppy disk available for $20.00 from Reston Publishing Company, Reston Computer Group, 11480 Sunset Hills Road, Reston, VA 22090 — or call (703) 437-8900.

ALSO FOR THE COMMODORE 64 COMPUTER . . .

Using the Commodore 64® Without Learning BASIC
Tim Kelly
This book is organized to help you operate your Commodore 64 with as much ease as you would operate any of your home appliances. Just as you can learn how to drive a car without learning all about the parts inside, you can use your Commodore 64 without learning BASIC. This exciting new book shows you how it's done!

Survival on Planet X With the Commodore 64® Computer
Michael Orkin and Ed Bogas
This clever adventure story provides an exciting way for you to learn the basics of programming. Some of the programs are simple, some are complicated—all provide an excellent introduction to computer programming on the Commodore 64.

Cover Design by Debbie Balboni

Reston Computer Group
RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

0-8359-0784-8

0 21898 07840 7