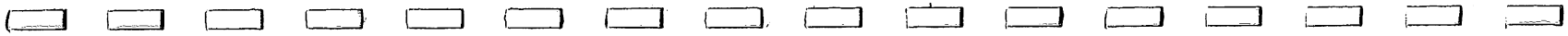
$18.95

COMAL COMAL COMAL COMAL HANDBOOK HANDBOOK HANDBOOK HANDBOOK

Len Lindsay



COMAL HANDBOOK

Len Lindsay

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

I dedicate this book
to my beautiful four-year-old daughter
RHIANON
and
to all other future programmers.

Library of Congress Cataloging in Publication Data
Lindsay, Len.
COMAL handbook.
1. COMAL (Computer program language) I. Title.
II. Title: COMAL handbook
QA76.73.C26L56 1982 001.64'24 82-12233
ISBN 0-8359-0878-X

PET and CBM are trademarks of Commodore.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

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 one year of testing and working with COMAL. In May, 1982, the manuscript was taken to Denmark, and reviewed in detail with the creators of CBM COMAL-80: Mogens Kjaer, Lars Laursen, and Jens Erik Jensen.

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, Dan Duckart, Tom Foth, Steve Kortendick, 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

"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.

Tonder, Denmark — Borge R. Christensen

INTRODUCTION

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 listed in **APPENDIX D** should give you a good start as you begin writing your own programs. Finally, the complete Standard COMAL Definition, referred to as the COMAL Kernal, is reproduced in Appendix H for those who wish to refer to it.

WHAT IS COMAL

COMAL-80 is a high level language that originated in DENMARK. This handbook is written specifically for CBM COMAL-80. However, since it follows the COMAL-80 definition quite accurately, other implementations should be very similar.

CBM COMAL-80, or just COMAL for short, is an interpreter as is CBM BASIC. COMAL retains the friendly environment of CBM BASIC, while it adds the power of structured programming, styled after PASCAL. Thus COMAL combines the BEST of BASIC with the BEST of PASCAL.

Many COMAL commands and statements may be executed immediately 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 line that the error message overwrote 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, convert all branching statements to the actual addresses, and then execute the program (this is known as a three pass interpreter, or semi-compiler).

Editing a program is similar to editing a BASIC program. One difference is that COMAL uses a delete command to delete lines, while BASIC uses a line number with no statement following it. 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.

The COMAL definition was revised and finalized during May and June of 1982. The new COMAL STANDARD is being referred to as the COMAL KERNAL. It included three major changes: substring specification, ZONE specification, and function definition.

There are several different versions of CBM COMAL, each written by INSTRUTEK. Introductory version 0.11 is the original COMAL that would run on any 32K CBM or PET with BASIC 4.0. It is now obsolete due to the changes in the COMAL definition. Introductory version 0.12 replaces it to meet those changes. If you only have obsolete version 0.11, you should try to get it upgraded to version 0.12. Both versions are in the public domain and may be copied freely. Version 1.02 completely meets the new COMAL STANDARD, plus it has many enhancements. It requires either 96K (CBM 8096) or the 64K COMAL ROM board from INSTRUTEK. This handbook is written to apply to all the above versions. VERSION 0.12 will be used to mean both versions 0.11 and 0.12. Where there are differences, they will be mentioned. Sample programs and procedures were tested with preliminary releases of version 0.12 and 1.02.

HOW TO USE THIS HANDBOOK

The HANDBOOK is structured to be useful, even to those without a computer running COMAL. Thus the sample programs are always followed by a typical sample run, showing how the program executes. 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

The COMAL KEYWORDS are presented in alphabetical order, for easy reference. Each KEYWORD has its own page (or more if needed) and 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.

SAMPLE KEYWORD PAGE

(1) KEYWORD
(2) CATEGORY: XXXXX
(3) COMAL STANDARD: [X] VERSION 0.12 [X] VERSION 1.02 [X]

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

(5) NOTES

XXXXX XXXXX XXXXX XXXXX XXXXX XXXXXXX XX XX XXXXXX XXXXXX XXXX
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 XXXXXXX
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, XXXXXXXXX

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

(1) The KEYWORD being presented on the page is identified on the top 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 standard or not, and in which versions of CBM COMAL it is implemented. The following notation is used:

COMAL STANDARD		VERSIONS	
YES	part of standard	-	not implemented
NO	an enhancement	+	partially implemented
		*	fully implemented

(a) COMAL Standard:
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 STANDARD, it should be the same in any other COMAL implementation that meets the COMAL KERNAL.

(b) Version 0.12 (update from version 0.11):
This is the introductory version of COMAL, placed in the public domain by COMMODORE. Version 0.11 (the original release) is now obsolete, due to changes in the COMAL definition. Version 0.12 is the update to meet those changes. **VERSION 0.12** will be used to mean either version. If there is a difference it will be mentioned. The four main differences are: substrings now specify a ⟨start⟩ and ⟨end⟩ character while the old method was to specify a ⟨start⟩ character and ⟨length⟩ (see **APPENDIX B** for more information on substring handling); functions follow the new COMAL definition using keywords FUNC and ENDFUNC with RETURN; ZONE is now a function and not a system variable; the keyword NEXT has been changed to ENDFOR. It is highly advisable to use version 0.12 in place of 0.11, which is no longer being supported. However, the *Handbook* is prepared to be applicable to both. Versions 0.12 and 0.11 are disk loaded. Version 0.11 included a SPLIT version in addition to the FULL COMAL interpreter. This SPLIT version provided the COMAL system in two parts to allow more memory available for the user:

(a) COMAL80IN - the input module.
(b) COMAL80EX - the execute module.

The FULL version provides the entire introductory COMAL system (missing some of the STANDARD COMAL and without the

enhancements added in version 1.02), both input and execute modules, and leaves about 6K free memory for the user.

(c) Version 1.02:
This is the enhanced COMAL, completely meeting the new COMAL definition. It is available in two forms, both written by INSTRUTEK and completely compatible:

(1) Disk loaded for the CBM 8096.

(2) Plug in board for any PET/CBM (except OLD ROM PET).
This board is available from:
INSTRUTEK
Christiansholmsgade
DK-8700 Horsens, DENMARK

A high resolution graphics board is also available from INSTRUTEK. It is compatible with the COMAL ROM board and includes excellent graphics system software. These COMAL compatible high resolution graphics are not possible with the CBM 8096; the COMAL ROM board is needed.

(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 DEFINITION (see APPENDIX H). The method includes the conventions explained below:

a) Items in UPPER CASE must be typed as shown, unshifted.

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 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 complete program, suitable for entering on your PET/CBM COMAL system. 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. Note that an upper case letter means enter that letter **UNSHIFTED**. 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. The statements will often show an assignment using "=" instead of ": =". This is because you may type just the "=" and the COMAL system will convert it to the assignment symbol, ": =", for you. The same is true for optional KEYWORDS. Often DO, THEN, OF, FILE, or OUTPUT may be left out because you need not type them. The COMAL system will supply them for you. Also, the sample program is 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 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)

(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 which include the KEYWORD presented on this page, see the other KEYWORDS listed here.

(12) USED IN PROCEDURES: - This KEYWORD is used in the sample procedures listed here. The procedure listings can be found

in **APPENDIX D**. A few functions are also included in APPENDIX D.

(13) SEE ALSO: - Other COMAL KEYWORDS that are related to this KEYWORD are listed here. Reading about them may give you further insight into this KEYWORD.

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:

⟨disk file name⟩
⟨disk or tape file name⟩
⟨variable name⟩
⟨identifier⟩
⟨expression⟩
⟨numeric expression⟩
⟨string expression⟩
⟨statements⟩
⟨unit⟩

Rather than fully explain these items each time they occur, they are explained below.

⟨disk file name⟩
and
⟨tape file name⟩

A file name follows the rules as specified by PET/CBM BASIC. It is a ⟨string expression⟩ represented by the following:

"[[@][⟨drive number⟩]:]⟨file name⟩" for a disk file name

"⟨file name⟩" for a tape file name

[@] is optional and when included
the file will be overwritten (updated) if it already exists.
⟨drive number⟩ is either a 0 or a 1,
it may be omitted when reading from disk.
when omitted, both drives are checked, beginning with the drive last accessed. When deleting a file or listing a program to a disk, it is not optional and must be supplied.
⟨file name⟩ is up to 16 ASCII characters.
when used in the OPEN statement, the file type is assumed to be

sequential (SEQ). To open a file of another type, follow the program name with a comma and the type. For example, to open a program file (PRG), use a file name like:

```
"0:MY'PROGRAM,PRG"
```

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"

The ⟨file name⟩ may be a string constant enclosed in quotes as shown above, or often 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. Below, the variable INFILE$ is used in place of the ⟨file name⟩:

```
INFILE$="CUSTOMER'FILE"
OPEN 1,INFILE$,READ
```

When retrieving or deleting a file, COMAL allows the same wildcard specifications as PET/CBM 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 PET/CBM computer, tape drive, and disk drive manuals.

⟨variable name⟩ and ⟨identifier⟩

An ⟨identifier⟩ may be up to 16 characters long in version 0.12 and up to 78 characters long in version 1.02. Each character is significant. These characters may be unshifted 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
real variable name	⟨identifier⟩	COUNT
string variable name	⟨identifier⟩$	NAME$
integer variable name	⟨identifier⟩#	SCORE#
label	⟨identifier⟩:	ERROR:
function name	⟨identifier⟩	EVEN
procedure name	⟨identifier⟩	SORT
integer procedure name	⟨identifier⟩#	COUNTER#
string function name	⟨indentifier⟩$	COMPLETE$
array element	⟨identifier⟩(...)	SCORE(X)
string array element	⟨identifier⟩$(...)	PLAYER$(Y)
integer array element	⟨identifier⟩#(...)	TRACK#(X,Y)

A specific ⟨identifier⟩ may be used only in 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. Thus if you have a procedure called **TEST** you may not also have a variable named **TEST, TEST#,** or **TEST$.** Some ⟨identifiers⟩ are reserved for use as a COMAL KEYWORD, and thus you must never have a variable named **NEXT** or **THEN,** etc.

⟨expression⟩

An expression is either a ⟨string expression⟩ or a ⟨numeric expression⟩.

⟨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
real variable	COUNT
integer variable	WINDOWS#
real array element	SCORE(4)
integer array element	WALLS#(X)
system function	ABS(X)
user defined function	EVEN X)
integer function	GCD#(X,Y)

IN operation	A$ IN VALID$
math operation	5 + 34
system constant	FALSE
boolean expression	NOT A=B

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 operation	DRIVE$ + NAME$

⟨statements⟩

This means that other COMAL statements may go here.

⟨unit⟩
or
⟨dev⟩

Unit is also known as a device number on the IEEE 488 bus. Commodore assigned units are listed in the chart below. Note that tape drive 2 is not supported by COMAL. In the future, it may be used for an RS-232 port.

UNIT	DESCRIPTION	UNIT	DESCRIPTION
0	keyboard	5	CBM 8010 modem
1	tape drive 1	6	user defined
2	tape drive 2 (not supported) future RS-232 port	7	user defined
		8	disk drive
3	screen	9	alternate disk drive
4	printer	10-30	user defined

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: `': '` 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'`.

SPECIAL NOTE ON FILE NUMBERS

COMAL supports the same file system as CBM BASIC. This means that when a file is opened it is given a number which is referenced in statements accessing the file. File numbers can range from 1-255. However, the COMAL system reserves the first and last file numbers for its own use. File 1 is used for system disk access, and file 255 is used for the printer. Thus a COMAL program is allowed to safely use file numbers 2-254.

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.

SPECIAL NOTE

This book was already typeset while final changes were being made to the COMAL KERNAL, the COMAL standard. The book was updated, even at the last minute to incorporate all changes, so as to present the final standard COMAL. There was one keyword change that will not be reflected in this book, since it is part of so many sample programs and procedures. The keyword NEXT was changed to ENDFOR to be compatible with all the other keywords ending a structure. But, the COMAL system will accept the word NEXT if you type it in, and will automatically convert it to ENDFOR for you. Thus all sample programs and procedures can be typed as presented, and the COMAL system will update the keyword NEXT to ENDFOR for you. Apparently the COMAL KERNAL will probably be updated to allow a FOR loop to end with an ENDFOR.

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 publisher's 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.

COMAL-80 is a high level language developed by Borge Christensen in Denmark. CBM COMAL-80 is a specific implementation of COMAL-80 written by Mogens Kjaer, Lars Laursen, and Jens Erik Jensen, of Instrutek in Denmark.

KEYWORD: ABS
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 non-negative number is not affected. A negative number is converted to its positive equivalent.

SYNTAX

```
ABS(⟨numeric expression⟩)
```

EXAMPLES

```
ABS(-3.2)
ABS(15-N)
ABS(INT(X))
ABS(HIGH - SCORE)
```

SAMPLE PROGRAM

```
REPEAT
   INPUT "NUMBER (0 TO STOP): " : N
   IF ABS(N)⟨⟩N THEN PRINT "THAT IS NEGATIVE"
   A=ABS(N)
   PRINT "ABSOLUTE VALUE OF";N;"IS";A
UNTIL N=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
```

SEE ALSO: INT, SGN

KEYWORD: AND
CATEGORY: OPERATOR
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Logical math operator evaluates to TRUE (a value of 1) only if the ⟨numeric expression⟩ on its left **AND** the ⟨numeric expression⟩ 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.

SYNTAX

```
⟨numeric expression⟩ AND ⟨numeric expression⟩
```

EXAMPLES

```
X$⟩="A" AND X$⟨="Z"
A=8 AND B=0
N=0 AND TEXT$⟨⟩"0"
```

SAMPLE PROGRAM

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

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$(1) = "TRUE"
ZONE 6
PRINT "AND CHART"
PRINT "———"
FOR A=FALSE TO TRUE
  FOR B=FALSE TO TRUE
    PRINT "A= "; TYPE$(A), "B= "; TYPE$(B),
    PRINT "A AND B= "; TYPE$(A AND B)
  NEXT B
NEXT 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 SAMPLE SEE: DIM (numeric arrays)
USED IN PROCEDURES: GET' ALPHA,LOWER'TO'UPPER, UPPER'TO'LOWER
SEE ALSO: NOT, OR

KEYWORD: APPEND
CATEGORY: Type of OPEN
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

SYNTAX

```
OPEN [FILE] ⟨number⟩,⟨filename⟩[,UNIT ⟨dev⟩[,⟨secndry adr⟩]],APPEND

 ⟨number⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨filename⟩ is a ⟨disk file name⟩
 ⟨dev⟩ is a ⟨numeric expression⟩ whose value is from 4-30;
    if omitted, the default value is 8 for a CBM disk drive
 ⟨secndry adr⟩ is a ⟨numeric expression⟩ whose value is from 0-15;
     if omitted, COMAL assigns the secondary address
```

EXAMPLES

```
OPEN FILE 2,"TEST",APPEND
OPEN 5+X,N$,APPEND
OPEN 5,"NAMES.DAT",UNIT 9,APPEND
```

SAMPLE PROGRAM

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

```
RUN
   (system opens a previously existing disk file to be referred
    to as file 2 named VISITORFILE and 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, UNIT, WRITE

KEYWORD: AT
CATEGORY: Special
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

Allows you to begin printing at any spot on the screen by specifying the row and column to start at. 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.

NOTE

PRINT USING and PRINT AT are not supported by version 0.12.

SYNTAX

```
PRINT AT ⟨row⟩,⟨col⟩: [⟨print list⟩] [⟨continue mark⟩]

 ⟨row⟩ is a ⟨numeric expression⟩ whose value is from 1-25
 ⟨col⟩ is a ⟨numeric expression⟩ whose value is from 1-80
 ⟨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 the USING image with 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.
 ⟨continue mark⟩ may be either a comma (,) or a semicolon (;)
   if omitted, a carriage return and line feed result
```

EXAMPLES

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

SAMPLE PROGRAM

```
REPEAT
  PRINT "[CLR]"
  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: "*"
  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
```

SEE ALSO: CURSOR, PRINT, TAB, USING, ZONE

KEYWORD: ATN
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Returns the arctangent of the specified number in radians. One radian equals approximately 57 degrees. **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(Z+Y-2)
ATN(10-INT(NUM))
```

SAMPLE PROGRAM

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

KEYWORD: AUTO
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 1.02, or hit ⟨return⟩ twice in version 0.12. **AUTO** will begin with line number 10, unless you specify another starting line. It will increment each line by 10, unless you specify a different interval.

NOTES

1) In version 1.02, 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) **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.
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, the starting line number will be 10
 ⟨increment value⟩ is a number from 1 - 9999;
    if omitted, the increment value will be 10
```

EXAMPLES

COMMAND	RESULTS
AUTO	Provides line number series: 10, 20, 30,...
AUTO 200	Provides line number series: 200, 210, 220,...
AUTO ,5	Provides line number series: 10, 15, 20,...
AUTO 400,2	Provides line number series: 400, 402, 404,...

SEE ALSO: DEL, EDIT, LIST, RENUM

KEYWORD: BASIC
CATEGORY: Command
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

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 PET/CBM computer with a cold start, returning you to the original BASIC operating system. To return to COMAL you must turn your computer off and back on if you are using the COMAL ROM BOARD, or you must reload COMAL from disk if you are using a disk loaded COMAL.

NOTES

1) **BASIC** is not supported by version 0.11.
2) To achieve the same results in version 0.11 use procedure BASIC' RESET in **APPENDIX D**, or issue the command:
   SYS 64790

SYNTAX

```
BASIC
```

EXAMPLE

```
BASIC
```

SAMPLE EXERCISE

```
BASIC
   (screen clears - computer is reset)
### commodore basic 4.0 ###          (this display will be the
                                     same as the one when you
31743 bytes free                     turn on the computer in BASIC
                                     mode)
```

SEE ALSO: NEW

KEYWORD: CASE
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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

KEYWORD: CAT
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

Gives a catalog (directory) of all disk files on the diskette currently in the disk drive specified by ⟨drive number⟩. If no specific drive is requested, the last accessed drive is used. 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" or SELECT "LP" prior to your **CAT** command (this selects the Line Printer).

NOTES

1) CAT is not supported by the execute module of version 0.11 SPLIT.
2) ⟨unit⟩ and ⟨pattern⟩ are not available with this command in version 0.12.
3) In version 1.02 hitting SPACE will pause the directory. Hit SPACE again to resume.

SYNTAX

```
CAT [⟨drive number⟩] [,⟨unit⟩] [ ⟨pattern⟩]

 ⟨drive number⟩is a ⟨numeric expression⟩ whose value is 1 or 0;
    if omitted, it defaults to both drives
 ⟨unit⟩is a ⟨numeric expression⟩ whose value is from 4-30;
    if omitted, the default value is 8 for a CBM disk drive
 ⟨pattern⟩ is a ⟨string expression⟩
    allows a selective catalog, using pattern matching
```

EXAMPLES

COMMAND	RESULT
CAT	directory of last accessed drive
CAT 0	directory of drive 0
CAT 1	directory of drive 1
CAT 0,9	directory of drive 0 on unit 9
CAT 1"P*"	directory of files beginning with P on drive 1
CAT 1"*=SEQ"	directory of sequential files on drive 1

SAMPLE EXERCISE

```
(to print the directory of drive 1 onto the printer)

SELECT "LP"          select the printer for output
CAT 1                drive 1 directory
 1"WORD PRO 4 AUTO " W1 2A
 3    "LOADWP"          PRG     the directory will
 72   "WP4"             PRG     print on your printer
 1    "PRINTCHARACTER" PRG      nothing nothing prints on the
 588 BLOCKS FREE.               screen
                                the directory will vary
                                depending upon the disk
                                currently in the drive

SELECT "DS"                     return output to the screen
```

ADDITIONAL SAMPLE SEE: SAVE

SEE ALSO: OPEN, OUTPUT, SELECT, UNIT

KEYWORD: CHAIN
CATEGORY: Command / Statement
COMAL STANDARD: [NO] VERSION 0.12 [+] VERSION 1.02[*]

LOADs and RUNs a program from disk or tape. The program must have previously been stored on disk or tape using either the SAVE command (or in version 0.11 SPLIT mode via RUN). 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 also is 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** is not available in version 0.11 SPLIT MODE as a direct command.
2) **CHAIN** and ENTER commands are not compatible.
3) **CHAIN** cannot be used to LOAD and RUN a program that was LISTed or EDITed to disk or tape.
4) The ⟨unit⟩ specification is not supported by version 0.12.

SYNTAX

```
CHAIN ⟨program name⟩[,⟨unit⟩]
  ⟨program name⟩ is a ⟨disk or tape file name⟩
  ⟨unit⟩ is a ⟨numeric expression⟩ whose value is 1 or 4-30;
    if omitted, the default value is 8 for a CBM disk drive
```

EXAMPLES

```
CHAIN "TEST"
CHAIN P$
CHAIN "NEXTONE",9              from unit 9
```

SAMPLE PROGRAM

```
DIM C$ OF 1
PRINT "E - EDIT DATA
PRINT "I - INPUT DATA
PRINT "P - PRINT REPORT
REPEAT
  INPUT "WHAT IS YOUR CHOICE: " : C$
  CASE C$ 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 now LOADed from disk and RUN. )
```

ADDITIONAL SAMPLE SEE: OBJLOAD

SEE ALSO: CON, LOAD, RUN, SAVE, UNIT

KEYWORD: CHR$
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Returns the character that has the specified numeric (ASCII) code. May be used to send a special character to a printer (ie. 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$(C+128)
```

SAMPLE PROGRAM

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

RUN
GYNJOFQKOGJSXSGKPRLWBAJFJNMKKT
```

SAMPLE PROGRAM

```
REPEAT
   INPUT "NUMBER (1-255 / 0 TO STOP) : " : N
   IF N THEN PRINT N; "IS REPRESENTED BY"; CHR$(N)
UNTIL N=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 will produce effects other than a printed character on the screen, i.e., try number 147 and your screen will clear

ADDITIONAL SAMPLES SEE: DO, GET$, IN, REF
USED IN PROCEDURES: BOLD'CHAR, DISK'GET'STRING, GET'CHAR,
SCAN, SET'PITCH
SEE ALSO: ORD, STR$, VAL

KEYWORD: CLOSE
CATEGORY: Command / Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Closes files currently open. If a ⟨file number⟩ is specified, only that file is closed. If no ⟨file number⟩ is specified, all files are closed. If you leave out the keyword FILE, it will be supplied by the COMAL system. No error is generated 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 CBM Disk Manual for information on these commands. Examples of how to use the commands from COMAL are included with the procedure DISK'COMMAND in **APPENDIX D** and with keyword PASS.

SYNTAX

```
CLOSE [[FILE] ⟨file number⟩]

 ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 2-254;
   if omitted, all files are closed
```

EXAMPLES

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

SAMPLE

```
CLOSE                    make sure all previous files are closed
DIM NAME$ OF 20, D$ OF 1
OUTFILE=3;NAME$="HAROLD";HIGH'SCORE=58
PRINT NAME$;"HAS THE HIGH SCORE OF";HIGH'SCORE
REPEAT
  INPUT "WHICH DRIVE (0 OR 1) SHALL WE WRITE THIS TO: " : D$
UNTIL D$ IN "01"
OPEN OUTFILE,"@" +D$ +":SCORE'FILE",WRITE
WRITE FILE OUTFILE : NAME$,HIGH'SCORE
CLOSE 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
USED IN PROCEDURE: DISK' COMMAND
SEE ALSO: APPEND, EOF, FILE, OPEN, PRINT, RANDOM, READ, STATUS, UNIT, WRITE

KEYWORD: CLOSED
CATEGORY: Type of procedure or function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 procedure variable. Version 1.02 also requires that any procedure or function called from within a **CLOSED** procedure or function be declared global with an IMPORT statement. Procedures and functions are always global in version 0.12. Within a **CLOSED** procedure, 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) Version 0.11 uses PROC for functions. Versions 0.12 and 1.02 use FUNC for functions.
3) For more information on the procedure structure see **APPENDIX A**.

SYNTAX

```
PROC ⟨procedure name⟩[(⟨parameter list⟩)] [CLOSED]
  plus versions 0.12 and 1.02 use FUNC for functions:
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
PROC QUICKSORT(LEFT,RIGHT,REF ITEMS$()) CLOSED
```

SAMPLE PROGRAM

```
TEST=1
PRINT "MAIN TEST IS";TEST
EXEC SAMPLE
PRINT "MAIN TEST STILL IS";TEST
END
//
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
SEE ALSO: ENDFUNC, ENDPROC, EXEC, FUNC, IMPORT, PROC, REF

KEYWORD: CON
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Continues (or resumes) program execution following a break caused by hitting the STOP key, a STOP or END statement, or an error. 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.

NOTE

CON is not supported by old version 0.11 SPLIT mode.

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

KEYWORD: COS
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Returns the cosine 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

```
COS(⟨numeric expression⟩)
```

EXAMPLES

```
COS(2)
COS(X*Y)
```

SAMPLE PROGRAM

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

SEE ALSO: ATN, SIN, TAN

KEYWORD: CURSOR
CATEGORY: Statement/Command
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

Positions cursor to the line and position 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.

NOTE

The **CURSOR** keyword is not supported by version 0.12. It may be simulated using the procedure **CURSOR** listed in **APPENDIX D**.

SYNTAX

```
CURSOR ⟨line⟩,⟨position⟩

 ⟨line⟩ is a ⟨numeric expression⟩ whose value is from 1-25
 ⟨position⟩ is a ⟨numeric expression⟩ whose value is from 1-80
```

EXAMPLES

```
CURSOR 9, 2
CURSOR ROW, COL
```

SAMPLE PROGRAM

```
DIM A$ OF 1
PRINT "[CLR]"                          clear the screen
PRINT "HIT ANY KEYS, HIT X TO EXIT"
REPEAT
  A$=KEY$                              see if a key is hit
  CURSOR 12,40                         40 column screens use 12,20
  PRINT A$
UNTIL A$="X"
PRINT "ALL DONE"
```

```
RUN
   (screen clears)
HIT ANY KEYS, HIT X TO EXIT
   (for every key you hit, its character 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: GET$, INPUT, PRINT, TAB

KEYWORD: DATA
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 can follow the keyword **DATA**, separated by commas, up to the limit of the program line length. Data statements are non-executable and may be placed anywhere within the program. 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). 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 semi-colons (;) 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.

NOTE

In version 1.02 data within a closed procedure or function is regarded as local data. See procedure DISK'GET'INIT in Appendix D for an example of local data.

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"
```

SAMPLE PROGRAM

```
DIM NAME$ OF 20
EXEC NAME'AGE
PRINT "ALL DONE"
//
PROC NAME'AGE
  RESTORE NAMES                                    not used in version 0.12
  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: RESTORE, SELECT, SGN
USED IN PROCEDURE: DISK'GET'INIT
SEE ALSO: EOD, LABEL, READ, RESTORE

KEYWORD: DEL
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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)))) **WARNING**: in version 0.11 **DEL** with no lines specified defaults to all lines and will delete the whole program.
2) To delete line number 10 you can not just enter a null line 10 (as in PET/CBM BASIC). You must use the **DEL** command:

```
DEL 10
```

SYNTAX

```
DEL [⟨start line number⟩] [-] [⟨end line number⟩]

 ⟨start line number⟩ is a number from 1 - 9999
 ⟨end line number⟩ is a number from 1 - 9999
   and is greater than ⟨start line number⟩ if 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	delete entire program (vers 0.11 only)

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

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

KEYWORD: DELETE
CATEGORY: Command / Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

Deletes a file from disk. Wildcard specifications may be used as with the PET/CBM SCRATCH command.

NOTES

1) Device number option is not supported by version 0.12.
2) The disk drive number must be specified in the file name.
3) Files may be deleted just as with the PET/CBM BASIC SCRATCH command.

SYNTAX

```
DELETE ⟨disk file name⟩[,⟨device⟩]
 ⟨device⟩ is a ⟨numeric expression⟩ whose value is from 4-30
```

EXAMPLES

COMMAND	RESULT
DELETE "0:MYFILE"	deletes MYFILE from drive 0
DELETE "1:TEMP*"	deletes all files on drive 1 beginning with TEMP
DELETE "0:FILE1,FILE2",9	deletes both FILE1 and FILE2 on drive 0 of unit 9

SAMPLE PROGRAM

```
DIM FILE'NAME$ OF 20, TEST$ OF 40
PRINT "DELETE FILES PROGRAM - HIT STOP KEY TO STOP"
INPUT "WHAT DRIVE TO USE: " : DRIVE
REPEAT
  INPUT "WHAT FILE TO DELETE (STOP TO STOP): " : FILE'NAME$;
  IF FILE'NAME$<>"STOP" THEN
    DELETE STR$(DRIVE)+":"+FILE'NAME$
    TEST$=STATUS$
    IF TEST$(5:22)="FILES SCRATCHED,00" THEN
      PRINT "ERROR: NO FILES SCRATCHED"
    ELIF VAL(TEST$)>0 THEN
      PRINT "ERROR: ";TEST$
    ENDIF
  ENDIF
UNTIL FILE'NAME$="STOP"
PRINT "ALL DONE"
```

```
RUN
DELETE FILES PROGRAM - HIT STOP KEY TO STOP
WHAT DRIVE TO USE: 0
WHAT FILE TO DELETE (STOP TO STOP) : MYFILE
WHAT FILE TO DELETE (STOP TO STOP) : TEMP
WHAT FILE TO DELETE (STOP TO STOP) : STOP
ALL DONE
```

SEE ALSO: PASS

KEYWORD: DIM (strings)
CATEGORY: Statement / Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Allocates (dimensions) space for strings. Multiple **DIMs** may occur on one line, separated by commas without repeating the **DIM** keyword. 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**.

SYNTAX

```
DIM ⟨string variable name⟩ OF ⟨maximum number of characters⟩
  ⟨maximum number of characters⟩ is a positive ⟨numeric expression⟩
```

EXAMPLES

```
DIM NAME$ OF 20        allows up to 20 characters for NAME$
DIM PLAYER$ OF MAX     up to the value of MAX characters 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: HOWARD
WHAT IS YOUR FAVORITE FOOD: PIZZA
I SEE THAT YOU LIKE PIZZA, HOWARD
```

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

KEYWORD: DIM (string arrays)
CATEGORY: Statement/Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Allocates (dimensions) space for string arrays. Multiple **DIM**s may occur on one line, separated by commas without repeating the **DIM** keyword. 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**.

SYNTAX

```
DIM ⟨string array name⟩(⟨array index⟩) OF ⟨maximum characters each⟩

 ⟨string array name⟩ is a ⟨string variable name⟩
 ⟨array index⟩ is represented by:
    [⟨bottom limit⟩:]⟨top limit⟩{,[⟨bottom limit⟩:]⟨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⟩
  ⟨maximum characters each⟩ is a positive ⟨numeric expression⟩
```

EXAMPLES

STATEMENT	MEANING
DIM PL$(4) OF 20	allows 4 strings of up to 20 characters each
DIM PL$(1:4) OF 20	the same as the first (strings are referenced by 1, 2, 3, or 4)
DIM I$(5:7) OF 10	allows 3 strings of up to 10 characters each (strings are referenced by 5, 6, or 7)
DIM A$(1:4,1:2) OF 6	two dimensional array, same results as below
DIM A$(4,2) OF 6	the same as the above (strings referenced 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 PLAYER'NAME$(NUMBER) OF 20
FOR X=1 TO NUMBER
   PRINT "PLAYER NUMBER"; X; "NAME PLEASE",
   INPUT ": " : PLAYER'NAME$(X)
NEXT X
FOR X=1 TO NUMBER DO PRINT "PLAYER"; X; "IS"; PLAYER'NAME$(X)

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, REM, RETURN, SGN

SEE ALSO: DIM (strings), DIM (numeric arrays), NEW, PROC, SIZE

KEYWORD: DIM (numeric arrays)
CATEGORY: Statement/Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Allocates (dimensions) space for numeric arrays. Multiple **DIM**s may occur on one line, separated by commas without repeating the **DIM** keyword. 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.

SYNTAX

```
DIM ⟨array name⟩(⟨array index⟩)

 ⟨array name⟩ is a ⟨numeric variable name⟩
 ⟨array index⟩ is represented by:
   [⟨bottom limit⟩:]⟨top limit⟩{,[⟨bottom limit⟩:]⟨top limit⟩}
     ⟨bottom limit⟩ is a ⟨numeric expression⟩ that is optional
       if omitted, the default value is 1
     ⟨top limit⟩ is a ⟨numeric expression⟩
       it must be greater than the ⟨bottom limit⟩
```

EXAMPLES

```
DIM B(-3:0)
DIM C(3,3,5)                                  a three dimensional array
DIM D(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   increment by 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

KEYWORD: DIV
CATEGORY: Operator
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

NOTES

1) Division by 0 is not allowed, therefore the ⟨divisor⟩ may not have a value of 0.
2) Since the **DIV** operator is at a higher level of precedence than the negative sign '-', the expression -5 **DIV** 3 is the same as -(5 **DIV** 3). See **APPENDIX E** for more information about operator precedence.

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

```
REPEAT
  INPUT "1ST NUMBER (0 TO STOP): " : 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
1ST NUMBER (0 TO STOP) : 25 DIVIDED BY: 4
25 DIVIDED BY 4 IS 6 REMAINDER 1
1ST NUMBER (0 TO STOP) : 33 DIVIDED BY: 11
33 DIVIDED BY 11 IS 3
1ST NUMBER (0 TO STOP) : 0
ALL DONE
```

ADDITIONAL SAMPLES SEE: IF, MOD

SEE ALSO: EXP, MOD, SQR

KEYWORD: DO
CATEGORY: Special
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 ⟨controlvariable⟩=⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] DO ⟨statement⟩
  or
WHILE ⟨comparison⟩ DO ⟨statement⟩

     (multi-line)

FOR ⟨controlvariable⟩=⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] [DO]
  or
WHILE ⟨comparison⟩ [DO]

  ⟨controlvariable⟩ 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 X=1 TO 39 DO PRINT "*";
WHILE NOT EOF(2) DO READ A(SCORE)
```

SAMPLE PROGRAM

```
FOR TEMP=1 TO 6 DO PRINT "*";

RUN
* * * * * *
```

SAMPLE

```
SHIFT=152
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 X=1 TO 3
20 PRINT X
30 NEXT

LIST
0010 FOR X: =1 TO 3 DO          notice- COMAL inserted the DO for you
0020    PRINT X                 notice- COMAL indents within
0030 NEXT X                             structures

RUN
1
2
3
```

ADDITIONAL SAMPLES SEE: EXEC, NOT, READ
USED IN PROCEDURES: BOLD'CHAR, BOLD'FACE, CLEAR'FROM, CLEAR'LINE, CURSOR, DISK'GET'STRING
SEE ALSO: FOR, WHILE

KEYWORD: EDIT
CATEGORY: Command
COMAL STANDARD: [NO] VERSION 0.12 [+] VERSION 1.02[*]

Lists lines of the program in the computer's memory, similar to LIST, but does not indent lines within structures. Use in place of LIST when LIST breaks lines on your screen. To store the lines on disk in ASCII format, include a ⟨filename⟩. To avoid confusion with files SAVEd to disk, you should end the ⟨filename⟩ with .L when **EDIT**ing or LISTing lines to disk. These program lines can be retrieved later with the ENTER command. Do not confuse the word **EDIT** to imply that the lines specified are meant to be edited or changed. **EDIT** merely provides a listing of the lines without any indentation within the structures.

NOTES

1) To slow down the listing speed on the screen, hold down the left arrow key (CBM 8000 series) or RVS key (CBM/PET 4000, 3000, and 2000 series).
2) In version 1.02 hit the SPACE bar to pause the listing. Hit it again to resume.
3) **EDIT** to tape is not supported by version 0.12.
4) A program **EDIT**ed to disk in version 0.11 is a PRG (program) type file, while in all other versions it will be a SEQ (sequential) type file. You can override the default PRG type by adding a ,S to the end of the filename. For example: `EDIT "TEST.L,S"`
5) **EDIT** is most useful on 40 column screens to help eliminate `"wrap around"`.

SYNTAX

```
EDIT [⟨range⟩] [[⟨specifier⟩]⟨filename⟩[,⟨unit⟩]]

 ⟨range⟩ is represented by
   ⟨procname⟩ or
   ⟨funcname⟩ or
   ⟨line range⟩ represented by
      [⟨start line⟩] [-] [⟨endline⟩]
        ⟨start line⟩ and ⟨end line⟩ are numbers from 1-9999
 ⟨specifier⟩ is represented by
   either a space or a comma in version 0.12
   either a space or the keyword TO in version 1.02
 ⟨filename⟩ is an optional ⟨disk file name⟩
 ⟨unit⟩ is a ⟨numeric expression⟩ whose value is from 1-30;
   if omitted, the default value is 8 for a CBM disk drive
```

SYNTAX variations	MEANING
EDIT ⟨filename⟩	lists all lines to disk with the file name specified
EDIT ⟨line number⟩	lists only the one line specified
EDIT ⟨start line number⟩-	lists from start line number to the end of the program
EDIT -⟨end line number⟩	lists up to end line number from the beginning of the program
EDIT ⟨start line⟩-⟨end line⟩ ⟨filename⟩	lists from the starting line up to the ending line to disk with the filename specified
EDIT ⟨procedure name⟩	lists the procedure named

EXAMPLES

COMMAND	RESULT
EDIT "TEST.L"	lists program to disk with the name TEST.L
EDIT 250	lists line 250
EDIT 9000-	lists lines 9000 thru end of program
EDIT -50	lists lines 1 - 50
EDIT 500-600 "T1.L"	lists lines 500 - 600 to disk as T1.L
EDIT "PROGRAM.L",1	lists whole program to tape unit 1 (version 1.02 only)
EDIT QUICKSORT	lists the procedure named QUICKSORT

SAMPLE EXERCISE

```
LIST
0010 FOR X:=1 TO 10 DO
0020    PRINT X*X;                    notice this line is indented
0030 NEXT X

EDIT
0010 FOR X:=1 TO 10 DO
0020 PRINT X*X;                       notice this line is not indented
0030 NEXT X
```

SEE ALSO: AUTO, DEL, LIST, RENUM

KEYWORD: ELIF
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 multi-layered structure of **ELIF**s 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 A$="YES" THEN
  EXEC INSTRUCTIONS
```

EXAMPLE 2

```
ELIF ERRORS)5 THEN
  PRINT "OOPS!!!"
```

SAMPLE PROGRAM

```
DIM C$ OF 1
REPEAT
  INPUT "ENTER A LETTER (* TO STOP): " : C$;
  IF C$ IN "AEIOU" THEN
    PRINT "VOWEL"
  ELIF C$ IN "BCDFGHJKLMNPQRSTVWXYZ" THEN
    PRINT "CONSONANT"
  ELIF C$ IN "1234567890" THEN
    PRINT "DIGIT"
  ELSE
    PRINT "NO"
  ENDIF
UNTIL C$="*"
PRINT "ALL DONE"
```

```
RUN
ENTER A LETTER (* TO STOP) : R CONSONANT
ENTER A LETTER (* TO STOP) : U VOWEL
ENTER A LETTER (* TO STOP) : # NO
ENTER A LETTER (* TO STOP) : 2 DIGIT
ENTER A LETTER (* TO STOP) : * NO
ALL DONE
```

ADDITIONAL SAMPLES SEE: ELSE, THEN

SEE ALSO: IF, ELSE, ENDIF, THEN

KEYWORD: ELSE
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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⟩
```

EXAMPLE 1

```
ELSE
  EXEC INSTRUCTIONS
```

EXAMPLE 2

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

SAMPLE PROGRAM

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

RUN
ENTER X-EXIT, Y-YES, OR N-NO: O 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, THEN
USED IN PROCEDURES: SCAN, VALUE
SEE ALSO: IF, ELIF, ENDIF, THEN

KEYWORD: END
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [+] VERSION 1.02[*]

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 (except in version 0.11 SPLIT mode). 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. 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 string expression is not supported by version 0.12. 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 "GOOD-BYE FOR NOW"
```

SAMPLE EXERCISE

```
10 DIM C$ OF 1
20 REPEAT
30 INPUT "TYPE 0 TO STOP: " : C$
40 UNTIL C$="0"
50 END
60 PRINT "ALL DONE"
```

```
LIST
0010 DIM C$ OF 1
0020 REPEAT
0030     INPUT "TYPE 0 TO STOP: ": C$
0040 UNTIL C$ = "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 SAMPLE SEE: LABEL

SEE ALSO: CHAIN, CON, RUN, STOP

KEYWORD: ENDCASE
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 C$ OF 1
REPEAT
  INPUT "YOUR CHOICE (A,S,H,X): ": C$;
  CASE C$ 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 C$="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

KEYWORD: ENDFOR
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

Terminates a multi-line 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 terminators of the other multi-line structures.

NOTE

COMAL will convert the keyword NEXT to ENDFOR for you.

SYNTAX

```
ENDFOR [⟨control variable⟩]

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

EXAMPLES

```
ENDFOR
ENDFOR TEMP
ENDFOR X
```

SAMPLE PROGRAM

```
FOR X=1 TO 3
  FOR Y=3 TO 4
    PRINT X; "PLUS"; Y; "IS"; X+Y
  ENDFOR Y
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, FOR, NEXT, STEP, TO

KEYWORD: ENDFUNC
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Marks the end of a function. Using the new definition of COMAL, all functions require an **ENDFUNC** statement to identify the end of the structure. Version 0.11 uses procedures as functions, and does not use the keywords FUNC or **ENDFUNC**. Versions 0.12 and 1.02 meet the new COMAL definition, which separates the function structure from the procedure structure. See **APPENDIX A** for a description of the PROCEDURE and FUNCTION structures.

NOTE

ENDFUNC is not supported by version 0.11

SYNTAX

```
ENDFUNC [⟨function name⟩]
 ⟨function name⟩ is an optional ⟨identifier⟩
   it must match the function name in its matching FUNC statement
   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(N)
  IF N 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

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

KEYWORD: ENDIF
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Marks the end of the multi-line 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 and 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;"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 TOO HIGH: 3
```

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

KEYWORD: ENDLOOP
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION: 0.12[-] VERSION 1.02 [*]

Marks the end of the multi-line 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 or EXITIF statement (at the time of writing it was not yet decided, but EXIT was implemented). See **Appendix A** for a description of the LOOP structure.

NOTE

ENDLOOP is not supported by version 0.12.

SYNTAX

```
ENDLOOP
```

EXAMPLE

```
ENDLOOP
```

SAMPLE PROGRAM

```
DIM TEMP$ OF 80, ARRAY$(100) OF 80
OPEN FILE 2, "TEST'LOOP",READ
POINTER=0
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: TEMP$
EXITIF TEMP$="*END*" // or IF TEMP$="*END*" THEN EXIT
  POINTER: +1
  ARRAY$(POINTER)=TEMP$
ENDLOOP
CLOSE FILE 2
FOR TEMP=1 TO POINTER DO PRINT ARRAY$(POINTER)
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, EXIT

SEE ALSO: EXIT, LOOP

KEYWORD: ENDPROC
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Marks the end of a procedure. In version 0.11 it can also mark the end of a procedure used as a function (versions 0.12 and 1.02 use the keyword ENDFUNC). 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⟩
   it must match the procedure name in its matching PROC statement;
   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$)
PRINT "GOOD BYE FOR NOW"
//
PROC WELCOME(A$)
  PRINT "WELCOME TO COMAL, ";A$
  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 MOST PROCEDURES
SEE ALSO: CLOSED, ENDFUNC, EXEC, FUNC, PROC, REF

KEYWORD: ENDWHILE
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Marks the end of a multi-line 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
  NUMB1=RND(1,9)
  NUMB2=RND(1,9)
  PRINT "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
WHAT IS 4 PLUS 2 ---> 6 YES
WHAT IS 5 PLUS 1 ---> 6 YES
WHAT IS 9 PLUS 5 ---> 14 YES
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

KEYWORD: ENTER
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

Enters into the computer's memory a program that was stored on disk in ASCII form via a LIST or EDIT command. However, **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 **ENTER**ed. To avoid confusion, end the file name of any file LISTed or EDITed to disk with .L to provide easy identification.

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. Since a program LISTed to disk in version 0.11 is a PRG (program) type file, and other versions expect it to be a SEQ (sequential) type file, the transfer from version 0.11 requires one additional step. After LISTing the program to disk, the file type must be converted from PRG to SEQ. This is accomplished using the program DIRECTORY, provided in **APPENDIX G**. Another way is to override the PRG type when LISTing to disk by adding a ,S to the end of the filename. Example: `LIST "INPROC.L,S"`.
3) **ENTER** from tape is not supported by version 0.12.
4) Future versions of CBM COMAL may have ENTER perform on automatic NEW first, and use a new keyword MERGE to merge sections, without regard to line numbers in the file being merged from tape or disk.
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 // just after the line number and hit ⟨return⟩. The system will continue to enter the remainder of the program. When it is finished, you can go back to the problem lines and correct them to match your version.

SYNTAX

```
ENTER ⟨file name⟩[,⟨unit⟩]

 ⟨file name⟩ is a ⟨disk file name⟩
 ⟨unit⟩ is a ⟨numeric expression⟩ whose value is 1 or 4-30
   if omitted, the default value is 8 for a CBM disk drive
```

EXAMPLES

```
ENTER "GET'CHAR.L"
ENTER "INPROC.L"
ENTER "SPECIALPROG.L",9
```

SAMPLE EXERCISE

```
10 EXEC SAMPLE

LIST
0010 EXEC SAMPLE
ENTER "SAMPLEPROC.L"          contents of file SAMPLEPROC.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: LIST, LOAD, OBJLOAD, SAVE

KEYWORD: EOD
CATEGORY: System function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 equal to TRUE (a value of 1) unless a RESTORE command is issued, after which it is equal to FALSE (a value of 0).

SYNTAX

```
EOD
```

EXAMPLE

```
EOD
```

SAMPLE PROGRAM

```
COUNT=0
WHILE NOT EOD DO
  READ A
  COUNT:+1                                  increment count
ENDWHILE
PRINT "THERE ARE";COUNT;"DATA ITEMS"
RESTORE                                     reset data pointer
REPEAT
  READ A
  PRINT A
UNTIL EOD
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, WRITE

SEE ALSO: DATA, EOF, READ, RESTORE

KEYWORD: EOF
CATEGORY: System function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 (⟨file number⟩)
 ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 2-254
```

EXAMPLES

```
EOF (3)
EOF (INFILE)
```

SAMPLE PROGRAM

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

RUN
   (disk file number 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

KEYWORD: ESC
CATEGORY: System variable
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

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 (except in during an INPUT request in version 0.11). 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.

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. TRUE (a value of 1).
2) While the STOP key is disabled (TRAP **ESC-** in effect) in a running COMAL program, the value of **ESC** will be FALSE (a value of 0) **EXCEPT** while the STOP key is depressed. **ESC** is equal to TRUE (a value of 1) while the STOP key is depressed. Once the key is let up the value of **ESC** returns to FALSE (a value of 0).

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
D=ESC	assign the variable D the value of ESC
PRINT ESC	prints the value of ESC

SAMPLE PROGRAM

```
TRAP ESC-                                  disable the STOP key
PRINT "STOP IS NOT DEPRESSED"
REPEAT                                     wait till STOP is depressed
UNTIL ESC
PRINT "SO YOU WISH TO STOP"
PRINT "PRESS THE SHIFT KEY TO STOP"
SHIFT=152
REPEAT                                     wait till SHIFT is down
UNTIL PEEK(SHIFT)
TRAP ESC+                                  enable the STOP key again
PRINT "ALL DONE"


RUN
   (STOP key is disabled)
STOP IS NOT DEPRESSED
   (nothing happens until you hit the STOP key)
SO YOU WISH TO STOP
PRESS THE SHIFT KEY TO STOP
   (nothing happens till you press the SHIFT key)
   (STOP key is enabled again)
ALL DONE
```

ADDITIONAL SAMPLES SEE: KEY$, TRAP

SEE ALSO: SETEXEC, TRAP

KEYWORD: EXEC
CATEGORY: Statement/Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 (except in version 0.11).

NOTES

1) The word **EXEC** is optional, and if omitted will be supplied by the system.
2) COMAL can suppress the word **EXEC** in a listing. See the keyword SETEXEC for more information about this option. Version 0.11 does not support this feature.
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 (except version 0.11). 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.

SYNTAX

```
[EXEC] ⟨procedure name⟩[(⟨actual parameter list⟩)]
EXEC is optional; if omitted it will be supplied by the system
⟨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 or a procedure function
```

EXAMPLES

```
EXEC INSTRUCTIONS
EXEC ERROR(5)
EXEC BOLD'STRING(NAME$,1)
```

SAMPLE PROGRAM

```
DIM WORD$ OF 80
FOR TEMP=1 TO 3 DO
   INPUT "WORD: " : WORD$;
   INPUT "COUNT: " : COUNT
   EXEC DEMO(COUNT, WORD$)
NEXT TEMP
PRINT "ALL DONE"
//
PROC DEMO(N, T$) CLOSED
   FOR TEMP=1 TO N DO PRINT T$;          variable TEMP is local and
   PRINT                                 won't conflict with the
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: BOLD'FACE, DOUBLE'STRIKE, FETCH, GET'ALPHA, GET'DIGIT, GET'VALID, TAKE'IN
SEE ALSO: ENDFUNC, ENDPROC, FUNC, OBJLOAD, OPTION, PROC, REF, SETEXEC

KEYWORD: EXIT or EXITIF
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

EXIT or **EXITIF** 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. The LOOP structure was a last minute addition to version 1.02 as this HANDBOOK was going to press. The loop exit method was not finalized yet, but will be either an **EXIT** or **EXITIF** statement (but not both). A loop structure should only have one exit or it is not part of structured programming. If there are no statements prior 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

EXIT or **EXITIF** is not supported by version 0.12.

SYNTAX

```
EXITIF ⟨condition⟩
  or
IF ⟨condition⟩ THEN EXIT

 ⟨condition⟩ is a ⟨numeric expression⟩
```

EXAMPLES

```
EXITIF TEXT$ = "*END*"
IF TEXT$ = "*END*" THEN EXIT
```

SAMPLE PROGRAM

```
DIM TEMP$ OF 80, ARRAY$(100) OF 80
OPEN FILE 2, "TEST'LOOP", READ
POINTER=0
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: TEMP$
EXITIF TEMP$="*END*" // or IF TEMP$="*END*" THEN EXIT
  POINTER: +1
  ARRAY$(POINTER)=TEMP$
ENDLOOP
CLOSE FILE 2
FOR TEMP=1 TO POINTER DO PRINT ARRAY$(POINTER)
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, LOOP

SEE ALSO: ENDLOOP, LOOP

KEYWORD: EXP
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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+N)
```

SAMPLE PROGRAM

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

KEYWORD: FALSE
CATEGORY: System constant
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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, REM, RETURN
USED IN PROCEDURE: FETCH
SEE ALSO: TRUE

KEYWORD: FILE
CATEGORY: special
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Specifies that the OPEN, CLOSE, INPUT, PRINT, READ, or WRITE in the statement is to or from a previously opened **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 PET/CBM 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.

SYNTAX (INPUT)

```
INPUT FILE ⟨file number⟩: ⟨variable list⟩
          (PRINT)
PRINT FILE ⟨file number⟩: ⟨value list⟩
          (READ)
READ FILE ⟨file number⟩: ⟨variable list⟩
          (WRITE)
WRITE FILE ⟨file number⟩: ⟨variable list⟩
          (OPEN)
OPEN [FILE] ⟨num⟩,⟨filename⟩[,UNIT ⟨dev⟩[,⟨secondry adr⟩]] [,⟨type⟩]
          (CLOSE)
CLOSE [[FILE] ⟨file number⟩]

 ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨num⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨filename⟩ is a ⟨disk or tape file name⟩
 ⟨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
 ⟨dev⟩ is a ⟨numeric expression⟩ whose value is from 0-30
   if omitted, the default value is 8 for a CBM disk drive
 ⟨secondry adr⟩ is a ⟨numeric expression⟩ whose value is from 0-15
   if omitted, COMAL will supply it
 ⟨type⟩ is either READ, WRITE, APPEND, or RANDOM ⟨record length⟩
   ⟨record length⟩ is a positive ⟨numeric expression⟩
```

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
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 PROCEDURE: DISK' COMMAND
SEE ALSO: APPEND, CLOSE, EOF, INPUT, OPEN, PRINT, READ, STATUS, WRITE

KEYWORD: FOR
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 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 a NEXT statement, except for the one line **FOR** statement which must not use a NEXT. Note that CBM COMAL (except version 0.11) 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 non integer (i.e., STEP .2).
3) See APPENDIX A for a description of the **FOR** structure.

SYNTAX (one line FOR)

```
FOR ⟨controlvariable⟩ =⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] DO ⟨statement⟩

      (multi line FOR)

FOR ⟨controlvariable⟩ =⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] [DO]
  ⟨statements⟩

 ⟨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
```

EXAMPLES

```
FOR TEMP = LOW TO HIGH STEP 2
FOR SCORE = 1 TO MAX
FOR X=1 to 40 DO PRINT " ",
```

SAMPLE PROGRAM

```
INPUT "HOW MANY SCORES: " : NUMBER'SCORES
TOTAL=0
FOR TEMP=1 TO NUMBER'SCORES
   INPUT "SCORE: " : SCORE
   TOTAL: +SCORE                                    add SCORE to TOTAL
NEXT TEMP                          converts to ENDFOR TEMP
PRINT "TOTAL WAS";TOTAL; "FOR AN AVERAGE OF"; TOTAL/NUMBER'SCORES

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

ADDITIONAL SAMPLES SEE: DIM (numeric arrays), EXEC, NEXT, OR, ORD, PEEK, READ, REM, SGN
USED IN PROCEDURES: BOLD'CHAR, BOLD'FACE, CLEAR'FROM, CLEAR'LINE, CURSOR, DISK'GET'INIT, DISK'GET'SKIP, LOWER'TO'UPPER
SEE ALSO: ENDFOR, NEXT, STEP, TO

KEYWORD: FUNC
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 (except in version 0.11 where a procedure is used as a function and ⟨function name⟩ must be assigned a value). Functions can be very flexible, versatile and easy to use. However, using all the options (REF, CLOSED, IMPORT, and parameter passing), can be complex. Functions 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 1.02 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 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.

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 name). If a variable or array is made global with an IMPORT statement, any changes made inside the function will have global effect. Version 1.02 requires that all functions called from within a closed procedure or function be declared global with an IMPORT statement. Version 0.12 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. Both integer and string functions are allowed in addition to the usual numeric function.

NOTES

1) Old version 0.11 uses a procedure as a function, and does not support the keywords FUNC or ENDFUNC. Versions 0.12 and 1.02 are updated to meet the new COMAL definition, which separates the function from the procedure.
2) The system will "remember" all procedure and function names once a program is run (except version 0.11). 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.
3) String functions are not supported by versions 0.12 and 1.02

SYNTAX

```
FUNC ⟨function name⟩ [ (⟨formal parameter list⟩) ] [CLOSED]

  ⟨function name⟩ is an ⟨identifier⟩
    may by ⟨identifier⟩# if used as an integer function
    may by ⟨identifier⟩$ if used as a string function
  ⟨formal parameter list⟩ is optional and represented by:
    [REF ]⟨variable name⟩ {, [REF ]⟨variable name⟩}
```

EXAMPLES

```
FUNC EVEN(N)
FUNC GCD(X,Y)
FUNC JIFFIES
```

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 (N)
   IF N MOD 2 = 0 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
```

ADDITIONAL SAMPLES SEE: ENDFUNC, RETURN

SEE ALSO: CLOSED, ENDFUNC, ENDPROC, EXEC, IMPORT, INTERRUPT, PROC, REF, RETURN

KEYWORD: GET$
CATEGORY: Function
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

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 file number 0. It will then get the specified number of characters from the keyboard before returning the string. (This is different from 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 0.12, however procedures DISK'GET, GET'CHAR, and SCAN can be used as viable substitutes.
2) FILE 0 is the keyboard and doesn't need to be opened or closed.
3) When **GET**ting one character at a time from a disk or tape file, a buffer is used allowing efficient operation.
4) See **APPENDIX C** for more information about sequential files.

SYNTAX

```
GET$(⟨file number⟩,⟨number of characters⟩)

 ⟨file⟩ is a ⟨numeric expression⟩ whose value is from 0-254
 ⟨number of characters⟩ is a ⟨numeric expression⟩
```

EXAMPLES

```
CODE$=GET$(0,8)
PRINT GET$(2,20)
CHAR$=GET$(INFILE,1)
```

SAMPLE PROGRAM

```
// Procedure is courtesy of Lars Laursen
// Horsens, Denmark
PROC LIST'DIRECTORY(DRIVE,MASK$) CLOSED
  DIM CH$ OF 1
  OPEN FILE 2,"$"+STR$(DRIVE)+":"+MASK$,UNIT 8,0,READ
  CH$=GET$(2,2)                              skip two bytes
  WHILE GET$(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
    PRINT
  ENDWHILE
  CLOSE FILE 2
ENDPROC LIST'DIRECTORY

RUN
END AT 0140                             nothing was executed yet

LIST'DIRECTORY(0,"A*")                 direct command

0 "MY DISK              " ID 2C
12    "ADD'DRILL"         PRG
31    "ASSEMBLER"         PRG
9     "A2"                SEQ
589 BLOCKS FREE.
```

SEE ALSO: CURSOR, INPUT, KEY$, READ

KEYWORD: GOTO
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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. In version 1.02, labels inside a closed procedure are considered local. 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 command, 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.
```

ADDITIONAL SAMPLE SEE: LABEL

SEE ALSO: LABEL

KEYWORD: IF
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 X=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      add number to total; increment count
    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: BOLD'CHAR, BOLD'FACE, FETCH, LOWER'TO'UPPER, MULTI'STRIKE, SCAN
SEE ALSO: ELIF, ELSE, ENDIF, THEN

KEYWORD: IMPORT
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [-] VERSION 1.02 [*]

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 they 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 0.12.
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 1.02 must be declared global by using the procedure name (the ⟨identifier⟩) in an **IMPORT** statement. In version 0.12, all procedures and functions are recognized in a closed procedure without the need for an **IMPORT** statement.
7) 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$ = "PET/CBM"
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 PET/CBM
ALL DONE
```

USED IN PROCEDURES: MANY OF THE PROCEDURES
SEE ALSO: CLOSED, ENDFUNC, ENDPROC, EXEC, FUNC, PROC, REF

KEYWORD: IN
CATEGORY: Operator
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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., ⟨string 1⟩ 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

```
⟨string 1⟩ IN ⟨string 2⟩

  ⟨string 1⟩ and ⟨string 2⟩ are ⟨string expressions⟩
```

EXAMPLES

```
"NO" IN VALID'ANSWERS$
ITEM$ IN CHOICES$
```

SAMPLE PROGRAM

```
DIM HIT$ OF 5, L$ 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$)=5
REPEAT
  L$=CHR$(RND(A,Z))
  PRINT L$,
  COUNT:+1
UNTIL L$ 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, ELIF, KEY$, THEN
USED IN PROCEDURES: GET'DIGIT, GET'VALID, LOWER'TO'UPPER
SEE ALSO: CHR$, ORD

KEYWORD: INPUT (from a sequential file)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Allows the user to enter data into a running program from a sequential file that was created by PET/CBM 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 previously have been opened as a READ type file. **INPUT** FILE may also be used to read characters directly off the screen. The file must be opened with a UNIT of 3. Example: OPEN 3," ",UNIT 3,READ.

NOTES

1) **INPUT** FILE is used to read files created by PRINT FILE or PET/CBM BASIC files created with PRINT#.
2) **INPUT** FILE is not compatible with READ FILE.
3) **INPUT** FILE cannot be used to read files created by WRITE FILE.
4) **INPUT** FILE can read characters directly off the screen, a line at a time.
5) **INPUT** FILE can be used to read, one line at a time, a program that was LISTed to tape or disk.
6) See **APPENDIX C** for more information about sequential files.

SYNTAX

```
INPUT FILE ⟨file number⟩ : ⟨variable list⟩

 ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨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 2, "SAMPLE'TEXT", READ
WHILE NOT EOF(2)
  INPUT FILE 2 : TEXT$
  PRINT TEXT$
ENDWHILE
CLOSE
PRINT "ALL DONE"
```

```
RUN
   (file 2 named SAMPLE'TEXT is opened for input)
TESTING
1 2 3 4
   (file 2 is closed)
ALL DONE
```

ADDITIONAL SAMPLE SEE: UNIT

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

KEYWORD: INPUT (from a random file)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Allows the user to enter data into a running program from a random access file that was created by PET/CBM 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 previously have been opened as a RANDOM type file. **INPUT** FILE may also be used to read characters directly off the screen. The file must be opened with a UNIT of 3.

NOTES

1) **INPUT** FILE is used to read files created by PRINT FILE or PET/CBM BASIC files created with PRINT#.
2) **INPUT** FILE is not compatible with READ FILE.
3) **INPUT** FILE cannot be used to read files created by WRITE FILE.
4) **INPUT** FILE can read characters directly off the screen.

SYNTAX

```
INPUT FILE ⟨file num⟩,⟨record number⟩[,⟨offset⟩]: ⟨variable list⟩

 ⟨file num⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨record number⟩ 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 2, "RANDOM'SAMPLE", 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'SAMPLE 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

KEYWORD: INPUT
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 **INPUT** accepts every character typed, and can only be terminated with a carriage return. It will accept commas, semicolons, colons, and quote marks as part of string data being input. 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 semi-colon, no line feed or carriage return is provided when the user hits the RETURN key after his or her input (one space is printed instead).

NOTE

The STOP key is disabled while a program is waiting for response to an **INPUT** statement in version 0.11. However, in versions 0.12 and 1.02 the STOP key can be used even during an **INPUT** request.

SYNTAX

```
INPUT [⟨prompt⟩: ]⟨variable name⟩ {, ⟨variable name⟩} [⟨mark⟩ ]

 ⟨prompt⟩ is a ⟨string expression⟩ ;
   if omitted, a question mark will be supplied by the system
 ⟨variable list⟩ is one or more ⟨variable names⟩ to be used
   for the operation separated by commas
 ⟨mark⟩ is a semicolon;
   if it is used, a space will be issued after the users input
    instead of a carriage return
   in version 1.02, it may also be a comma and if it is used, spaces
    will be issued up to the next zone.
```

EXAMPLES

```
INPUT GUESS
INPUT "WHAT IS YOUR NAME: " : NAME$
INPUT PROMPT$ : ANSWER;
INPUT "COORDINATES: " : X, Y, Z
INPUT "AGE, NAME: " : AGE, NAME$
```

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: DIM (string arrays), INT, RND, SELECT, THEN

SEE ALSO: CURSOR, FILE, GET$, KEY$, READ

KEYWORD: INT
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

SYNTAX

```
INT (⟨numeric expression⟩)
```

EXAMPLES

```
INT(GUESS)
INT(5.65)
INT(X/Y)
```

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
```

SEE ALSO: ABS, SGN, VAL

KEYWORD: INTERRUPT
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

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 0.12.
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 monitoring for interrupt
```

SAMPLE

```
INTERRUPT BLINKER              set up the interrupt
PRINT "[CLR]"                   clear the screen
PRINT "MONITORING THE IEEE 488 BUS FOR AN INTERRUPT"
PRINT "TIME IN JIFFIES: "
REPEAT
  CURSOR 3,18                  position the cursor
  PRINT TIME, SPC$(7)          print the number of jiffies
UNTIL FALSE                    forever
//
PROC BLINKER
  REPEAT
    CURSOR 12,20                                position the cursor
    IF RND(1,2)=1 THEN PRINT "[RVS]",           turn reverse on sometimes
    PRINT "INTERRUPT REQUEST - HIT SHIFT WHEN READY"
    FOR X=1 TO 99 DO NULL // PAUSE
  UNTIL PEEK(SHIFT)                             wait till SHIFT is hit
  PRINT "[CLR]"                                 clear screen
  INTERRUPT                                     disable interrupt
  PRINT "INTERRUPT NOW DISABLED"
ENDPROC BLINKER                                 returns to previous place

RUN
  (screen clears)
MONITORING THE IEEE 488 BUS FOR AN INTERRUPT

TIME IN JIFFIES: 458924
  (the time is continually updated until a signal is sensed
   on the SRQ line -
   then INTERRUPT REQUEST begins to blink in the middle
   of the screen)
  (hit SHIFT to stop - and it returns to printing the time)
```

SEE ALSO: OBJLOAD, OPTION

KEYWORD: KEY$
CATEGORY: Function
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

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 version 0.12; however, procedures GET'CHAR and SCAN (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"           if no key was hit or wrong one, go
PRINT CHOICE$                      and repeat looking at the keyboard
PRINT "ALL DONE"                   until an A,E,I,O,or U is hit
TRAP ESC+                          enable the STOP key

RUN
PLEASE ENTER A VOWEL: E          hit any 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

KEYWORD: LABEL
CATEGORY: Identifier
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 1.02) and under normal situations is rarely used. A **label** statement is non-executable 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; however, it takes up extra memory like a variable.

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 1.02.
3) A label inside a CLOSED procedure is considered local in version 1.02. In version 0.12 it is always considered global.

SYNTAX

```
⟨label name⟩:

 ⟨label name⟩ is an ⟨identifier⟩
```

EXAMPLES

```
RESPONSE:
QUICK'QUIT:
```

SAMPLE PROGRAM

```
DIM ITEM$ OF 1
INPUT "WHICH JUMP - A OR B: " : ITEM$
IF ITEM$ = "A" THEN
  GOTO JUMPA
ELIF ITEM$ = "B" THEN
  GOTO JUMPB
ENDIF
END
//
JUMPA:
PRINT "THIS IS TERRIBLE PROGRAMMING"
END
//
JUMPB:
PRINT "YOU SHOULD NEVER HAVE TO USE A LABEL JUMP"
```

```
RUN
WHICH JUMP - A OR B: A
THIS IS TERRIBLE PROGRAMMING
```

ADDITIONAL SAMPLES SEE: DATA, GOTO

SEE ALSO: GOTO, RESTORE

KEYWORD: LEN
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Returns the length of the specified string. All characters, even blanks 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$)
LEN("TESTING")
LEN(A$+B$)
```

SAMPLE PROGRAM

```
DIM TEXT$ OF 80
REPEAT
  INPUT "TYPE SOMETHING: " : TEXT$
  IF TEXT$⟩"" THEN PRINT "THE LENGTH OF ⟨" ;TEXT$; "⟩ IS";LEN(TEXT$)
UNTIL TEXT$="""
PRINT "ALL DONE"

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

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

KEYWORD: LET
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

LET is an implied keyword. You never need to explicitly use it 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, a := 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 semi-colon (;). A :+ is used for incremental assignment (to add 5 to the variable TOTAL you would code: TOTAL:+5), and a :- 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 0.12.
2) The keyword **LET** is not listed in a program listing.
3) The new COMAL definition changed the way a substring is specified. Using the old method (version 0.11) you specified the ⟨start⟩ character and the number of characters to use (⟨length⟩). Using the new method (versions 0.12 and 1.02) you specify the ⟨start⟩ and ⟨end⟩ character.

SYNTAX (direct assignment)

```
⟨variable or array element⟩ : = ⟨value⟩

  ⟨value⟩ is an ⟨expression⟩ of the same type (string or numeric)
    as the ⟨variable or array⟩ on the left side of the statement

        (partial string assignment)

⟨string variable name⟩ (⟨substring part⟩) : =⟨string expression⟩
        or
⟨string array⟩ (⟨index part⟩) (⟨substring part⟩) : =⟨string expression⟩

  ⟨string array⟩ is the string array name
  ⟨index part⟩ specifies which element in the array:
    ⟨index number⟩ {, ⟨next index number⟩}
  ⟨substring part⟩ specifies which part of the string:
    ⟨start⟩ [: ⟨end⟩ ]
      or
    ⟨start⟩ [: ⟨length⟩ ]                    in OLD version 0.11
      ⟨start⟩ is the position in the string where the
        substring will begin
      ⟨length⟩ is how many consecutive characters to use
        if omitted only the ⟨start⟩ character is used
      ⟨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)

⟨string A⟩ : + ⟨string expression⟩          version 1.02 only
        or
⟨string A⟩ : = ⟨string A⟩+⟨string expression⟩

  ⟨string A⟩ 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$: =REPLY$+A$                     concatenate REPLY$ and A$
REPLY$: +A$                            same as above (version 1.02 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 : = and LET is gone
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

KEYWORD: LIST
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 **LIST**ing or EDITing lines to disk. A program **LIST**ed to disk or tape can be retrieved later via the ENTER command. To **LIST** lines to your printer, issue the command SELECT "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** cancels any previous SELECT "LP" command).

NOTES

1) To slow down the listing speed on the screen hold down the left arrow key (CMB 8000 series) or RVS key (CBM/PET 4000, 3000, and 2000 series).
2) In version 1.02, hit SPACE to pause a listing. Hit SPACE again to resume.
3) **LIST** to tape is not supported by version 0.12.
4) A program **LIST**ed to disk (or tape in version 1.02) can later be read with INPUT FILE statements (except 0.11).
5) Use EDIT instead of **LIST** if you do not want the indentation of structures.
6) A program **LIST**ed to disk in version 0.11 is a PRG (program) type file, while it is a SEQ (sequential) type file in the other versions. The default PRG type can be overridden by adding a ,S to the end of the filename. Example: `LIST "TEST'PROGRAM.L,S"`
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 0.12.
9) Version 1.02 requires a drive number as part of the disk file name.

SYNTAX

```
LIST [⟨range⟩] [ [⟨specifier⟩]⟨file name⟩[,⟨unit⟩]]

 ⟨range⟩ is represented by
   ⟨procname⟩ or
   ⟨funcname⟩ or
   ⟨line range⟩ represented by
      [⟨start line⟩] [-] [⟨end line⟩]
        ⟨start line⟩ and ⟨end line⟩ are numbers from 1-9999
 ⟨specifier⟩ is represented by
   either a space or a comma in version 0.12
   either a space or the keyword TO in version 1.02
 ⟨file name⟩ is an optional ⟨disk or tape file name⟩
 ⟨unit⟩ is a ⟨numeric expression⟩ whose value is 1 or 4-30;
   if omitted, it defaults to 8 for a CBM disk drive
```

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 "TEST'PROGRAM.L",1	lists all lines to tape, unit 1.
LIST QUICKSORT	lists procedure named QUICKSORT

SAMPLE EXERCISE

```
10 FOR TEMP=1 TO 20
30 NEXT 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 NEXT 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, EDIT, ENTER, RENUM

KEYWORD: LOAD
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 or while in SPLIT MODE via the RUN command (not via a LIST or EDIT to disk or tape).

NOTES

1) **LOAD** and ENTER commands are not compatible.
2) **LOAD** from tape is not supported by version 0.12.

SYNTAX

```
LOAD ⟨program name⟩ [,⟨unit⟩ ]

 ⟨program name⟩ is a ⟨disk or tape file name⟩
   it cannot be a variable name
 ⟨unit⟩ is a ⟨numeric expression⟩ whose value is 1 or 4-30;
   if omitted, the default is 8 for a CBM disk drive
```

EXAMPLES

```
LOAD "PROCESS"
LOAD "SPECIALS",9
```

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"          program will vary
0210 PRINT "IT JUST CAME FROM DISK"           with contents of ANOTHER

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

SEE ALSO: ENTER, LIST, OBJLOAD, SAVE, STATUS, UNIT, VERIFY

KEYWORD: LOG
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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*F)
LOG(39)
LOG(ABS(X))
```

SAMPLE PROGRAM

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

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

SEE ALSO: ATN, COS, SIN, TAN

KEYWORD: LOOP
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

Provides a loop structure with its one exit condition inside the body of the 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. The LOOP structure was a last minute addition to version 1.02 as this Handbook was going to press. The loop exit method was not finalized yet, but will be either an **EXIT** or **EXITIF** statement (but not both). 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. See **APPENDIX D** for more information on the loop structure.

NOTE

LOOP is not supported by version 0.12.

SYNTAX

```
LOOP
```

EXAMPLE

```
LOOP
```

SAMPLE PROGRAM

```
DIM TEMP$ OF 80, ARRAY$(100) OF 80
OPEN FILE 2,"TEST'LOOP",READ
POINTER=0
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: TEMP$
EXITIF TEMP$="*END*" // or IF TEMP$="*END*" THEN EXIT
  POINTER: +1
  ARRAY$(POINTER)=TEMP$
ENDLOOP
CLOSE FILE 2
FOR TEMP=1 TO POINTER DO PRINT ARRAY$(POINTER)
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, EXIT

SEE ALSO: ENDLOOP, EXIT

KEYWORD: MOD
CATEGORY: Operator
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

NOTES

1) Since you cannot divide by 0, the ⟨divisor⟩ may not have a value of 0.
2) Since the **MOD** operator is at a higher level of precedence than the negative sign '-', the expression -5 **MOD** 3 is the same as -(5 **MOD** 3). See **APPENDIX E** for more information on operator precedence.

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

SEE ALSO: DIV, INT

KEYWORD: NEW
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

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)
```

SEE ALSO: BASIC, DEL

KEYWORD: NEXT
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Terminates a multi-line 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 structure's terminators.

SYNTAX

```
NEXT [⟨control variable⟩]

 ⟨control variable⟩ is a ⟨numeric variable name⟩
   it matches the ⟨control variable⟩ in its matching FOR statement
   if omitted, it will 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

KEYWORD: NOT
CATEGORY: Operator
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 ⟨numeric expression⟩
```

EXAMPLES

```
NOT TEMP
NOT (A AND B)
```

SAMPLE PROGRAM

```
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

```
DIM TYPE$ (FALSE: TRUE) OF 5
TYPE$ (FALSE) = "FALSE"; TYPE$ (TRUE) = "TRUE"
ZONE 6
PRINT "NOT CHART"
PRINT "———"
FOR A=FALSE TO TRUE
   PRINT "A= "; TYPE$ (A) ,  "NOT A= "; TYPE$ (NOT A)
NEXT 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, OR

KEYWORD: NULL
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

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 program's variables.

NOTE

NULL is not supported by version 0.11.

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(1)
NEXT TEMP
PRINT
PRINT "ALL DONE"
//
PROC PAUSE(SECONDS) CLOSED
  FOR TEMP= 1 TO 775*SECONDS 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

KEYWORD: OBJLOAD
CATEGORY: Command / Statement
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

Loads an object code file from disk or tape into the top of the computer's memory, and lowers the top of memory pointer so it will be protected. Object code file names should end with .OBJ to provide for easy identification. A technical paper describing the process of creating a file for use with **OBJLOAD** is available to those who purchased the COMAL ROM board from INSTRUTEK.

NOTE

OBJLOAD is not supported by version 0.12.

SYNTAX

```
OBJLOAD ⟨filename⟩[,⟨unit⟩]

 ⟨filename⟩ is a ⟨disk or tape file name⟩
 ⟨unit⟩ is a ⟨numeric expression⟩ whose value is 1 or 4-30;
   if omitted it defaults to 8 for a CBM disk drive
```

EXAMPLES

```
OBJLOAD "MACHINE.OBJ"
OBJLOAD NAME$
```

SAMPLE

```
PRINT "LOADING OBJECT CODE"
OBJLOAD "0:COMPRESS.OBJ"
IF VAL(STATUS$))0 THEN
  PRINT "LOAD ERROR"
ELSE
  PRINT "NOW LOADING COMPRESSOR PROGRAM"
  CHAIN "COMPRESSOR"
ENDIF

RUN
LOADING OBJECT CODE
   (the object code in the file named COMPRESS.OBJ is loaded)
NOW LOADING COMPRESSOR PROGRAM
   (the program named COMPRESSOR is now loaded and run)
```

SEE ALSO: ENTER, INTERRUPT, LOAD, OPTION

KEYWORD: OF
CATEGORY: Special
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 ⟨maximum characters each⟩

 ⟨maximum characters each⟩ is a positive ⟨numeric expression⟩
 ⟨maximum string length⟩ is a positive ⟨numeric expression⟩
 ⟨array index⟩ is represented by:
   [⟨bottom limit⟩:]⟨top limit⟩{, [⟨bottom limit⟩:]⟨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 C$ OF 1
REPEAT
   INPUT "ARE YOU DONE YET: " : C$
   CASE C$ 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

KEYWORD: OPEN
CATEGORY: Command / Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 disk sequential 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. To open the screen as a file, specify a UNIT of 3 (i.e., **OPEN** 3,"",UNIT 3, READ).
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).

SYNTAX

```
OPEN [FILE] ⟨filenum⟩,⟨name⟩[,UNIT ⟨dev⟩[,⟨secondry adr⟩]],[⟨type⟩]

 ⟨filenumb⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨filename⟩ is a ⟨disk or tape file name⟩
 ⟨dev⟩ is a ⟨numeric expression⟩ whose value is from 0-30
   if omitted, the default value is 8 for a CBM disk drive
 ⟨secondry adr⟩ is a ⟨numeric expression⟩ whose value is from 0-15
   if omitted, COMAL will supply it
 ⟨type⟩ is either READ, or WRITE, or APPEND, or RANDOM ⟨record len⟩
   ⟨record len⟩ is a positive ⟨numeric expression⟩
```

EXAMPLES

```
OPEN FILE 2,"TESTFILE",READ
OPEN INFILE,INNAME$,RANDOM REC'SIZE
OPEN 5,"1:SPECIAL'FILE",UNIT 9,2,WRITE
```

SAMPLE

```
DIM FILENAME$ OF 20, ITEM$ OF 40
FILENAME$ = "VISITORFILE"
INFILE = 4
OPEN INFILE, FILENAME$, READ
REPEAT
  READ FILE INFILE : ITEM$
  PRINT ITEM$
UNTIL EOF(INFILE)
CLOSE INFILE
PRINT "ALL DONE"

RUN
   (system opens disk file number 4 named VISITORFILE for input)
COMAL USERS GROUP          varies depending upon contents of file
   (file number 4 is closed)
ALL DONE
```

ADDITIONAL SAMPLES SEE: CLOSE, WRITE
USED IN PROCEDURE: DISK' COMMAND
SEE ALSO: APPEND, CLOSE, EOF, FILE, INPUT, LOAD, PRINT, RANDOM, READ, SAVE, STATUS, WRITE, UNIT

KEYWORD: OPTION
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

Allows a COMAL program to select a COMAL compatible ROM with a specified ⟨starting address⟩. The names of the external procedures and functions held in the ROM are copied into the program's symbol table together with their addresses. They now may be treated as part of the COMAL system and may be called in the same way as a user defined procedure or function.

NOTES

1) **OPTION** is not supported by version 0.12.
2) To use **OPTION** a COMAL compatible ROM must be available to the system beginning at the memory address specified.
3) **OPTION** can also address RAM memory as if it were ROM after an OBJLOAD statement has been performed.

SYNTAX

```
OPTION ⟨starting address⟩

 ⟨starting address⟩ is a ⟨numeric expression⟩
   it must have a value of a valid memory location
```

EXAMPLES

```
OPTION 8*4096
OPTION LOC
```

SAMPLE

```
OPTION 8*4096              ROM is in the 8th socket
DIM TEMP$ OF 30
PRINT "TESTING THE EXTERNAL ROM PROCEDURES"
REPEAT
   INPUT "ENTER A DECIMAL NUMBER (0 TO STOP) : ": TEMP$
   X=INTVAL(TEMP$,2) // INTVAL IS ROM PROCEDURE
   PRINT "THE VALUE OF";TEMP$; "BECOMES";FLOAT(X)
UNTIL TEMP$="0"
PRINT "ALL DONE"
```

```
RUN                          this requires the COMAL compatible ROM
TESTING THE EXTERNAL ROM PROCEDURES
ENTER A DECIMAL NUMBER (0 TO STOP) : 536.86
THE VALUE OF 536.86 BECOMES 53686
ENTER A DECIMAL NUMBER (0 TO STOP) : 123.4
THE VALUE OF 123.4 BECOMES 12340
ENTER A DECIMAL NUMBER (0 TO STOP) : 0
THE VALUE OF 0 BECOMES 0
ALL DONE
```

SEE ALSO: INTERRUPT, LABEL, OBJLOAD, SYS

KEYWORD: OR
CATEGORY: Operator
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

SYNTAX

```
⟨numeric expression⟩ OR ⟨numeric expression⟩
```

EXAMPLES

```
A(65 OR A)90
ITEM$ = "Y" OR ITEM$ = "N"
```

SAMPLE PROGRAM

```
DIM C$ OF 1
REPEAT
   INPUT "ENTER A LETTER (0 TO STOP) : " : C$;
   IF C$("A" OR C$)"Z" THEN PRINT "NO";
   PRINT // CARRIAGE RETURN
UNTIL C$ = "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

```
DIM TYPE$ (FALSE: TRUE) OF 5
TYPE$ (FALSE) = "FALSE"; TYPE$ (TRUE) = "TRUE"
ZONE 6
PRINT "OR CHART"
PRINT "——"
FOR A=FALSE TO TRUE
  FOR B=FALSE TO TRUE
    PRINT "A= "; TYPE$ (A) , "B= " ; TYPE$ (B) ,
    PRINT "A OR B= "; TYPE$ (A OR B)
  NEXT B
NEXT 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, NOT

KEYWORD: ORD
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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$)
ORD(ITEM$(X))
```

SAMPLE PROGRAM

```
ZONE 12                                          for easy tab to second column
A=ORD("A");Z=ORD("Z")
DIM WORD$ OF 40, LETTERS(A:Z)
INPUT "WORD: " : WORD$
FOR X=1 TO LEN(WORD$)
  CHAR=ORD(WORD$(X))
  LETTERS(CHAR):+1                               increment letter count
NEXT X
FOR X=A TO Z
  IF LETTERS(X) THEN
    PRINT "LETTER: ";CHR$(X), "OCCURANCES: ";LETTERS(X)
  ENDIF
NEXT X
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 PROCEDURE: VALUE
SEE ALSO: CHR$, STR$, VAL

KEYWORD: OTHERWISE
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 C$ OF 1
REPEAT
  INPUT "GOOD, BAD, OR UGLY (G,B,U): " : C$;
  CASE C$ OF
  WHEN "G"
    PRINT "GOOD BYE"
  WHEN "B"
    PRINT "BAD"
  WHEN "U"
    PRINT "UGLY"
  OTHERWISE
    PRINT "TRY AGAIN"
  ENDCASE
UNTIL C$="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

KEYWORD: OUTPUT
CATEGORY: Special
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

Allows you to select (choose) the output location. Recognized output locations include "DS" for Data Screen and "LP" for Line Printer. All printed output from a running program is directed to the output location selected via SELECT **OUTPUT**. If no SELECT OUTPUT statement is given, the screen is used. While in interactive mode, "DS" is default. After a LIST command, output returns to the screen (LIST cancels any previous SELECT **OUTPUT** "LP" command). The word **OUTPUT** is optional, and if omitted will be supplied by the COMAL interpreter.

NOTES

1) INPUT prompts and error messages are directed to the screen even if a SELECT "LP" has been issued.
2) The device number of the printer may be specified as part of the "LP" string by including it immediately after the "LP" (i.e., "LP5"). (Not supported by version 0.12.)

SYNTAX

```
SELECT [OUTPUT] ⟨type⟩

 ⟨type⟩ is a ⟨string expression⟩ representing the output location
   recognized locations are: "DS" for Data Screen
                             "LP" for Line Printer
                             "LPx" for Line Printer
                                x is a digit 4-9 for the device no.
```

EXAMPLES

```
SELECT OUTPUT "DS"
SELECT OUTPUT O$
SELECT "LP5"
```

SAMPLE PROGRAM

```
SELECT OUTPUT "DS"              make sure we are using the screen
DIM OP$ OF 2
PRINT "WHERE WOULD YOU LIKE THE OUTPUT: "
REPEAT
   INPUT "SCREEN OR PRINTER (S/P): " : OP$
UNTIL OP$ = "S" OR OP$ = "P"
IF OP$ = "S" THEN
   OP$ = "DS"
ELSE
   OP$ = "LP"
ENDIF
SELECT OP$                        no need to type the word OUTPUT
PRINT "ALL DONE"
SELECT OUTPUT "DS"          make sure to reroute output to screen

RUN
WHERE WOULD YOU LIKE THE OUTPUT:
SCREEN OR PRINTER (S/P): Y
SCREEN OR PRINTER (S/P): S
ALL DONE
```

SEE ALSO: PRINT, SELECT, USING

KEYWORD: PASS
CATEGORY: Command / Statement
COMAL STANDARD: [NO] VERSION 0.12 [+] VERSION 1.02[*]

PASSes a string expression to the CBM disk. These strings will be interpreted by the disk drive as DOS commands.

NOTES

1) **PASS** is not supported by version 0.11. Procedure DISK'COMMAND, listed in **APPENDIX D**, may be used instead.
2) ⟨device⟩ may not be used in version 0.12.

SYNTAX

```
PASS ⟨disk command⟩ [,⟨device⟩]

 ⟨disk command⟩ is a ⟨string expression⟩
 ⟨device⟩ is a ⟨numeric expression⟩
```

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=0: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, DELETE

KEYWORD: PEEK
CATEGORY: Function
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

Returns the decimal value of the contents in the specified memory location. The result is always an integer from 0 - 255. **PEEK** tends to be very machine dependent, i.e., not very portable between the various models of Commodore computers or other COMAL computers. A chart listing some memory locations of interest follows:

LOCATION	PURPOSE OF LOCATION
141-143	JIFFY CLOCK
151	LAST KEY PRESSED (255=NONE)
152	SHIFT KEY (0=UP 1=DOWN)
158	KEYSTROKE BUFFER COUNT
159	REVERSE FIELD (0=OFF 1=ON)
198	POSITION OF CURSOR ON LINE (0-79)
205	QUOTE MODE (0=OFF 1=ON)
216	LINE THAT THE CURSOR IS ON (0-24)
623-632	KEYBOARD BUFFER
32768-33767	VIDEO SCREEN MEMORY (40 COLUMN)
32768-34767	VIDEO SCREEN MEMORY (80 COLUMN)

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

```
VIDEOSTART=32768
VIDEOLENGTH=2000                          use 1000 for a 40 column screen
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
NEXT X
PRINT "ALL DONE"

RUN
   (each character on the screen reverses)
ALL DONE
```

SAMPLE PROGRAM

```
SHIFT=152
REPEAT
   IF NOT PEEK(SHIFT) THEN PRINT "[CLR]"
   IF PEEK(SHIFT) THEN PRINT "[HOME]SHIFT IS DOWN"
UNTIL FALSE          note: replace [CLR] and [HOME] with that key

RUN
   (screen clears)
   (depress the SHIFT key and the words SHIFT IS DOWN appear)
   (HIT the RUN/STOP key to stop this program)
```

ADDITIONAL SAMPLE SEE: DO
USED IN PROCEDURES: DISK'GET, GET'CHAR, JIFFIES, POS, SCAN, SHIFT
SEE ALSO: POKE, SYS

KEYWORD: POKE
CATEGORY: Function
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

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: **POKE**ing 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 **POKE**ing around with.

LOCATION	PURPOSE OF LOCATION
141-143	JIFFY CLOCK
144	LOCATION TO DISABLE STOP KEY
158	KEYSTROKE BUFFER COUNT
159	REVERSE FIELD (0=OFF 1=ON)
198	POSITION OF CURSOR ON LINE (0-79)
205	QUOTE MODE (0=OFF 1=ON)
216	LINE THAT THE CURSOR IS ON (0-24)
623-632	KEYBOARD BUFFER
32768-33767	VIDEO SCREEN MEMORY (40 COLUMN)
32768-34767	VIDEO SCREEN MEMORY (80 COLUMN)
59468	12=GRAPHIC MODE 14=LOWER CASE MODE

To clear the keyboard buffer you could use the following statement:

```
POKE 158,0
```

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 from 0-255
```

EXAMPLES

```
POKE 59468,12
POKE LOC,CONTENTS
```

SAMPLE PROGRAM

```
VIDEOSTART=32768
VIDEOLENGTH=2000                          use 1000 for a 40 column screen
A=ORD("A"); Z=ORD("Z")
PRINT "[CLR]"          note: replace [CLR] with the clear screen key
FOR TEMP=1 TO 99
  POKE VIDEOSTART+RND(0,VIDEOLENGTH),RND(A,Z)
NEXT TEMP
PRINT "ALL DONE"

RUN
   (the screen clears)
   (the screen now begins to fill randomly with random letters)
   (99 random screen locations are changed)
ALL DONE
```

ADDITIONAL SAMPLE SEE: PEEK
USED IN PROCEDURES: CLEAR'FROM, CLEAR'LINE, CLEAR'TO, DISK'GET, DISK'GET'INIT, GET'CHAR, SCAN
SEE ALSO: OPTION, PEEK, SYS

KEYWORD: PRINT
CATEGORY: Command / Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 set ZONE. (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. A CHR$(13) and CHR$(10) are used as delimiters between records by the system with a file created by **PRINT** FILE statements. Version 1.02 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.

NOTES

1) **PRINT** USING and PRINT AT are not supported by version 0.12.
2) 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).
3) **PRINT** FILE and WRITE FILE are not compatible.
4) Use INPUT FILE to read a file created with **PRINT** FILE.
5) **PRINT** FILE is compatible with PET/CBM BASIC PRINT#.

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 ⟨file number⟩:] [⟨print list⟩] [⟨continue mark⟩]

  plus version 1.02 allows the addition of AT ⟨row⟩,⟨col⟩:
PRINT AT ⟨row⟩,⟨col⟩: [⟨print list⟩] [⟨continue mark⟩]
 ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨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 the USING image with 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.
⟨continue 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 1-25
⟨col⟩ is a ⟨numeric expression⟩ whose value is from 1-80
```

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: "*"
```

SAMPLE PROGRAM

```
DIM I$ OF 40
I$ = "VARIABLES"; MONEY = 25.6
PRINT TAB(5), "CONSTANTS AND"; I$
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: BOLD'CHAR, BOLD'FACE, CURSOR, DOUBLE'STRIKE, FETCH, MULTI'CHAR, SET'PITCH, TAKE'IN
SEE ALSO: AT, CURSOR, FILE, INPUT, OPEN, OUTPUT, READ, SELECT, TAB, UNIT, USING, WRITE, ZONE

KEYWORD: PRINT (to a RANDOM file)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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.

NOTES

1) **PRINT** USING is not supported by version 0.12.
2) **PRINT** FILE and WRITE FILE are not compatible.
3) Use INPUT FILE to read a record created with **PRINT** FILE.

SYNTAX

```
PRINT FILE ⟨filenum⟩,⟨recordnum⟩[,⟨offset⟩]: ] [⟨printlist⟩] [⟨mark⟩]

 ⟨filenum⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨recordnum⟩ is a positive ⟨numeric expression⟩
 ⟨offset⟩is a positive ⟨numeric expression⟩;
   if omitted, no bytes will be skipped
 ⟨printlist⟩ 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⟩: ⟨numeric expression list⟩
     ⟨string expression⟩ may be a constant or variable
       used to set up the USING image with the following reserved
         #  reserves a digit place
         .  the location of the decimal point
         -  floating minus sign (optional)
     ⟨numeric expression list⟩ is a list of the expressions 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
```

EXAMPLES

```
PRINT FILE 2,5 : "NAME: "+NAME$
PRINT FILE OUTFILE,RECNUM : TEXT$
```

SAMPLE PROGRAM

```
DIM TEXT$ OF 80
OPEN 2,"RANDOM'SAMPLE",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'SAMPLE 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)
ALL DONE
```

SEE ALSO: FILE, INPUT, OPEN, TAB, UNIT, USING, WRITE, ZONE

KEYWORD: PROC
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Start of a procedure, allowing parameter passing and local or global variables. In OLD version 0.11 it may also be used as a multi-line function if somewhere within the procedure the ⟨procedure name⟩ is assigned a value (versions 0.12 and 1.02 meet the new COMAL definition and use keywords FUNC and ENDFUNC to define a function). Procedures can be very flexible, versatile and easy to use. However, using all the options (REF, CLOSED, IMPORT, and parameter passing) can be complex but powerful. Procedures may call other procedures, or even call itself. Version 1.02 also allows procedures to be nested (i.e., you can have 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 1.02 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). 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 procedure parameter variable or array 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 1.02 also requires that

all procedures or functions called from within a closed procedure be declared global with an IMPORT statement. Version 0.12 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.

NOTES

1) Old version 0.11 may use a procedure as a function, and does not support the keywords FUNC or ENDFUNC. Versions 0.12 and 1.02 are updated to meet the new COMAL definition, which separates the function from the procedure.
2) The system will "remember" all procedure names once a program is run (except in version 0.11). 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.

SYNTAX

```
PROC ⟨procedure name⟩ [ (⟨formal parameter list⟩) ] [CLOSED]

 ⟨procedure name⟩ is an ⟨identifier⟩
 ⟨formal parameter list⟩ 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 will be considered LOCAL
```

EXAMPLES

```
PROC FIND'IT
PROC ERRORPRINT(E, REF ER$(,))
PROC SORT CLOSED
PROC PRINT'IT(N$, A$, C$, S$, Z$) CLOSED
```

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 X=LEN(A$) TO 1 STEP -1 DO PRINT A$(X),
  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

KEYWORD: RANDOM
CATEGORY: Type of file
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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. This is more complicated than a sequential file (see READ, WRITE, and APPEND). If you are new to programming, you may wish to master sequential file manipulation first. The word FILE is optional and if omitted will be supplied by the system.

SYNTAX

```
OPEN [FILE] ⟨num⟩,⟨name⟩[,UNIT ⟨dev⟩[,⟨sec addr⟩]],RANDOM ⟨rec len⟩

 ⟨num⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨name⟩ is a ⟨disk file name⟩
 ⟨dev⟩ is a ⟨numeric expression⟩ whose value is from 4-30
   if omitted, the default value is 8 for a CBM disk drive
 ⟨sec addr⟩ is a ⟨numeric expression⟩ whose value is from 0-15
   if omitted, COMAL will supply it
 ⟨rec len⟩ is a positive ⟨numeric expression⟩
```

EXAMPLES

```
OPEN FILE 2,"CUSTOMERS",RANDOM 100
OPEN INFILE,FILE'NAME$,RANDOM REC'SIZE
```

SAMPLE PROGRAM

```
DIM A$ OF 80
OPEN 2,"0:RANDOM'SAMPLE",RANDOM 80
PRINT "SAMPLE RANDOM ACCESS"
REPEAT
  INPUT "WHAT RECORD NUMBER (0 TO STOP): " : NUMBER
  IF NUMBER>0 AND NUMBER < 10 THEN        allow only a few records
    INPUT "READ OR WRITE IT: " : A$
    CASE A$ 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: A$
  PRINT A$
ENDPROC READ'IT
//
PROC WRITE'IT
  PRINT "ENTER UP TO 80 CHARACTERS TO WRITE"
  INPUT ")" : A$
  WRITE FILE 2,NUMBER: A$
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, FILE, OPEN, READ, STATUS, UNIT, WRITE

KEYWORD: READ (from DATA statements)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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).

SYNTAX

```
READ ⟨variable name⟩ {,⟨variable name⟩}
```

EXAMPLES

```
READ A,B
READ SCORE,PLAYER$
```

SAMPLE PROGRAM

```
DIM MONTH$(1:12) OF 3
DATA "JAN","FEB","MAR","APR", "MAY","JUN"
DATA "JUL","AUG","SEP","OCT", "NOV","DEC"
FOR X=1 TO 12 DO READ MONTH$(X)
REPEAT
  INPUT "WHICH MONTH NUMBER (0 TO STOP): " : M;
  CASE M OF
  WHEN 0
    PRINT "STOP"
  WHEN 1,2,3,4,5,6,7,8,9,10,11,12
    PRINT MONTH$(M)
  OTHERWISE
    PRINT "OOOPS."
    PRINT "I DON'T KNOW A MONTH WITH THAT NUMBER"
  ENDCASE
UNTIL M=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, SGN
USED IN PROCEDURE: DISK'GET'INIT
SEE ALSO: APPEND, CLOSE, DATA, EOD, EOF, FILE, OPEN, RANDOM, STATUS, WRITE

KEYWORD: READ (from a sequential file)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [+]VERSION 1.02 [*]

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 1.02 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 2,"0:TABLE'VALUES",READ
READ FILE 2: TABLE
CLOSE 2
```

2) READ FILE is not compatible with INPUT FILE.
3) READ FILE is used to read files created with WRITE FILE statements.
4) See **APPENDIX C** for more information about sequential files.

SYNTAX

```
READ FILE ⟨file number⟩ : ⟨variable⟩ {,⟨variable⟩}

 ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 2-254
```

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$="VISITORFILE"
OPEN INFILE,FILENAME$,READ
READ FILE INFILE : ITEM$
PRINT ITEM$
CLOSE INFILE
PRINT "ALL DONE"
```

```
RUN
   (system opens disk file number 4 named VISITORFILE for input)
COMAL USERS GROUP       varies depending upon contents of VISITORFILE
   (system closes file number 4)
ALL DONE
```

ADDITIONAL SAMPLE SEE: EOF

SEE ALSO: APPEND, CLOSE, DATA, EOD, EOF, FILE, GET$, INPUT, OPEN, RANDOM, STATUS, WRITE

KEYWORD: READ (from random / direct access file)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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⟩.

SYNTAX

```
READ FILE ⟨file number⟩,⟨record number⟩[,⟨offset⟩]: ⟨variable list⟩

 ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 1-255
 ⟨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
 ⟨variable list⟩ is ⟨variable⟩ {,⟨variable⟩}
```

EXAMPLES

```
READ FILE 3,5: NAME$
READ FILE INFILE,REC'NO: ITEM$,PRICE
```

SAMPLE PROGRAM

```
DIM TEXT$ OF 80
OPEN 5,"RANDOM'SAMPLE",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'SAMPLE 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, STATUS, UNIT, WRITE

KEYWORD: READ
CATEGORY: Type of OPEN
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 1.02 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., OPEN 3,"",UNIT 3,READ). The word FILE is optional and if omitted will be supplied by the system.

SYNTAX

```
OPEN [FILE] ⟨num⟩,⟨filename⟩[,UNIT ⟨dev⟩[,⟨sec adr⟩]],READ

 ⟨num⟩ is a ⟨numeric expression⟩ whose value is from 2-254
 ⟨filename⟩ is a ⟨disk or tape file name⟩
 ⟨dev⟩ is a ⟨numeric expression⟩ whose value is 0, 1 or 3-30
   if omitted, the default value is 8 for a CBM disk drive
 ⟨sec adr⟩ is a ⟨numeric expression⟩ whose value is from 0-15;
   if omitted, COMAL will supply it
```

EXAMPLES

```
OPEN FILE 4,"SCORES",READ
OPEN INFILE,FILENAME$,READ
```

SAMPLE PROGRAM

```
DIM ITEM$ OF 40
OPEN 4,"VISITORFILE",READ
READ FILE 4 : ITEM$
PRINT ITEM$
CLOSE
PRINT "ALL DONE"

RUN
   (system opens disk file number 4 named VISITORFILE for input)
COMAL USERS GROUP      varies depending upon contents of VISITORFILE
   (system closes file number 4)
ALL DONE
```

ADDITIONAL SAMPLES SEE: EOF, GET$

SEE ALSO: APPEND, CLOSE, DATA, EOF, FILE, OPEN, RANDOM, STATUS, WRITE

KEYWORD: REF
CATEGORY: Type of parameter
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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. 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 multi-dimension 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.

NOTE

Old version 0.11 may use a procedure as a function, and does not support the keywords FUNC and ENDFUNC. Versions 0.12 and 1.02 are updated to meet the new COMAL definition, which separates the function from the procedure.

SYNTAX

```
PROC ⟨procedure name⟩[ (⟨parameter list⟩) ] [CLOSED]
  or
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

```
PROC SCORES (REF S$)
PROC COMPRESS (REF C$, X) CLOSED
```

SAMPLE PROGRAM

```
A=ORD("A"); Z=ORD("Z")
DIM JUNK$(3) OF 1
FOR X=1 TO 3 DO JUNK$(X)=CHR$(RND(A,Z))          random letter
REPEAT
  INPUT "WHICH JUNK - 1,2, OR 3 (OR 0 TO STOP): " : WHICH
  CASE WHICH OF
  WHEN 1,2,3
    EXEC SEE(WHICH, JUNK$)
  OTHERWISE
    PRINT "NO SUCH JUNK"
  ENDCASE
UNTIL NOT WHICH
PRINT "ALL DONE"
//
PROC SEE(X, REF A$()) CLOSED
  PRINT "JUNK NUMBER"; X; "IS"; A$(X)
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: DISK'GET, DISK'GET'STRING, GET'ALPHA, GET'CHAR, LOWER'TO'UPPER, SCAN
SEE ALSO: CLOSED, ENDFUNC, ENDPROC, EXEC, FUNC, IMPORT, PROC

KEYWORD: REM
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

Allows comments (remarks) to be included in a program listing. Standard COMAL uses // to represent the beginning of a remark, thus CBM COMAL converts **REM** to // for you. Anything on a line after the **REM** (//) will be ignored and execution continued on the next line. COMAL places one space before the // which represents the word **REM**. 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 EDITed to disk or tape. A remark statement is non-executable, but takes up memory. They are useful to relate the reasoning behind a program section or to identify sections of the program.

NOTES

1) The keyword REM is converted to // by the system.
2) An exclamation point is converted to // by the system.
3) The COMAL definition uses only // for a remark statement.
4) Version 1.02 also allows an empty line.

SYNTAX

```
// [⟨anything⟩]

 ⟨anything⟩ is optional and may be any set of printable characters
```

EXAMPLES

```
//
// CLEAR THE SCREEN
// GET THE ANSWER
REM LAST REVISION DATE: 11-23-82
```

SAMPLE PROGRAM

```
// PRINT AND/OR TABLE
//
DIM TYPE$ (FALSE: TRUE) OF 5 // STRING 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 EACH TWO POSSIBILITIES, TRUE OR FALSE
  FOR B=FALSE TO TRUE //    DO EACH TWO POSSIBILITIES, TRUE OR FALSE
    PRINT "A= "; TYPE$ (A) , "B= "; TYPE$ (B) , // PRINT 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)
  NEXT B //                                                DO THE NEXT B
NEXT 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

KEYWORD: RENUM
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 1.02 will print an error message. Version 0.12 will renumber it with some lines numbered over the maximum 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 1.02 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 [ [⟨start source line⟩; ] [⟨target start line⟩] [,⟨target increment⟩]

 ⟨start source line⟩ is a number from 1-9999
   only lines from this number to end of program are renumbered
   if omitted, all lines are renumbered
 ⟨target start line⟩ 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, ...

SAMPLE EXERCISE

```
10 PRINT "FIRST"
20 PRINT "SECOND"
30 PRINT "THIRD"

RENUM 9000

LIST
9000 PRINT "FIRST"
9010 PRINT "SECOND"
9020 PRINT "THIRD"
```

SEE ALSO: AUTO, DEL, EDIT, LIST

KEYWORD: REPEAT
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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
USED IN PROCEDURES: FETCH, GET'ALPHA, GET'CHAR, GET'VALID, SHIFT
SEE ALSO: UNTIL

KEYWORD: RESTORE
CATEGORY: Command / Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 0.12 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 a ⟨numeric variable name⟩
```

EXAMPLES

```
RESTORE
RESTORE MONTHS                          version 1.02 only
```

SAMPLE

```
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

KEYWORD: RETURN
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 and return 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. OLD version 0.11 does not support the keyword **RETURN**. Instead, it assigns the function value to the function name. The new COMAL definition does not allow this, using the **RETURN** statement instead. The RETURN statement can be part of the one line IF structure.

NOTES

1) RETURN is not supported by version 0.11.
2) String functions are not supported by versions 1.02 or 0.12.

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 here
UNTIL NUMBER=0
PRINT "ALL DONE"
//
FUNC EVEN(N)
   IF N 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 SAMPLE SEE: FUNC

SEE ALSO: CLOSED, FUNC, PROC

KEYWORD: RND
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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).

SYNTAX (between 0 and 1)

```
RND(⟨parameter⟩)

  ⟨parameter⟩ is a ⟨numeric expression⟩ used as the "seed"
    use 0 for a random number OR
    use -1 to seed a pseudo random sequence, using 1 thereafter

        (integer within the range)

RND(⟨start number⟩,⟨end number⟩)

  ⟨start number⟩ and ⟨end number⟩ are ⟨numeric expessions⟩
    the integer returned will be between them (inclusive)
```

EXAMPLES

```
RND(0)
RND(1,6)
RND(1,52)
```

SAMPLE PROGRAM

```
DIM R$ OF 1
PRINT "ROLL TWO DICE UNTIL YOU ROLL A SEVEN"
PRINT "-----------"
REPEAT
  INPUT "HIT [RETURN] TO ROLL THE DICE" : R$
  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, REF, REPEAT, THEN, WHILE

SEE ALSO: INT

KEYWORD: RUN
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

The program currently in the computer is executed beginning with the first line. All variables are cleared prior to execution.

SPECIAL NOTE FOR OLD VERSION 0.11:
While in version 0.11 SPLIT mode the program is automatically prepassed (pass 2) and stored on disk, under the file name you specify when it asks for a name. Then you are allowed the option of automatically switching to the COMAL-80 EXECUTE module to **RUN** the program, or to return to interactive mode in the COMAL-80 INPUT module (this is how to save a prepassed version of a program to disk). You may wish to end the file name of any program stored on disk via a **RUN** in version 0.11 SPLIT mode with a .P so it can be distinguished from programs not yet prepassed.

NOTES

1) Programs stored on disk with a RUN while in version 0.11 SPLIT mode may later be retrieved with a LOAD or CHAIN command, but not with an ENTER command.
2) The SPLIT mode of COMAL is no longer supported, but is only available in the obsolete version 0.11.

SYNTAX

```
RUN
```

EXAMPLE

```
RUN
```

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, OBJLOAD, SAVE, VERIFY

KEYWORD: SAVE
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

Stores the program currently in the computer onto disk or tape in compressed form (remark statements are not deleted). Later, the program can be retrieved via a LOAD or CHAIN command. OLD version 0.11 SPLIT MODE won't execute a program from disk that was stored via the **SAVE** command. Rather it must be prepassed prior to storing on disk by using the RUN command while under the SPLIT MODE INPUT module (see RUN).

NOTES

1) **SAVE** to tape is not supported by version 0.12.
2) The compressed forms of the programs as **SAVE**d will not be compatible between version 0.12 and 1.02. To "transfer" a program to another version, first LIST it to disk, then ENTER it under the other version (see ENTER).

SYNTAX

```
SAVE ⟨program name⟩[,⟨unit⟩]

 ⟨program name⟩ is a ⟨disk or tape file name⟩
 ⟨unit⟩ is a ⟨numeric expression⟩ whose value is 1 or 4-30;
   if omitted, the default value is 8 for a CBM disk drive
```

EXAMPLES

```
SAVE "TEMP1"
SAVE "SPECIAL",9
```

SAMPLE EXERCISE

```
10 PRINT "THIS IS ONLY A SAMPLE"

CAT 1
1"SAMPLE DISK "          SD 2A
3    "LOADER"              PRG
661 BLOCKS FREE.

SAVE "SAMPLE'ONLY"
CAT 1
1"SAMPLE DISK "          SD 2A
3    "LOADER"              PRG
1    "SAMPLE'ONLY"         PRG
660 BLOCKS FREE.
```

SEE ALSO: LOAD, RUN, STATUS, UNIT, VERIFY

KEYWORD: SELECT
CATEGORY: Command / Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

Allows you to select (choose) the output location. Recognized locations include "DS" for Data Screen and "LP" for Line Printer. 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 cancels any previous **SELECT** "LP"). The word OUTPUT is optional and will be supplied by COMAL if omitted.

NOTES

1) INPUT prompts and error messages are directed to the screen even if a SELECT OUTPUT "LP" has been issued.
2) The device number of the printer can be specified as part of the location string, "LPx", where x is replaced by the printer device number. If a device is not specified, 4 is used, matching the normal CBM printer device number. This option is not available in version 0.12.

SYNTAX

```
SELECT [OUTPUT] ⟨type⟩
 ⟨type⟩ is a ⟨string expression⟩ representing the output location
   recognized locations are: "DS" for Data Screen
                             "LP" for Line Printer
                             "LPx" for Line Printer
                                x is a digit from 4-9 which
                                specifies the printer device number
```

EXAMPLES

```
SELECT OUTPUT "DS"
SELECT O$
SELECT "DS"
```

SAMPLE PROGRAM

```
DIM NAME$ OF 20, C$ OF 1
DATA "JIM","SUE","FRED","RHIANON", "CHARLY","STEVE"
PRINT "REPLY 'Y' TO PRINT NAME ON PRINTER"
REPEAT
   READ NAME$
   INPUT NAME$ : C$
   IF C$ = "Y" THEN
      SELECT "LP"                    output to printer
      PRINT NAME$
   ENDIF
   SELECT "DS"                       output to screen
UNTIL EOD
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, OUTPUT
USED IN PROCEDURES: BOLD'CHAR, DOUBLE'CHAR, MULTI'CHAR
SEE ALSO: OUTPUT, PRINT

KEYWORD: SETEXEC
CATEGORY: Command
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

Allows the choice of whether or not the keyword EXEC will be listed in a program listing. The system initially suppresses the word EXEC.

NOTES

1) This feature is not supported by version 0.11.
2) **SETEXEC** must be issued as a direct command. It cannot be part of a program.
3) The initial system default is **SETEXEC-**.
4) See **APPENDIX A** for a description of the procedure structure.

SYNTAX

```
SETEXEC⟨type⟩
 ⟨type⟩ is either + or -
   + means that the keyword EXEC should be listed
   - means that the keyword EXEC should not be listed
```

EXAMPLES

```
SETEXEC+          list any EXEC keyword in the program
SETEXEC -         do not list any EXEC keyword in the program
```

SAMPLE EXERCISE

```
10 EXEC MISTAKE("SILLY")
20 EXEC MISTAKE("COMPLICATED")
30 PRINT "ALL DONE"
40 //
50 PROC MISTAKE(TYPE$)
60 PRINT TYPE$; "MISTAKE"
70 ENDPROC MISTAKE

SETEXEC-                        set not to list the keyword EXEC
LIST
0010 MISTAKE("SILLY")
0020 MISTAKE("COMPLICATED")
0030 PRINT "ALL DONE"
0040 //
0050 PROC MISTAKE(TYPE$)
0060    PRINT TYPE$; "MISTAKE"
0070 ENDPROC MISTAKE
```

```
RUN
SILLY MISTAKE
COMPLICATED MISTAKE
ALL DONE
```

SEE ALSO: EXEC, SETMSG, TRAP

KEYWORD: SETMSG
CATEGORY: Command
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [-]

Allows the choice of whether or not the error message will be printed during program entry. This option is only available in version 0.12, which uses a disk file for its error messages. With the messages turned off, you will simply receive an error number, bypassing the delay involved with the error messages from disk. In either case, the COMAL system will position the cursor on the faulty line at the suspected location of the syntax problem.

NOTES

1) This feature is supported only by version 0.12.
2) **SETMSG** must be issued as a direct command. It cannot be part of a program.
3) The initial system default is **SETMSG+**.

SYNTAX

```
SETMSG⟨type⟩
 ⟨type⟩ is either + or -
   + means that the error messages should be printed
   - means that the error messages should not be printed
```

EXAMPLES

```
SETMSG+          print any error messages during program entry
SETMSG-          do not print the program entry error messages
```

SAMPLE EXERCISE

```
SETMSG+

10 OPEN FILE 2                          each line 10 printed here
", " EXPECTED                           is actually the same line
10 OPEN FILE 2,                         on the screen
EXPRESSION EXPECTED
10 OPEN FILE 2, "TEST'DATA"
", "EXPECTED
10 OPEN FILE 2, "TEST'DATA",
SYNTAX ERROR
10 OPEN FILE 2, "TEST'DATA", READ

SETMSG-
20 OPEN FILE 3                          each line 20 printed here
ERROR 56                                is actually the same line
20 OPEN FILE 3,                         on the screen
ERROR 58
20 OPEN FILE 3, "NEW'DATA"
ERROR 56
20 OPEN FILE 3, "NEW'DATA",
ERROR 1
20 OPEN FILE 3, "NEW'DATA", WRITE
```

SEE ALSO: SETEXEC

KEYWORD: SGN
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

SYNTAX

```
SGN(⟨numeric expression⟩)
```

EXAMPLES

```
SGN(-32)
SGN(NUMBER)
```

SAMPLE PROGRAM

```
DIM SIGN$(-1:1) OF 10
DATA "NEGATIVE","ZERO","POSITIVE"
FOR X=-1 TO 1 DO READ SIGN$(X)
REPEAT
  INPUT "NUMBER (0 TO STOP): " : NUMBER
  PRINT "THE SGN RETURNED IS: ";SGN(NUMBER); SIGN$(SGN(NUMBER))
UNTIL NUMBER=0
PRINT "ALL DONE"

RUN
NUMBER (0 TO STOP): -43
THE SGN RETURNED IS: -1 NEGATIVE
NUMBER (0 TO STOP): 235
THE SGN RETURNED IS: 1 POSITIVE
NUMBER (0 TO STOP): -5026.41
THE SGN RETURNED IS: -1 NEGATIVE
NUMBER (0 TO STOP): 0
THE SGN RETURNED IS: 0 ZERO
ALL DONE
```

SEE ALSO: ABS, VAL

KEYWORD: SIN
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Returns the sine of the specific 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 (X)
SIN(22)
```

SAMPLE EXERCISE

```
REPEAT
  INPUT "ENTER AN ANGLE IN RADIANS (0 TO STOP) : " : X
  IF X THEN PRINT "THE SINE OF";X;"IS";SIN(X)
UNTIL X=0
PRINT "ALL DONE"

RUN
ENTER AN ANGLE IN RADIANS (0 TO STOP) : 5
THE SINE OF 5 IS -.958924274
ENTER AN ANGLE IN RADIANS (0 TO STOP) : 25
THE SINE OF 25 IS -.132351746
ENTER AN ANGLE IN RADIANS (0 TO STOP) : 0
ALL DONE
```

SEE ALSO: ATN, COS, TAN

KEYWORK: SIZE
CATEGORY: Command
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

Prints the amount of free memory (in bytes) left in the system. Bytes used for the program and its variables are not considered free.

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.

SYNTAX

```
SIZE
```

EXAMPLE

```
SIZE
```

SAMPLE EXERCISE

```
SIZE                          direct - not as part of program

3569 BYTES FREE.
```

SEE ALSO: DIM

KEYWORD: SPC$
CATEGORY: Function
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

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 0.12.
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, TAB, ZONE

KEYWORD: SQR
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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(X)
```

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 the 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 "THE SQUARE ROOT OF THEIR AVERAGE 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
THE SQUARE ROOT OF THEIR AVERAGE IS 3.87298335
ALL DONE
```

SEE ALSO: EXP

KEYWORD: STATUS
CATEGORY: Command / Function
COMAL STANDARD: [NO] VERSION 0.12 [+] VERSION 1.02 [*]

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 PET/CBM BASIC. Version 1.02 automatically checks the disk status after LOAD, SAVE, LIST, ENTER, CAT, CHAIN, and VERIFY, 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 (except in old version 0.11).
2) STATUS$ may be used as a function (except in old version 0.11). It may be used in an assignment statement or may be compared to another string.
3) The CBM Floppy Disk User Manual includes a list of possible error messages.

SYNTAX (disk status)

```
STATUS                converts to PRINT STATUS$ in version 1.02

 or

STATUS$                                    not old version 0.11

       (disk file status)

 STATUS (⟨file number⟩)

   ⟨file number⟩ is a ⟨numeric expression⟩ evaluating to 2 - 254
```

EXAMPLES

```
STATUS
STATUS(INFILE)
```

SAMPLE PROGRAM

```
PRINT "THE CURRENT DISK STATUS IS";
STATUS
OPEN 6,"0:WRONGNAME",READ
PRINT "THE STATUS OF FILE 6 IS";
PRINT STATUS(1)
PRINT "THE DISK STATUS NOW IS"
STATUS
CLOSE
PRINT "ALL DONE"
```

```
RUN
THE CURRENT DISK STATUS IS 00, OK, 00, 00
THE STATUS OF FILE 6 IS 0
THE DISK STATUS NOW IS
62, FILE NOT FOUND, 00, 00
ALL DONE
```

ADDITIONAL SAMPLES SEE: DELETE, OBJLOAD
SEE ALSO: APPEND, CLOSE, EOF, FILE, INPUT, LOAD, OPEN, PRINT
RANDOM, READ, SAVE, VERIFY, WRITE

KEYWORD: STEP
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 ⟨controlvariable⟩=⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] DO ⟨statement⟩

          (Multi-line)

FOR ⟨controlvariable⟩=⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] [DO]

 ⟨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
FOR CARD=52 TO 1 STEP -1
```

SAMPLE PROGRAM

```
REPEAT
  INPUT "WHAT NUMBER TO COUNT BY (0 TO STOP): " : NUMB
  IF NUMB THEN
    FOR COUNT= NUMB TO NUMB*6 STEP NUMB
      PRINT COUNT;
    NEXT 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, NEXT, TO

KEYWORD: STOP
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 (except in old version 0.11 SPLIT mode). Variables remain intact and may be displayed and changed before continuing. This message is displayed when a **STOP** is encountered (0030 represents the line number where the **STOP** was encountered):

STOP AT 0030

NOTE

⟨message⟩ is not supported by version 0.12.

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
USED IN PROCEDURES: BOLD'CHAR, DOUBLE'CHAR, MULTI'CHAR
SEE ALSO: CHAIN, CON, END, RUN

KEYWORD: STR$
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [-] VERSION 1.02 [*]

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

STR$ is not supported by version 0.12.

SYNTAX

```
STR$(⟨number⟩)

 ⟨number⟩ is ⟨numeric expression⟩
```

EXAMPLES

```
STR$(-3.58)              converts it to "-3.58"
STR$(2+5)                converts it to "7"
SDR$(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, VAL

KEYWORD: SYS
CATEGORY: Statement/Command
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

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 among 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

RUN
   (screen clears and the computer resets
     back to BASIC mode of operation)
```

USED IN PROCEDURES: BASIC'RESET, DISK'GET
SEE ALSO: EXEC, INTERRUPT, OBJLOAD, POKE

KEYWORD: TAB
CATEGORY: System function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 non-positive column number is specified.

NOTE

TAB prints spaces up to the specified column, not cursor rights as in CBM/PET 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

KEYWORD: TAN
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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)
degrees = radians * 57.2957795

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

SYNTAX

```
TAN (⟨numeric expression⟩)
```

EXAMPLES

```
TAN(24)
TAN(NUM)
```

SAMPLE PROGRAM

```
REPEAT
   INPUT "ENTER AN ANGLE IN RADIANS (0 TO STOP): " : A
   IF A THEN PRINT "THE TANGENT OF"; A; "IS"; TAN(A)
UNTIL A=0
PRINT "ALL DONE"

RUN
ENTER AN ANGLE IN RADIANS (0 TO STOP): 5
THE TANGENT OF 5 IS -3.380515
ENTER AN ANGLE IN RADIANS (0 TO STOP): 25
THE TANGENT OF 25 IS -.133526403
ENTER AN ANGLE IN RADIANS (0 TO STOP): 0
ALL DONE
```

SEE ALSO: ATN, COS, SIN

KEYWORD: THEN
CATEGORY: Special
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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.

SYNTAX (multi-line structure)

```
IF ⟨condition⟩ [THEN]
 ⟨statements⟩

          (single line IF)

IF ⟨condition⟩ THEN ⟨statement⟩

 ⟨condition⟩ is a ⟨numeric expression⟩
```

EXAMPLE

```
IF GUESS=NUM THEN PRINT "WINNER"
```

SAMPLE PROGRAM

```
DIM C$ 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 "": C$;            the null prompt avoids a "?" prompt
  IF C$ IN "+-" THEN
    PRINT NUMB2;"=";
    IF C$="+" THEN
      PRINT NUMB1+NUMB2
    ELSE
      PRINT NUMB1-NUMB2
    ENDIF
  ELIF C$⟨⟩"X" THEN
    PRINT "ENTER + OR -, OR X TO EXIT"
  ENDIF
UNTIL C$="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, MOD, OBJLOAD, OUTPUT, PEEK, RANDOM
USED IN PROCEDURES: BOLD'CHAR, BOLD'FACE, FETCH, LOWER'TO'UPPER, MULTI'STRIKE, SCAN
SEE ALSO: ELIF, ELSE, ENDIF, IF

KEYWORD: TIME
CATEGORY: Function/Statement
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

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 0.12. 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

```
PRINT "[CLR]"                          clear the screen
REPEAT
  CURSOR 12,35
  A=TIME DIV 60
  PRINT A; "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

KEYWORD: TO
CATEGORY: Special
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 or EDIT is going to a file.

NOTE

Version 0.12 does not support the use of TO with LIST or EDIT.

SYNTAX

```
FOR ⟨controlvariable⟩=⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] DO ⟨statement⟩
  or
FOR ⟨controlvariable⟩=⟨start⟩ TO ⟨end⟩[STEP ⟨step⟩] [DO]
  or
LIST [⟨range⟩] TO ⟨filename⟩[,⟨device⟩]
  or
EDIT [⟨range⟩] TO ⟨filename⟩[,⟨device⟩]

 ⟨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 X=1 TO 40 DO PRINT " ",
FOR TEMP=COUNT TO MAX STEP 5
LIST TO "0: MY PROGRAM.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
NEXT TEMP
PRINT "TOTAL WAS";TOTAL; "FOR AN AVERAGE OF"; TOTAL/NUMBER'SCORES

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

ADDITIONAL SAMPLES SEE: EXEC, OR, ORD, PEEK, READ, REM
USED IN PROCEDURES: BOLD'CHAR, CLEAR'FROM, CLEAR'LINE, CURSOR, DISK'GET'INIT, DISK'GET'STRING, LOWER'TO'UPPER
SEE ALSO: DO, ENDFOR, FOR, NEXT, STEP

KEYWORD: TRAP
CATEGORY: Statement
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

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.

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) 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).

SYNTAX

```
TRAP ESC ⟨type⟩

⟨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

SAMPLE

```
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: ESC, KEY$

SEE ALSO: ESC, SETEXEC

KEYWORD: TRUE
CATEGORY: System constant
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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, REM, RETURN
USED IN PROCEDURE: FETCH
SEE ALSO: FALSE

KEYWORD: UNIT
CATEGORY: Part of OPEN statement
COMAL STANDARD: [NO] VERSION 0.12 [*] VERSION 1.02 [*]

Specifies the device number to be used for the file in the OPEN statement. It is optional and, when omitted, defaults to 8 (this matches the default device number Commodore assigns to their disk drives).

CBM ASSIGNED UNITS

0 = Keyboard
1 = Tape unit 1
2 = Tape unit 2 (not supported)
future versions may use
unit 2 for an RS-232 port
3 = Screen
4 = Printer
5 = Alternate printer
6-7 = Unassigned
8 = Disk drive
9 = Alternate disk drive
10-30 = Unassigned

CBM SECONDARY ADDRESSES WITH TAPE FILES

0 = OPEN FOR A READ
1 = OPEN FOR A WRITE with an End Of File (EOF) mark
2 = OPEN FOR A WRITE with an End Of Tape (EOT) mark

SYNTAX

```
OPEN [FILE] ⟨filenumb⟩,⟨filename⟩[,UNIT ⟨dev⟩[,⟨sec adr⟩]] [,⟨type⟩]

⟨filenumb⟩ is a ⟨numeric expression⟩ whose value is from 2-254
⟨filename⟩ is a ⟨disk or tape file name⟩
⟨dev⟩ is a ⟨numeric expression⟩ whose value is 0, 1, or 3-30
  if omitted, the default value is 8 for a CBM disk drive
⟨sec adr⟩ is a ⟨numeric expression⟩ whose value is from 0-15;
  if omitted, COMAL will assign it
⟨type⟩ is either READ, or WRITE, or APPEND, or RANDOM ⟨record len⟩
  ⟨record len⟩ is a positive ⟨numeric expression⟩
```

EXAMPLES

```
OPEN 2,"TEST'DATA",UNIT 1,2,READ
OPEN 4,"",UNIT 4,7,WRITE
```

SAMPLE PROGRAM

```
DIM TEXT$ OF 80
OPEN 3, "", UNIT 3, READ
PRINT "[CLR]",                          clear screen
PRINT "TESTING 1, 2, 3"
PRINT "NOW WE WILL READ THE TOP LINE RIGHT OFF THE SCREEN"
PRINT "[HOME]",                         home cursor
INPUT FILE 3: TEXT$
PRINT
PRINT "THE TOP LINE READS: "
PRINT TEXT$
CLOSE
PRINT "ALL DONE"

RUN
   (file number 3 is opened to the screen)
   (the screen is cleared)
TESTING 1, 2, 3
NOW WE WILL READ THE TOP LINE RIGHT OFF THE SCREEN
   (the cursor is put in the home position)
   (the next line appears just beneath the previous one)
THE TOP LINE READS:
TESTING 1, 2, 3
   (file 3 is closed)
ALL DONE
```

ADDITIONAL SAMPLE SEE: GET$
USED IN PROCEDURE: DISK' COMMAND
SEE ALSO: CLOSE, ENTER, LOAD, OBJLOAD, OPEN, SAVE, VERIFY

KEYWORD: UNTIL
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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⟩
```

EXAMPLES

```
UNTIL EOD
UNTIL I$="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
USED IN PROCEDURES: FETCH, GET'ALPHA, GET'CHAR, GET'VALID, SHIFT
SEE ALSO: REPEAT

KEYWORD: USING
CATEGORY: Special
COMAL STANDARD: [YES] VERSION 0.12 [-] VERSION 1.02 [*]

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 0.12

SYNTAX

```
PRINT [AT ⟨row⟩, ⟨col⟩: ] USING ⟨format string⟩ : ⟨variable list⟩

 ⟨row⟩ is a ⟨numeric expression⟩ whose value is 1-25
 ⟨col⟩ is a ⟨numeric expression⟩ whose value is 1-80
 ⟨format string⟩ is a ⟨string expression⟩
   # reserves a digit place
   . specifies the location of the decimal point
   - floating minus sign is an option
```

EXAMPLES

```
PRINT USING "$###.##" : COST
PRINT USING MONEY$ : PRICE
```

SAMPLE PROGRAM

```
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
```

SEE ALSO: AT, PRINT, TAB, ZONE

KEYWORD: VAL
CATEGORY: Function
COMAL STANDARD: [YES] VERSION 0.12 [-] VERSION 1.02 [*]

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 non-valid 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 0.12.
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 exponent notation (unshifted E).

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
```

SAMPLE

```
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): 155 155
NUMBER (0 TO STOP): +123 123
NUMBER (0 TO STOP): -123 -123
NUMBER (0 TO STOP): 123ABC 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: FUNCTION ARGUMENT ERROR
```

ADDITIONAL SAMPLE SEE: OBJLOAD

SEE ALSO: CHR$, INT, ORD, STR$

KEYWORD: VERIFY
CATEGORY: Command
COMAL STANDARD: [NO] VERSION 0.12 [-] VERSION 1.02 [*]

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 0.12.
2) **VERIFY** cannot be used in conjunction with LISTing or EDITing files to tape or disk.

SYNTAX

```
VERIFY ⟨filename⟩[,⟨unit⟩]

 ⟨filename⟩ is a ⟨disk file name⟩
 ⟨unit⟩ is a ⟨numeric expression⟩ whose value is 1 or 4-30;
   if omitted, the default value is 8 for a CBM disk drive
```

EXAMPLES

```
VERIFY "TEST1"
VERIFY NAME$,9
```

SAMPLE EXERCISE

```
10 // MINE

SAVE "MY'PROGRAM"

VERIFY "MY'PROGRAM"          a 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, UNIT

KEYWORD: WHEN
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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. See **APPENDIX A** for a description of the CASE structure.

SYNTAX

```
WHEN ⟨list of values⟩
  ⟨statements⟩

 ⟨list of values⟩ is a series of either ⟨numeric expressions⟩
    or ⟨string expressions⟩
   if more than one is used, a comma is placed between them
   each must be of the same type as the CASE ⟨control expression⟩
```

EXAMPLE 1

```
WHEN "H", "HELP", "?"
  EXEC INSTRUCTIONS
```

EXAMPLE 2

```
WHEN 5, 6, 7
  PRINT "YOU WIN"
```

SAMPLE

```
DIM CHOICE$ OF 1
REPEAT
  INPUT "ENTER A VOWEL (X TO EXIT) : " : CHOICE$;
  CASE CHOICE$ OF
  WHEN "A", "E", "I", "O", "U"
    PRINT "RIGHT"
  WHEN "X"
    PRINT "EXIT"
  OTHERWISE
    PRINT "NOPE"
  ENDCASE
UNTIL 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, REF
USED IN PROCEDURE: FETCH
SEE ALSO: CASE, ENDCASE, OF, OTHERWISE

KEYWORD: WHILE
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 ⟨comparison⟩ DO ⟨statement⟩

      (multi-line)

WHILE ⟨comparison⟩ [DO]
  ⟨statements⟩

  ⟨comparison⟩ is an ⟨expression⟩
```

EXAMPLES

```
WHILE A$ = "" DO EXEC GETPROC
WHILE OK DO
WHILE NOT EOF(2) DO
```

SAMPLE PROGRAM

```
RIGHT = 0; ERRORS = 0; MAXERRORS = 1          initialize
WHILE ERRORS( MAXERRORS DO
  NUMB1 = RND(1,98) ; NUMB2 = RND(NUMB1,99) numb1 is larger than numb2
  PRINT "WHAT IS"; NUMB2; "MINUS"; NUMB1;
  INPUT ": " : ANSWER;
  IF ANSWER = NUMB2-NUMB1 THEN
    RIGHT: +1                                 increment right count
    PRINT "YES"
  ELSE
    PRINT "NO"
    ERRORS: +1
  ENDIF
ENDWHILE
PRINT "OOPS -"; ERRORS; "ERROR(S) AFTER"; RIGHT; "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

KEYWORD: WRITE (to a sequential file)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 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, BACKUPed (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.

NOTES

1) **WRITE** FILE and PRINT FILE are not compatible.
2) A file created by **WRITE** FILE must be read with READ FILE statements (INPUT FILE cannot be used).
3) See **APPENDIX C** for more information about sequential files.
4) In version 1.02, the value of each element of an entire array may be written to a file using only one statement. Just use the array name without the parentheses section:

```
DIM TABLE(3,3)
FOR X=1 TO 3
  FOR Y=1 TO 3
    TABLE(X,Y)=X*Y
  NEXT Y
NEXT X
OPEN 3,"0:TABLE'VALUES",WRITE
WRITE FILE 3: TABLE
CLOSE 3
```

SYNTAX

```
WRITE FILE ⟨file number⟩ : ⟨variable list⟩

  ⟨file number⟩ is a ⟨numeric expression⟩ whose value is from 2-254
  ⟨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 2, "WINNERS3", WRITE
PRINT "WE WILL NOW WRITE 3 WINNER NAMES TO DISK"
FOR X=1 TO 3 DO
   INPUT "WINNERS NAME: " : NAME$;
   WRITE FILE 2 : NAME$
   PRINT "*"
NEXT X
CLOSE 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 - the file now is closed)
ALL DONE
```

ADDITIONAL SAMPLE SEE: CLOSE
USED IN PROCEDURE: DISK' COMMAND
SEE ALSO: CLOSE, FILE, OPEN, PRINT, READ, STATUS, UNIT

KEYWORD: WRITE (to a random/direct access file)
CATEGORY: Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 a sequential file. If you are new to programming, you may wish to master sequential file manipulation first.

SYNTAX

```
WRITE FILE ⟨filenumber⟩,⟨record number⟩[,⟨offset⟩]: ⟨variable list⟩

  ⟨filenumber⟩ is a ⟨numeric expression⟩ whose value is from 2-254
  ⟨record number⟩ is a positive ⟨numeric expression⟩
    it must evaluate to a valid record number
  ⟨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

```
DIM TEXT$ OF 80
OPEN 7,"RANDOM'SAMPLE",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'SAMPLE 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
```

ADDITIONAL SAMPLE SEE: RANDOM
USED IN PROCEDURE: DISK' COMMAND
SEE ALSO: CLOSE, FILE, OPEN, PRINT, READ, STATUS, UNIT

KEYWORD: WRITE
CATEGORY: Type of an OPEN
COMAL STANDARD: [YES] VERSION 0.12 [+] VERSION 1.02[*]

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 0.12.
2) You may open a file to a printer by specifying its device number as UNIT ⟨device⟩ (i.e., UNIT 4). 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] ⟨num⟩,⟨name⟩[,UNIT ⟨dev⟩[,⟨secondry adr⟩]],WRITE

 ⟨num⟩ is a ⟨numeric expression⟩ whose value is from 1-255
 ⟨name⟩ is a ⟨disk or tape file name⟩
 ⟨dev⟩ is a ⟨numeric expression⟩ whose value is from 0-30;
   if omitted, the default value is 8 for a CBM disk drive
 ⟨secondry adr⟩ is a ⟨numeric expression⟩ whose value is from 0-15;
   if omitted, COMAL will supply it
```

EXAMPLES

```
OPEN FILE 2,"0:TESTFILE",WRITE
OPEN INFILE,FILENAME$,WRITE
```

SAMPLE

```
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 2, "0: ACCOUNT LIST", WRITE
WHILE NOT EOD DO
  READ ACCOUNT$
  COUNT: +1
  WRITE FILE 2 : ACCOUNT$
  PRINT "ACCOUNT NUMBER"; COUNT; "-"; ACCOUNT$
  DATA "OFFICE SUPPLIES", "POSTAGE AND SHIPPING"
  DATA "PROFESSIONAL FEES", "ADVERTISING"
ENDWHILE
CLOSE 2
PRINT "ALL DONE"

RUN
WE WILL NOW WRITE AN ACCOUNT NAME FILE
IT COULD BE USED LATER TO EXPAND CODED ACCOUNTS
  (system opens sequential disk file number 2 named ACCOUNT LIST)
  (system writes OFFICE SUPPLIES to disk file number 2)
ACCOUNT NUMBER 1 - OFFICE SUPPLIES
  (system writes POSTAGE AND SHIPPING to disk file number 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, STATUS, UNIT

KEYWORD: ZONE
CATEGORY: Function/Statement
COMAL STANDARD: [YES] VERSION 0.12 [*] VERSION 1.02 [*]

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 while SELECT "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.

NOTE

OLD version 0.11 implemented **ZONE** as a system variable instead of a function. Thus to assign the **ZONE** it used **ZONE** = ⟨value⟩ instead of the new method of **ZONE** ⟨value⟩.

SYNTAX

```
ZONE                        used as a function
  or
ZONE ⟨tab interval⟩         used as a statement

  ⟨tab interval⟩ is a non-negative ⟨numeric expession⟩
    if omitted ZONE will be used as a function and return the
     current zone value.
```

EXAMPLES

```
ZONE 5
OLDZONE = ZONE
```

SAMPLE PROGRAM

```
ZONE 4
STARTZONE = ZONE                    remember what the current ZONE setting is
PRINT "THE CURRENT ZONE SETTING IS"; STARTZONE
PRINT "NOW WE WILL SHOW YOU SOME INCREASING ZONE SETTINGS"
FOR TEMP = 2 TO 9
  ZONE TEMP                          old version 0.11 use ZONE = TEMP
  PRINT TEMP, "B", "C", "D"
NEXT TEMP
ZONE STARTZONE                      return zone to what it was at the start
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

```
ZONE 11                                 old version 0.11 use 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, REM
USED IN PROCEDURES: BOLD'CHAR, CURSOR, DOUBLE'CHAR, FETCH, MULTI'CHAR, SET'PITCH, TAKE'IN
SEE ALSO: INPUT, PRINT, TAB, USING

APPENDIX A

THE COMAL STRUCTURES

The power of COMAL lies in its multi-line 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/NEXT
LOOP ... EXIT / EXITIF ... ENDLOOP
```

MODULES:

```
FUNC ... REF ... CLOSED ... IMPORT ... RETURN ... ENDFUNC
PROC ... REF ... CLOSED ... IMPORT ... RETURN ... ENDPROC
EXEC
```

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 ... ELIF ... ENDIF
d) IF ... THEN ... ELSE ... 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=0 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=0 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	EXAMPLE

IF ⟨condition⟩ THEN
  ⟨true statements set⟩
ELSE
  ⟨false statements set⟩
ENDIF
	IF SCORE⟩ HIGH THEN
	  HIGH=SCORE
	  EXEC NEW'HIGH
	ELSE
	  EXEC REPORT
	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 condition 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	EXAMPLE

IF ⟨condition-1⟩ THEN
  ⟨statements-1⟩
ELIF ⟨condition-2⟩ THEN
  ⟨statements-2⟩
{ELIF ⟨condition-X⟩ THEN
  ⟨statements-X⟩}
ENDIF
	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	EXAMPLE

IF ⟨condition-1⟩ THEN
  ⟨statements-1⟩
ELIF ⟨condition-2⟩ THEN
  ⟨statements-2⟩
{ELIF ⟨condition-X⟩ THEN
  ⟨statements-X⟩}
ELSE
  ⟨statements-else⟩
ENDIF
	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 type 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 conditional execution expression list. After the CASE expression has been evaluated, it is compared with each of the expressions 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	EXAMPLE

CASE ⟨expression⟩ OF
WHEN ⟨expression list⟩
  ⟨statements⟩
{WHEN ⟨expression list⟩
  ⟨statements⟩}
[OTHERWISE
  ⟨statements⟩]
ENDCASE
	CASE LABEL'LINE OF
	WHEN 0
	  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	EXAMPLE

REPEAT
  ⟨statements⟩
UNTIL ⟨condition⟩
	REPEAT
	  EXEC ANOTHER
	  EXEC SORT
	UNTIL ERRORS⟩0

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	EXAMPLE

WHILE ⟨condition⟩ DO
  ⟨statements⟩
ENDWHILE
	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 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 NEXT or 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 "*",
NEXT X
```

NOTE: CBM COMAL (except version 0.11) will automatically convert all NEXT statements into ENDFOR statements, which are more compatible with other loop-terminating keywords. It appears that STANDARD COMAL may soon allow ENDFOR as the terminating statement for a FOR loop. The keyword NEXT is still allowed, but will appear in listings as ENDFOR.

STRUCTURE FORM

FOR ⟨control variable⟩=⟨start⟩ TO ⟨end⟩[STEP ⟨step⟩] DO
  ⟨statements⟩
NEXT [⟨control variable⟩] or ENDFOR [⟨control variable⟩]

EXAMPLE

```
FOR TEMP=1 TO MAX STEP 5 DO
  EXEC ASK
  EXEC COMPARE
NEXT TEMP or ENDFOR TEMP
```

A simple one line version of the FOR structure is also available. It must not use the keyword NEXT or ENDFOR.

STRUCTURE FORM

FOR ⟨control variable⟩=⟨start⟩ TO ⟨end⟩ [STEP ⟨step⟩] DO ⟨statement⟩

EXAMPLE

```
FOR X=1 TO 40 DO PRINT "-",
```

LOOP STRUCTURE

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 EXITIF 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. The LOOP structure was a last minute addition to version 1.02 as this *Handbook* was going to press. The loop exit method was not finalized yet, but will be either an EXIT or EXITIF statement (but not both). A loop structure should have only one exit or it is not part of structured programming. If there are no statements prior the 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. This structure will rarely be used, and should never have more than one EXIT statement.

NOTE

The LOOP ... ENDLOOP structure is not supported by version 0.12.

STRUCTURE FORM

```
LOOP
  ⟨statements⟩
EXITIF ⟨condition⟩        or        IF ⟨condition⟩ THEN EXIT
  ⟨statements⟩
ENDLOOP
```

EXAMPLE

```
LOOP
  PRINT "THIS IS A SILLY LOOP"
  READ FILE 2: TEMP$
EXITIF TEMP$="*END*"        or        IF TEMP$="*END*" THEN EXIT
  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 1.02. Versions 0.12 and 1.02 are updated to meet the new COMAL definition and separate the function from the procedure, using the keywords FUNC and ENDFUNC for defining the function. 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 yet implemented in any of the versions of CBM COMAL). The value of the function is returned via the RETURN statement. In version 0.11 if the procedure name is assigned a value someplace within the procedure, it can be used as a multi-line function. 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. Versions 0.12 and 1.02 use FUNC and ENDFUNC to identify a function, while version 0.11 uses PROC and ENDPROC (the old method).

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	EXAMPLE
PROC ⟨procedure name⟩	PROC ASK

CALLING STATEMENT SYNTAX	EXAMPLE
[EXEC] ⟨procedure name⟩	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	EXAMPLE
PROC ⟨procedure name⟩ CLOSED	PROC PAUSE CLOSED

CALLING STATEMENT SYNTAX	EXAMPLE
[EXEC] ⟨procedure name⟩	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, that receive initial values from the calling EXEC 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 1.02 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 0.12.

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 (except in version 0.11). 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 for more information on this option.

The whole procedure, with all its options, looks like this:

USED AS A PROCEDURE

(all versions)

PROCEDURE STRUCTURE FORM

```
PROC ⟨procedure name⟩[ (⟨formal parameter list⟩)] [CLOSED]
  {IMPORT ⟨identifier list⟩}                    not version 0.12
  ⟨statements⟩
ENDPROC ⟨procedure name⟩
```

CALLING STATEMENT FORM

```
[EXEC] ⟨procedure name⟩[ (⟨actual parameter values⟩)]
```

PROCEDURE EXAMPLE

```
PROC ERROR(NUMBER) CLOSED
  IMPORT NAME$, ERROR'TEXT$
  PRINT "ERROR! "
  PRINT ERROR'TEXT$(NUMBER)
  PRINT "PLEASE TRY AGAIN, "; NAME$
ENDPROC ERROR
```

CALLING STATEMENT EXAMPLE

```
EXEC ERROR(5)          or          ERROR(5)
```

FUNCTION

(versions 0.12 and 1.02)

FUNCTION STRUCTURE FORM

```
FUNC ⟨function name⟩[ (⟨formal parameter list⟩) ] [CLOSED]
  {IMPORT ⟨identifier list⟩}                    not version 0.12
  ⟨statements⟩
  RETURN ⟨value⟩
ENDFUNC ⟨function name⟩
```

FUNCTION CALL FORM

```
⟨function name⟩[ (⟨actual parameter values⟩) ]        do not use EXEC
```

FUNCTION EXAMPLE

```
FUNC EVEN(INTEGER)
  RETURN (INTEGER MOD 2)
ENDFUNC EVEN
```

FUNCTION CALL EXAMPLE

```
TYPE: =EVEN(AGE)
```

USED AS A FUNCTION

(old version 0.11)

FUNCTION STRUCTURE FORM

```
PROC ⟨procedure name⟩[ (⟨formal parameter list⟩) ] [CLOSED]
  ⟨statements⟩
  ⟨procedure name⟩ : =⟨value⟩
ENDPROC ⟨procedure name⟩
```

FUNCTION CALL FORM

```
⟨procedure name⟩[ (⟨actual parameter values⟩) ]        do not use EXEC
```

FUNCTION EXAMPLE

```
PROC EVEN (INTEGER)
  EVEN: = TRUE          initialize to a TRUE answer
  IF INTEGER MOD 2 THEN
    EVEN: = FALSE       set to FALSE if a remainder exists
  ENDIF
ENDPROC EVEN
```

FUNCTION CALL EXAMPLE

```
TYPE: =EVEN (AGE)
```

APPENDIX B

STRING HANDLING

String handling in COMAL is a simple matter. It is, however, different from the method used by CBM BASIC. COMAL allows strings as well as string arrays and string functions. If you are using old version 0.11 please note that the method of specifying a substring is different.

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 CBM 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 1.02 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
```

OLD Version 0.11 specifies the substring differently. Specify a ⟨length⟩ instead of the ⟨end character⟩. The ⟨length⟩ is the number of consecutive characters to use for the substring. If it is 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$(4:6)                    start with character 4 and end with 6
                           OLD Version 0.11 use A$(4:3)
```

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$(4:4):="X"               OLD version 0.11 use A$(4:1):="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 3 spaces beginning at the third position like this:

```
A$(3:5):=""
   or                      A$ then equals "AB    FG"
A$(3:5):=" "               OLD version 0.11 use A$(3:3)=""
```

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
```

OLD Version 0.11 separates the ⟨array index part⟩ from the ⟨substring part⟩ with a comma, instead of using two sets of parentheses. Version 0.12 will accept this old format as well as the correct new method. OLD Version 0.11 also specifies a ⟨length⟩ instead of ⟨end character⟩. The ⟨length⟩ is the number of consecutive characters to use. If it is 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$(2,1)(1:3):="XYZ"                    OLD Version 0.11 use:
                                         B$(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 (in **OLD version 0.11**, the 3 means use three consecutive characters). 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 10 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 (note: string functions are not yet implemented in any CBM COMAL version):

```
⟨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
```

CBM BASIC string handling to COMAL conversion

You may wish to convert a program written in CBM BASIC to run in COMAL. BASIC uses three keywords to specify substrings: LEFT$, RIGHT$, and MID$.

```
   CBM/PET 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
```

If a string constant is used, simply determine what the result is and use the result as a string constant:

BASIC	COMAL
LEFT$(⟨string constant⟩,⟨number⟩)	⟨string constant result⟩
LEFT$("ABCDEFG",3)	"ABC"

If a string variable is specified, use 1 as the starting character and ⟨number⟩ as the end character:

BASIC	COMAL
LEFT$(⟨string variable⟩,⟨number⟩)	⟨string variable⟩(1:⟨number⟩)
LEFT$(A$,4)	A$(1:4)

If a string array is used, keep the same array index specification, use 1 as the starting character, and ⟨number⟩ as the end character:

BASIC
```
LEFT$(⟨string array name⟩(⟨array index⟩),⟨number⟩)
LEFT$(B$(2,1),3)
```

COMAL
```
⟨string array name⟩(⟨array index⟩)(1:⟨number⟩)
B$(2,1)(1:3)
```

CBM/PET 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
```

If a string constant is used, simply determine what the result is and use the result as a string constant:

BASIC	COMAL
RIGHT$(⟨string constant⟩,⟨number⟩)	⟨string constant result⟩
RIGHT$("ABCDEFG",3)	"EFG"

If a string variable is specified, use the length of the string minus the ⟨number⟩ plus 1 as the starting character, and the length of the string as the ⟨end character⟩ (**OLD Versions** use ⟨number⟩ as the length):

BASIC
```
RIGHT$(⟨string var⟩,⟨number⟩)
RIGHT$(A$,4)
```

COMAL
```
⟨string var⟩(LEN(⟨string var⟩)-⟨number⟩+1:LEN(⟨string var⟩))
A$(LEN(A$)-4+1:LEN(A$))        or    A$(LEN(A$)-3:LEN(A$))
```

If a string array is used, keep the same array index specification, use the length of the string minus the ⟨number⟩ plus 1 as the starting character, and the length of the string as the ⟨end character⟩ (**OLD Version 0.11** uses ⟨number⟩ as the length).

BASIC
```
RIGHT$(⟨string array name⟩(⟨array index⟩),⟨number⟩)
RIGHT$(B$(2,1),3)
```

COMAL
```
TEMP=LEN(⟨string array name⟩(⟨array index⟩))
⟨string array name⟩(⟨array index⟩)(TEMP-⟨number⟩+1:TEMP)

TEMP=LEN(B$(2,1))
B$(2,1)(TEMP-3+1:TEMP)        or        B$(2,1)(TEMP-2:TEMP)
```

CBM/PET 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
```

If a string constant is used, simply determine what the result is and use the result as a string constant:

BASIC
```
MID$(⟨string constant⟩,⟨start⟩[,⟨number⟩])
MID$("ABCDEFG",3,2)
```

COMAL
```
⟨string constant result⟩
"CD"
```

If a string variable is specified, use ⟨start⟩ as ⟨start⟩, and ⟨start⟩ plus ⟨number⟩ minus 1 as the ⟨end character⟩ (if ⟨number⟩ is omitted use the length of the string as the ⟨end character⟩) (**OLD Version 0.11** uses ⟨number⟩ as the length (if ⟨number⟩ is omitted use the length of the string minus the ⟨start⟩ plus one)):

BASIC
```
MID$(⟨string variable⟩,⟨start⟩[,⟨number⟩])
MID$(A$,4)          and      MID$(B$,2,3)
```

COMAL
```
⟨string variable⟩(⟨start⟩ :⟨start⟩+⟨number⟩-1)
A$(4:LEN(A$))    and          B$(2:2+3-1)
                                  or
                              B$(2:4)
```

If a string array is used, keep the same array index specification, use ⟨start⟩ as ⟨start⟩, and ⟨start⟩ plus ⟨number⟩ minus 1 as the ⟨end character⟩ (if ⟨number⟩ is omitted use the length of the string as the ⟨end character⟩) (**OLD Version 0.11** uses ⟨number⟩ as the length (if ⟨number⟩ is omitted use LEN(⟨string array name⟩(⟨array index⟩)) minus the ⟨start⟩ plus one)):

BASIC
```
MID$(⟨string array name⟩(⟨array index⟩),⟨start⟩[,⟨number⟩])
```

a) ⟨number⟩ is specified
```
MID$(B$(2,1),3,1)
```
b) ⟨number⟩ is not specified
```
MID$(C$(3,2),2)
```

COMAL
a) if ⟨number⟩ is specified:
```
⟨string array name⟩(⟨array index⟩) (⟨start⟩:⟨start⟩+⟨number⟩-1)

B$(2,1)(3:3+1-1)
       or
B$(2,1)(3:3)
       or
B$(2,1)(3)
```

b) if ⟨number⟩ is not specified:
```
TEMP:=LEN(⟨string array name⟩(⟨array index⟩))
⟨string array name⟩(⟨array index⟩) (⟨start⟩:TEMP)

C$(3,2)(2:LEN(C$(3,2)))
       or
TEMP=LEN(C$(3,2))
C$(3,2)(2:TEMP)
```

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 - 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 5 byte binary coded record, and integer data as a 2 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 - 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 statement. The two byte delimiter used by COMAL is CHR$(13) and CHR$(10) (a carriage return, linefeed). Both string records and numeric records use this method. (A COMAL INPUT FILE statement will also read a CBM BASIC file, which uses only 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 CBM BASIC. COMAL represents a numeric value just as it is, thus a 5 is represented as **5**. However, CBM BASIC precedes each number with one byte for the sign (- for negative, ⟨space⟩ for non-negative) and ends each number with a cursor right. Therefore, a 5 is represented as **⟨space⟩5⟨cursor right⟩**. Thus for COMAL to read a CBM BASIC numeric file, a short conversion routine would have to be used. However, COMAL can read a CBM 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 5 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

HOW 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 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 (not LOAD) to retrieve it.

To avoid line number problems 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.

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.

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                          just hit return here
```

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.

```
ENTER "TAKE'IN.L"
RENUM
9999 //
ENTER "FETCH.L"
RENUM
```

```
9999 //
ENTER "GET'VALID.L"
RENUM
9999 //
ENTER "GET'CHAR.L"
RENUM
```

Now, list the complete program:

```
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  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 0.12
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$,max) closed
0310  import get'valid,get'char // not needed in version 0.12
0320  dim valid$ 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 va
      lid$
```

```
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 "= =", // 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$) closed
0750  import get'char // not used in version 0.12
0760  repeat
0770   get'char(c$)
0780  until c$ in valid$
0790 endproc get'valid
0800 //
0810 proc get'char(ref c$) closed
0820  buffer'count'loc:=158
0830  buffer'loc:=623
0840  poke buffer'count'loc,0 //        clear keyboard buffer
0850  repeat
0860  until peek(buffer'count'loc) //   wait till count>0
0870  c$:=chr$(peek(buffer'loc)) //     assign string
0880  poke buffer'count'loc,0 //        reset buffer count to 0
0890 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)
```

PROCEDURE NAME: BASIC'RESET
NOTE: Needed only in version 0.11, other versions use command BASIC.

This procedure is used to reset the computer to the CBM/PET BASIC mode, simulating a cold start reset. Thus you do not have to turn the power off and back on again to go from COMAL to BASIC (which is hard on the chips in the computer). Of course, this SYS command can be used as part of the main program instead of a procedure, or even as a direct command.

SYNTAX FOR CALLING STATEMENT

```
EXEC BASIC'RESET
```

EXAMPLE OF CALLING STATEMENT

```
EXEC BASIC'RESET
```

PROCEDURE LISTING

```
PROC BASIC'RESET
  SYS 64790
ENDPROC BASIC'RESET
```

PROCEDURE NAME: BOLD'CHAR
NOTE1: Written for the STARWRITER printer
NOTE2: SELECT OUTPUT "LP" previously

This procedure will print the specified ⟨character⟩ in boldface on the printer. It is designed to be called by the procedure BOLD'FACE, but may be called directly if needed. Only one character at a time should be passed to this procedure or it will not function correctly. To bold face more than one character use the BOLD'FACE procedure. This procedure is written specifically for a STARWRITER. Other printers that have boldface capability will probably require modifications to the control codes sent to the printer. It also assumes that output is already directed to the printer via a previous SELECT OUTPUT "LP" command.

SYNTAX FOR CALLING STATEMENT

```
EXEC BOLD'CHAR(⟨character⟩)

  ⟨character⟩ is a ⟨string expression⟩, INPUT PARAMETER
    this is the character to be printed in bold face
```

EXAMPLES OF CALLING STATEMENT

```
EXEC BOLD'CHAR("A")
EXEC BOLD'CHAR(C$)
```

PROCEDURE LISTING

```
PROC BOLD'CHAR(B$) CLOSED
  //IF LEN(B$))1 THEN                                  optional error
  //   SELECT OUTPUT "DS"                              handling lines
  //   PRINT "ERROR IN PROCEDURE BOLD'CHAR"            may be included
  //   PRINT "TOO MANY CHARACTERS IN B$"               if they may be
  //   PRINT "B$ IS: ";B$                              needed – they stop
  //   STOP                                            the program if more
  //ENDIF                                              than 1 char received
  Z=ZONE                          remember the current ZONE setting
  ZONE 0                          set ZONE for tab in each column
  PRINT CHR$(27),"g",             ESCAPE G=set to graphics mode
  FOR X=1 TO 3 DO PRINT B$,           (shifted G)
  PRINT CHR$(27),"4",             ESCAPE 4=reset to normal mode
  PRINT " ",              move over 1 character, make up for prev mode
  ZONE Z                  reset zone back to the original setting
ENDPROC BOLD'CHAR
```

PROCEDURE NAME: BOLD'FACE
REQUIRES PROCEDURE: BOLD'CHAR
NOTE1: Written for the STARWRITER printer.
NOTE2: SELECT OUTPUT "LP" previously

This procedure will print each character of the ⟨string⟩ sent to it in bold face type on the printer. It calls the procedure BOLD'CHAR one character at a time to do the actual printing. It is written specifically for the STARWRITER. Other printers with the capability of bold face style type will probably have to modify the control codes used in procedure BOLD'CHAR. The last parameter in the procedure, ⟨carriage return⟩, allows you the option of printing a carriage return after the string. If you are bold facing a word in the middle of a line, you will not want a carriage return, and should send a 0 (FALSE) to this parameter. Any other value (TRUE) will result in a carriage return after the string is printed. It is assumed that the output is already directed to the printer via a previous SELECT OUTPUT "LP" statement.

SYNTAX FOR CALLING STATEMENT

```
EXEC BOLD'FACE (⟨string⟩,⟨carriage return⟩)

  ⟨string⟩ is a ⟨string expression⟩, INPUT PARAMETER
    these are the characters to be printed in bold face
  ⟨carriage return⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    if TRUE (a value not equal to 0) a carriage return is issued
    if FALSE (a value of 0) a carriage return is not issued
```

EXAMPLES OF CALLING STATEMENT

```
EXEC BOLD'FACE ("TESTING", 1)
EXEC BOLD'FACE (NAME$, FALSE)
```

If a variable named C'RETURN is assigned the value of TRUE at the beginning of your program, a carriage return can be specified in a much clearer manner as the following examples illustrate.

```
DIM N$ OF 10
C'RETURN=TRUE; N$="TESTING"
... // program is here
EXEC BOLD'FACE (N$, C'RETURN)              a carriage return is issued
EXEC BOLD'FACE (N$, C'RETURN=TRUE)         a carriage return is issued
EXEC BOLD'FACE (N$, C'RETURN=FALSE)        no carriage return is issued
EXEC BOLD'FACE (N$, NOT C'RETURN)          no carriage return is issued
```

PROCEDURE LISTING

```
PROC BOLD'FACE(B$,C'RETURN) CLOSED
  IMPORT BOLD'CHAR                          not used in version 0.12
  FOR CHAR=1 TO LEN(B$) DO EXEC BOLD'CHAR(B$(CHAR))
  IF C'RETURN THEN PRINT                    do a carriage return if TRUE
ENDPROC BOLD'FACE
```

PROCEDURE NAME: CLEAR'FROM
REQUIRES FUNCTION: SCREEN'POS

This procedure will clear the bottom section of the screen starting with the screen line specified by the parameter ⟨line⟩.

NOTE

The screen lines are referred to as lines 1-25, with the top line as line 1.

SYNTAX FOR CALLING STATEMENT

```
EXEC CLEAR'FROM(⟨line⟩)

  ⟨line⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the line used as the start of the area to clear
```

EXAMPLES OF CALLING STATEMENT

```
EXEC CLEAR'FROM(20)              clears lines 20 thru 25
EXEC CLEAR'FROM(X)
EXEC CLEAR'FROM(PANEL)
```

PROCEDURE LISTING

```
PROC CLEAR'FROM(LINE) CLOSED
  IMPORT SCREEN'POS                  not used in version 0.12
  FOR X=SCREEN'POS(LINE,1) TO SCREEN'POS(26,1)-1 DO POKE X,32
ENDPROC CLEAR'FROM
```

PROCEDURE NAME: CLEAR'LINE
REQUIRES FUNCTION: SCREEN'POS

This procedure will clear the line specified by the parameter ⟨line⟩.

NOTE

The screen lines are referred to as lines 1-25, with the top line as line 1.

SYNTAX FOR CALLING STATEMENT

```
EXEC CLEAR'LINE (⟨line⟩)

  ⟨line⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the screen line to be cleared
```

EXAMPLES OF CALLING STATEMENT

```
EXEC CLEAR'LINE(20)              clears line 20
EXEC CLEAR'LINE(X)
EXEC CLEAR'LINE(LINE)
```

PROCEDURE LISTING

```
PROC CLEAR'LINE(LINE) CLOSED
  IMPORT SCREEN'POS                    not used in version 0.12
  FOR X=SCREEN'POS(LINE,1) TO SCREEN'POS(LINE+1,1)-1 DO POKE X,32
ENDPROC CLEAR'LINE
```

PROCEDURE NAME: CLEAR'TO
REQUIRES FUNCTION: SCREEN'POS

This procedure will clear the top of the screen up to and including the line specified by the parameter ⟨line⟩.

NOTE

The screen lines are referred to as lines 1-25, with the top line as line 1.

SYNTAX FOR CALLING STATEMENT

```
EXEC CLEAR'TO(⟨line⟩)

  ⟨line⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the last screen line to be cleared
```

EXAMPLES OF CALLING STATEMENT

```
EXEC CLEAR'TO(20)              clears lines 1 thru 20
EXEC CLEAR'TO(X)
EXEC CLEAR'TO(LINE)
```

PROCEDURE LISTING

```
PROC CLEAR'TO(LINE) CLOSED
  IMPORT SCREEN'POS                not used in version 0.12
  FOR X=SCREEN'POS(1,1) TO SCREEN'POS(LINE+1,1)-1 DO POKE X,32
ENDPROC CLEAR'TO
```

PROCEDURE NAME: CLEAR'WINDOW
REQUIRES PROCEDURE: CLEAR'LINE
REQUIRES FUNCTION: SCREEN'POS

This procedure will clear the section of the screen specified by the ⟨top line⟩ and ⟨bottom line⟩ parameters. It can clear any number of consecutive lines on the screen.

NOTE

The screen lines are referred to as lines 1-25, with the top line as line 1.

SYNTAX FOR CALLING STATEMENT

```
EXEC CLEAR'WINDOW (⟨top line⟩,⟨bottom line⟩)

  ⟨top line⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the first line to be cleared
  ⟨bottom line⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the last line to be cleared
```

EXAMPLES OF CALLING STATEMENT

```
EXEC CLEAR'WINDOW (5, 9)              clears lines 5, 6, 7, 8, and 9
EXEC CLEAR'WINDOW (TOP, BOTTOM)
```

PROCEDURE LISTING

```
PROC CLEAR'WINDOW (START'LINE, END'LINE) CLOSED
  IMPORT CLEAR'LINE                    not used in version 0.12
  FOR X= START'LINE TO END'LINE DO EXEC CLEAR'LINE (X)
ENDPROC CLEAR'WINDOW
```

PROCEDURE NAME: CURSOR
NOTE: Not needed in version 1.02 where it is a statement that does the same thing.

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 0                                    set ZONE to tab in every column
  POKE 205, 0                               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
```

PROCEDURE NAME: DISK'COMMAND
NOTE: Needed only by version 0.11. Other versions may use the keyword PASS to do the same thing.

This procedure is used to relay commands to CBM disk drives. The commands must follow Commodore DOS 1.0 conventions (see the examples for clarification). For complete descriptions of the DOS 1.0 disk commands, see your CBM disk user manual. The parameter ⟨command⟩ is relayed to the disk drive. Although this method is a bit troublesome, it makes up for the lack of disk utility commands built into COMAL.

SYNTAX FOR CALLING STATEMENT

```
EXEC DISK'COMMAND(⟨command⟩)

  ⟨command⟩ is a ⟨string expression⟩, INPUT PARAMETER
    the command that will be issued to the disk drive
```

EXAMPLES OF CALLING STATEMENT

```
EXEC DISK'COMMAND("I0")                          initialize drive 0
EXEC DISK'COMMAND("S0:TEMP'FILE")                scratch TEMP'FILE on drive 0
EXEC DISK'COMMAND("N1:WORK DISK,ID")             new (format) disk in drive 1
EXEC DISK'COMMAND("V0")                          validate (collect) drive 0
EXEC DISK'COMMAND("D1=0")                        duplicate drive 0 to drive 1
EXEC DISK'COMMAND("D0=1")                        duplicate drive 1 to drive 0
EXEC DISK'COMMAND("R0:NEW'NAME=0:OLD'NAME")                    rename a file
EXEC DISK'COMMAND("C0:FILENAME=1:FILENAME")                    copy a file
EXEC DISK'COMMAND(DISK'COM$)                  string variables may be used
```

PROCEDURE LISTING

```
PROC DISK'COMMAND(COMMAND$) CLOSED
  CLOSE 15                    make sure the file is closed
  OPEN 15,"",UNIT 8,15
  PRINT FILE 15 : COMMAND$
  CLOSE 15
ENDPROC DISK'COMMAND
```

FUNCTION NAME: DISK'GET
REQUIRES PROCEDURE: DISK'GET'INIT
NOTE: This function is needed only with version 0.12. In version 1.02 use GET$ instead.

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. (**OLD Version 0.11** should change this to a procedure used as a function).

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 0) while not at the end of file
```

EXAMPLES OF THE FUNCTION CALL

```
B$=CHR$(DISK'GET(INFILE,FILE'END))
DC=DISK'GET(1,EOF1)
```

FUNCTION LISTING

```
FUNC DISK'GET(FILE'NUM,REF FILE'END) CLOSED

  POKE 636,FILE'NUM
  SYS 635                          execute machine language routine
  FILE'END=PEEK(150)               set to TRUE at end of file
  RETURN PEEK(634)                 ASCII value of the character
  // DISK'GET=PEEK(634)            OLD version 0.11 use this line in
ENDFUNC DISK'GET                     place of the above RETURN
```

NOTE: OLD Version 0.11 replace FUNC with PROC and replace ENDFUNC with ENDPROC.

PROCEDURE NAME: DISK'GET'INIT
NOTE: Needed only with version 0.12.

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

```
PROC DISK'GET'INIT CLOSED
  FOR LOCATION=634 TO 649                cassette buffer locations
    READ VALUE
    POKE LOCATION,VALUE                  put the code into the buffer
  NEXT LOCATION
  DATA 0,162,0,32,198,255,32,207
  DATA 255,141,122,2,32,204,255,96
ENDPROC DISK'GET'INIT
```

PROCEDURE NAME: DISK'GET'SKIP
REQUIRES PROCEDURE: DISK'GET'INIT
REQUIRES FUNCTION: DISK'GET
NOTE: Needed only with version 0.12.

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 0) 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
  IMPORT DISK'GET                  not used in version 0.12
  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 0.12.

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 0) 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$,COUNT,FILE'NUM,REF FILE'END) CLOSED
  IMPORT DISK'GET                          not used in version 0.12
  ITEM$=""                                 initialize the string
  FOR X=1 TO COUNT DO ITEM$(X)=CHR$(DISK'GET(FILE'NUMS,FILE'END))
ENDPROC DISK'GET'STRING
```

PROCEDURE NAME: DOUBLE'CHAR
NOTE1: SELECT OUTPUT "LP" previously
NOTE2: Printer must have backspace capability.

This procedure will double strike the specified ⟨character⟩ on the printer. It is written for printers that have the ability to backspace, and will not work on other printers. It is designed to be called from the DOUBLE'STRIKE procedure, but you may call it directly if you wish. It assumes that output is already directed to the printer via a previous SELECT OUTPUT "LP" command. Only one character at a time should be passed to this procedure or it will not function properly. To double strike more than one character use the DOUBLE'STRIKE procedure.

SYNTAX FOR CALLING STATEMENT

```
EXEC DOUBLE'CHAR(⟨character⟩)

  ⟨character⟩ is a ⟨string expression⟩, INPUT PARAMETER
    the character to be printed twice
```

EXAMPLES OF CALLING STATEMENT

```
EXEC DOUBLE'CHAR("D")              prints the letter D twice
EXEC DOUBLE'CHAR(D$)
```

PROCEDURE LISTING

```
PROC DOUBLE'CHAR(D$) CLOSED
  //IF LEN(D$)>1 THEN                          error handling
  //   SELECT OUTPUT "DS"                      can be included
  //   PRINT "AN ERROR HAS OCCURRED"           if it is needed
  //   PRINT "IN PROCEDURE DOUBLE'CHAR"        just remove the //
  //   PRINT "VARIABLE D$ WAS TOO LONG"        in front of the lines
  //   PRINT "D$ WAS: ";D$                     it will stop the
  //   STOP                                    program if D$ is
  //ENDIF                                      too long
  Z=ZONE                        remember current ZONE
  ZONE 0                        set ZONE for a tab in each column
  PRINT D$,CHR$(8),D$,          CHR$(8) is backspace
  ZONE Z                        reset back to original ZONE
ENDPROC DOUBLE'CHAR
```

PROCEDURE NAME: DOUBLE'STRIKE
REQUIRES PROCEDURE: DOUBLE'CHAR
NOTE1: SELECT OUTPUT "LP" previously
NOTE2: Printer must have backspace capability.

This procedure will double strike each character in the ⟨string⟩ sent to it on the printer. It is written for printers that have the ability to backspace, and will not work on other printers. It calls procedure DOUBLE'CHAR one character at a time to do the actual printing. The last parameter ⟨carriage return⟩ allows you the option of printing a carriage return after the string. If you are double striking a word in the middle of a sentence, you will not want a carriage return, and should send a 0 (FALSE) to the parameter. Any other value (TRUE) will result in a carriage return after the string is printed.

SYNTAX FOR CALLING STATEMENT

```
EXEC DOUBLE'STRIKE (⟨string⟩, ⟨carriage return⟩) CLOSED

  ⟨string⟩ is a ⟨string expression⟩, INPUT PARAMETER
    the characters to be printed twice (double striked)
  ⟨carriage return⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    if TRUE (a value not equal to 0) a carriage return is issued
    if FALSE (a value of 0) a carriage return is not issued
```

EXAMPLES OF CALLING STATEMENT

```
EXEC DOUBLE'STRIKE ("TESTING", TRUE)      a carriage return is issued
EXEC DOUBLE'STRIKE (NAME$, 0)             a carriage return is not issued
```

If a variable named C'RETURN is assigned the value of TRUE at the beginning of your program, a carriage return can be specified in a much clearer manner as the following examples illustrate.

```
DIM N$ OF 10
C'RETURN=TRUE; N$="TESTING"
... // program goes here
EXEC DOUBLE'STRIKE (N$, C'RETURN)          a carriage return is sent
EXEC DOUBLE'STRIKE (N$, C'RETURN=TRUE)     a carriage return is sent
EXEC DOUBLE'STRIKE (N$, C'RETURN=FALSE)    no carriage return is sent
EXEC DOUBLE'STRIKE (N$, NOT C'RETURN)      no carriage return is sent
```

PROCEDURE LISTING

```
PROC DOUBLE'STRIKE(N$,C'RETURN) CLOSED
  IMPORT DOUBLE'CHAR                    not used in version 0.12
  FOR CHAR= 1 TO LEN(D$) DO EXEC DOUBLE'CHAR(D$(CHAR))
  IF C'RETURN THEN PRINT                provides a carriage return
ENDPROC DOUBLE'STRIKE
```

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", 40)
EXEC FETCH (Y'OR'N$, "YN", 1)
```

PROCEDURE LISTING

```
PROC FETCH(REF A$, V$, MAX) CLOSED
  IMPORT GET'VALID                    not used in version 0.12
  DIM VALID$ OF 40, B$ OF 1
  Z=ZONE                              remember current ZONE
  ZONE 0                              set ZONE for tab in each column
  A$=""                               initialize
  CASE V$ OF                          the cases can be changed to your needs
  WHEN "A"
    VALID$="ABCDEFGHIJKLMNOPQRSTUVWXYZ "      include final space
  WHEN "D"
    VALID$="0123456789"
  WHEN "B"
    VALID$="ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789"      include space
  OTHERWISE
    VALID$=V$
  ENDCASE
  DONE=FALSE; NUM=0                   initialize
  REPEAT
    EXEC GET'VALID(B$, VALID$+CHR$(13)+CHR$(20))        add CR & DELETE
    CASE B$ OF
    WHEN CHR$(13)                     carriage return
      DONE=TRUE                       end of input
    WHEN CHR$(20)                     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
```

PROCEDURE NAME: GET'ALPHA
REQUIRES PROCEDURES: GET'CHAR

This procedure is used to get one character from the keyboard and will only accept a letter of the alphabet. All other characters are ignored. The procedure GET'CHAR is used actually to get the character.

SYNTAX FOR CALLING STATEMENT

```
EXEC GET'ALPHA (⟨reply⟩)

  ⟨reply⟩ is a ⟨string variable⟩, OUTPUT PARAMETER
    the character of the key hit is assigned to this variable
```

EXAMPLES OF CALLING STATEMENT

```
EXEC GET'ALPHA (CHOICE$)
EXEC GET'ALPHA (X$)
```

PROCEDURE LISTING

```
PROC GET'ALPHA (REF C$) CLOSED
  IMPORT GET'CHAR                    not used in version 0.12
  REPEAT
    EXEC GET'CHAR (C$)
  UNTIL C$⟩="A" AND C$⟨="Z"
ENDPROC GET'ALPHA
```

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 PET/CBM 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. A version is included for both version 0.12 and 1.02.

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 0.12

```
PROC GET'CHAR(REF C$) CLOSED
  BUFFER'COUNT'LOC=158
  BUFFER'LOC=623
  POKE BUFFER'COUNT'LOC,0 //          clear keyboard buffer
  REPEAT
  UNTIL PEEK(BUFFER'COUNT'LOC) //     wait till count⟩0
  C$=CHR$(PEEK(BUFFER'LOC)) //        assign string
  POKE BUFFER'COUNT'LOC,0            reset buffer count to 0
ENDPROC GET'CHAR
```

VERSION 1.02

```
PROC GET'CHAR(REF C$) CLOSED
  FOR TEMP=1 TO 9 DO C$=KEY$         clear keyboard buffer
  C$=GET$(0,1)                       file 0 is the keyboard
ENDPROC GET'CHAR
```

PROCEDURE NAME: GET'DIGIT
REQUIRES PROCEDURE: GET'CHAR

This procedure is used to get one digit (i.e., 0,1,2,3,4,5,6,7,8,9) from the keyboard without the need for hitting the RETURN key. It calls procedure GET'CHAR actually to get the character.

SYNTAX FOR CALLING STATEMENT

```
EXEC GET'DIGIT(⟨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'DIGIT(D$)
EXEC GET'DIGIT(NUMBER$)
```

PROCEDURE LISTING

```
PROC GET'DIGIT(REF C$) CLOSED
  IMPORT GET'CHAR                not used in version 0.12
  REPEAT
    EXEC GET'CHAR(C$)
  UNTIL C$ IN "0123456789"
ENDPROC GET'DIGIT
```

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 0.12
  REPEAT
    EXEC GET'CHAR(C$)
  UNTIL C$ IN VALID$
ENDPROC GET'VALID
```

FUNCTION NAME: JIFFIES

This function will return the current number of jiffies (60 jiffies in 1 second) using the PET/CBM 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). It is not needed in version 1.02 where the keyword TIME does the same thing. **(OLD Version 0.11** change it to a procedure used as a function).

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)
  RETURN J                    OLD version 0.11 use JIFFIES:=J
ENDFUNC JIFFIES
```

(NOTE: **OLD Version 0.11** replace FUNC with PROC and replace ENDFUNC with ENDPROC)

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 character 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$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  FOR X=1 TO LEN(A$)
    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
  NEXT X
ENDPROC LOWER'TO'UPPER
```

PROCEDURE NAME: MULTI'CHAR
NOTE1: SELECT OUTPUT "LP" previously
NOTE2: Printer must have backspace capability.

This procedure will multi-strike the specified ⟨character⟩ as many times as specified by the parameter ⟨strikes⟩. It is written for printers that use CHR$(8) for backspacing. It is written to be called from the procedure MULTI'STRIKE, but can be called directly if needed. To multi-strike more than one character, use the procedure MULTI'STRIKE.

SYNTAX FOR CALLING STATEMENT

EXEC MULTI'CHAR(⟨character⟩,⟨strikes⟩)

⟨character⟩ is a ⟨string expression⟩, INPUT PARAMETER
the character to be printed several times in the same position
⟨strikes⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
the number of times to print the character

EXAMPLES OF CALLING STATEMENT

```
EXEC MULTI'CHAR("M",4)          prints the letter M four times
EXEC MULTI'CHAR(C$,MSNUM)
```

PROCEDURE LISTING

```
PROC MULTI'CHAR(M$,COUNT) CLOSED
  //IF LEN(M$))1 THEN                          error handling
  //   SELECT OUTPUT "DS"                      can be included
  //   PRINT "AN ERROR HAS OCCURRED"           if it is needed
  //   PRINT "IN PROCEDURE MULTI'CHAR"         just remove the //
  //   PRINT "TOO MANY CHARACTERS IN M$"       in front of the lines
  //   PRINT "M$ IS: ";M$                      it will stop the
  //   STOP                                    program if M$ is
  //ENDIF                                      too long
  Z=ZONE                          remember the current ZONE
  ZONE 0                          set ZONE to tab at each column
  PRINT M$,
  FOR X=1 TO COUNT-1 DO PRINT CHR$(8),M$,     CHR$(8) is backspace
  ZONE Z                          reset back to original ZONE
ENDPROC MULTI'CHAR
```

PROCEDURE NAME: MULTI'STRIKE
REQUIRES PROCEDURE: MULTI'CHAR
NOTE1: SELECT OUTPUT "LP" previously
NOTE2: Printer must have backspace capability.

This procedure will multi-strike the specified string one character at a time, as many times as specified by the parameter ⟨strikes⟩. It is written for printers that use CHR$(8) for backspacing. It is designed to be called by procedure MULTI'STRIKE, but can be called directly if needed.

SYNTAX FOR CALLING STATEMENT

```
EXEC MULTI'STRIKE(⟨characters⟩,⟨strikes⟩,⟨carriage return⟩)

  ⟨characters⟩ is a ⟨string expression⟩, INPUT PARAMETER
    the characters to be multi-striked
  ⟨strikes⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the number of times to print each character
  ⟨carriage return⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    if TRUE (a value not equal to 0) a carriage return is issued
    if FALSE (a value of 0) a carriage return is not issued
```

EXAMPLES OF CALLING STATEMENT

```
EXEC MULTI'STRIKE("YES",4,0)                    each letter of YES four times
EXEC MULTI'STRIKE(C$,MSNUM,LF)
```

If a variable named C'RETURN is assigned the value of TRUE at the beginning of your program, a carriage return can be specified in a much clearer manner as the following examples illustrate.

```
DIM N$ OF 10
C'RETURN=TRUE; N$="TESTING"
... // program goes here
EXEC MULTI'STRIKE(N$,5,C'RETURN)                a carriage return is issued
EXEC MULTI'STRIKE(N$,5,C'RETURN=TRUE)           a carriage return is issued
EXEC MULTI'STRIKE(N$,5,C'RETURN=FALSE)          no carriage return issued
EXEC MULTI'STRIKE(N$,5,NOT C'RETURN)            no carriage return issued
```

PROCEDURE LISTING

```
PROC MULTI'STRIKE(M$,COUNT,C'RETURN) CLOSED
  IMPORT MULTI'CHAR                    not used in version 0.12
  FOR CHAR=1 TO LEN(M$) DO EXEC MULTI'CHAR(M$(CHAR),COUNT))
  IF C'RETURN THEN PRINT               provide carriage return
ENDPROC MULTI'STRIKE
```

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, POSITION)
```

PROCEDURE LISTING

```
PROC POS (REF LINE, REF COL) CLOSED
  LINE'PEEK=216
  COL'PEEK=198
  LINE=PEEK (LINE'PEEK) +1
  COL=PEEK (COL'PEEK) +1
ENDPROC POS
```

PROCEDURE NAME: SCAN
NOTE: Not needed in version 1.02 - use KEY$ instead.

This procedure is similar to the GET command in PET/CBM 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 SCAN (⟨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 SCAN (MOVE$)
EXEC SCAN (A$)
```

PROCEDURE LISTING

```
PROC SCAN (REF C$) CLOSED
  BUFFER'COUNT'LOC=158
  BUFFER'LOC=623
  A=PEEK (BUFFER'COUNT'LOC)              number of keys hit
  IF A THEN                              at least one key was hit
    POKE BUFFER'COUNT'LOC, A-1           decrement count
    C$=CHR$ (PEEK (BUFFER'LOC-1+A) )     last key hit character
  ELSE                                   no key was hit
    C$=CHR$ (0)                          same as KEY$ in version 1.02
  ENDIF
ENDPROC SCAN
```

FUNCTION NAME: SCREEN'POS

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. In OLD Version 0.11 change it to a procedure used as a function.

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 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                    OLD version 0.11 use SCREEN'POS:=S
ENDFUNC SCREEN'POS
```

NOTE: **OLD Version 0.11** replace FUNC with PROC and replace ENDFUNC with ENDPROC.

PROCEDURE NAME: SET'LINE'HGT
NOTE1: SELECT OUTPUT "LP" previously
NOTE2: Written for the STARWRITER

This procedure is used to set the specific line spacing on printers that allow it. It was written for the STARWRITER. Other printers may require modifications. Make sure that a SELECT OUTPUT "LP" is issued prior to calling this routine. Standard line heights are 6 (for 8 lines per inch) and 8 (for 6 lines per inch). If your printer is double spacing your lines, and you wish it weren't, simply set the line height to 4 (half a line), then two half lines will appear like single spacing.

SYNTAX FOR CALLING STATEMENT

```
EXEC SET'LINE'HGT (⟨number⟩)

  ⟨number⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the number of 1/48ths of a line to use in between lines
```

EXAMPLES OF CALLING STATEMENT

```
EXEC SET'LINE'HGT (13)
EXEC SET'LINE'HGT (WIDE)
```

PROCEDURE LISTING

```
PROC SET'LINE'HGT (N) CLOSED
  Z = ZONE                                  remember ZONE setting
  ZONE 0                                    set ZONE for tab in each column
  PRINT CHR$ (27) , CHR$ (30) , CHR$ (N) ,  control codes
  ZONE Z                                    reset original ZONE
ENDPROC SET'LINE'HGT
```

PROCEDURE NAME: SET'PITCH
NOTE1: SELECT OUTPUT "LP" previously
NOTE2: Written for the STARWRITER

This procedure is used to set the pitch (characters per inch) on the printer. Common pitches are 8, 10, 12, and 15. PICA is 10 pitch, while ELITE is 12 pitch. The larger the number, the closer together the characters are printed. This procedure was written for the STARWRITER. Other printers may require modifications. Make sure to issue the SELECT OUTPUT "LP" command prior to calling this routine. The STARWRITER specifies its pitch in the number of 1/120ths of an inch.

SYNTAX FOR CALLING STATEMENT

```
EXEC SET'PITCH(⟨pitch⟩)

  ⟨pitch⟩ is a ⟨numeric expression⟩, INPUT PARAMETER
    the pitch desired
```

EXAMPLES OF CALLING STATEMENT

```
EXEC SET'PITCH(PICA)
EXEC SET'PITCH(12)
```

PROCEDURE LISTING

```
PROC SET'PITCH(P) CLOSED
  Z=ZONE                                  remember current ZONE
  ZONE 0                                  set ZONE for tab in each column
  PRINT CHR$(27),CHR$(31),CHR$(120 DIV P)      control codes
  ZONE Z                                  reset ZONE to original
ENDPROC SET'PITCH
```

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. In OLD Version 0.11 change this to a procedure used as a function.

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
  S=PEEK(SHIFT'FLAG)
  RETURN S                          OLD version 0.11 use SHIFT:=S
ENDFUNC SHIFT
```

NOTE: **OLD Version 0.11** replace FUNC with PROC and ENDFUNC with ENDPROC.

PROCEDURE NAME: SHIFT'WAIT

This procedure waits until the SHIFT key is depressed. Once SHIFT is depressed the procedure returns control back to the calling EXEC statement. Since the keyboard will seem "locked out" during this routine, make sure to include instructions to press SHIFT to continue.

SYNTAX FOR CALLING STATEMENT

```
EXEC SHIFT'WAIT
```

EXAMPLE OF CALLING STATEMENT

```
EXEC SHIFT'WAIT
```

PROCEDURE LISTING

```
PROC SHIFT'WAIT CLOSED
  SHIFT'FLAG=152
  REPEAT
  UNTIL PEEK(SHIFT'FLAG)          wait till SHIFT key depressed
ENDPROC SHIFT'WAIT
```

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 0.12
  Z=ZONE                                        remember current zone
  ZONE 0                                        set ZONE to tab in each column
  PRINT PROMPT$,                                print the prompt
  PRINT "⟨",                                    define left of input area
  FOR X=1 TO MAX DO PRINT " ",                  blank out input area
  PRINT "⟩",                                    define right of input area
  FOR X=1 TO MAX+1 DO PRINT "[LEFT]",           cursor left back to first spot
  EXEC FETCH(REPLY$,VALID$,MAX)                 get the input
  PRINT                                         carriage return
  ZONE Z                                        reset ZONE to original
ENDPROC TAKE'IN
```

PROCEDURE NAME: UPPER'TO'LOWER

This procedure converts any upper case letter in the specified string to lower 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 letter from a string.

SYNTAX FOR CALLING STATEMENT

```
EXEC UPPER'TO'LOWER(⟨string⟩)
```

⟨string⟩ is a ⟨string variable⟩, UNIVERSAL INPUT/OUTPUT PARAMETER
any upper case characters in this string are converted

EXAMPLES OF CALLING STATEMENT

```
EXEC UPPER'TO'LOWER(NAME$)
EXEC UPPER'TO'LOWER(TEXT$)
```

PROCEDURE LISTING

```
PROC UPPER'TO'LOWER(REF A$) CLOSED
  DIM Z$ OF 26, C$ OF 1
  Z$="abcdefghijklmnopqrstuvwxyz"
  FOR X=1 TO LEN(A$)
    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
  NEXT X
ENDPROC UPPER'TO'LOWER
```

FUNCTION NAME: VALUE
NOTE: Not needed in version 1.02 where VAL is a built in function.

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 1.02 does not need this function since VAL is a built in function. OLD Version 0.11 change this to a procedure used as a 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$) CLOSED
  L = LEN (S$)
  ONES = ORD (S$ (L) ) -ORD ("0")
  IF L = 1 THEN
    RETURN ONES
    // OLD version 0.11 use the line below in place of RETURN
    // VALUE = ONES
  ELSE
    RETURN ONES + VALUE (S$ (1: L-1) ) *10        recursive call
    // OLD version 0.11 use the line below in place of RETURN
    // VALUE = ONES + VALUE (S$ (1: L-1) ) *10
  ENDIF
ENDFUNC VALUE
```

NOTE: **OLD Version 0.11** replace FUNC with PROC and ENDFUNC with ENDPROC.

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 eight 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	=	RELATIONAL	EQUAL	A=B
5	<>	RELATIONAL	NOT EQUAL	A<>B
5	<	RELATIONAL	LESS THAN	A<B
5	>	RELATIONAL	GREATER THAN	A>B
5	<=	RELATIONAL	LESS THAN OR EQUAL	A<=B
5	>=	RELATIONAL	GREATER THAN OR EQUAL	A>=B
5	IN	RELATIONAL	SUBSTRING POSITION	A$ IN B$
6	NOT	BOOLEAN	LOGICAL NEGATION	NOT A
7	AND	BOOLEAN	LOGICAL CONJUNCTION	A AND B
8	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
//
```

APPENDIX G

DIRECTORY - CHANGE FILE TYPE PRG TO SEQ

```
//    directory modify:
//
dim types$ of 15, answer$ of 2, name$ of 17
dim pattern$ of 16
types$:="delseqprgusrrel"
input "disk type: 1: 4040, 2: 8050: ": type
input "drive: ": drive
input "pattern: ": pattern$
track:=18+21*(type=2)
sector:=1
open file 3,"i"+str$(drive),unit 8,15,read
open file 2,"#",unit 8,2,read
repeat
 print file 3: "u1: 2 "+str$(drive)+" "+str$(track)+" "
 +str$(sector)+chr$(13),
 change:=false
 print file 3: "b-p: 2 0"+chr$(13),
 ntrack:=ord(get$(2,1)); nsector:=ord(get$(2,1))
 for i:=1 to 8 do
  print file 3: "b-p: 2 "+str$(2+(i-1)*32)+chr$(13),
  type:=ord(get$(2,1))
  if type<>0 then
   print types$((type mod 128)*3+1:(type mod 128)*3+3);
   print tab(10);
   dummy:=ord(get$(2,2))
   name$:=""
   name$:=get$(2,16)
   name$:+" "
   name$:=name$(1:(" " in name$)-1)
   print name$;
   print tab(40);
   ok:=len(pattern$)=0
   if not ok then
    ok:=pattern$ in name$
   endif
   if type=130 and ok then
    input "change to seq? y{LEFT]": answer$
    if answer$="y" then
     print file 3: "b-p: 2 "+str$(2+(i-1)*32)+chr$(13),
     change:=true
     print file 2: chr$(129),
    endif
   else
    print
   endif
  endif
 endfor i
 if change then
  print file 3: "u2: 2 "+str$(drive)+" "+str$(track)+"
  "+str$(sector)+chr$(13),
 endif
 track:=ntrack; sector:=nsector
until track=0
close file 2
close file 3
```

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
```

```
<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>
```

```
<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> | ' }
```

```
<letter> ::=
            implementation dependent

<digit> ::=
            0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

APPENDIX A

STANDARD BUILT IN FUNCTIONS

```
ABS(<numeric expression>) |
COS(<numeric expression>) |
SIN(<numeric expression>) |
TAN(<numeric expression>) |
ATN(<numeric expression>) |
LOG(<numeric expression>) |
EXP(<numeric expression>) |
SQR(<numeric expression>) |
INT(<numeric expression>) |
EOF(<numeric expression>) |
SGN(<numeric expression>) |
RND[(<numeric expression>)] |
RND(<numeric expression>,<numeric expression>) |
LEN(<string expression>) |
ORD(<string expression>) |
EOD |
ZONE |
VAL(<string expression>) |
STR$(<numeric expression>) |
CHR$(<numeric expression>)
```

COMAL COMMANDS IN AN INTERACTIVE ENVIRONMENT.

```
<command> ::=
            <new command> |
            <run command> |
            <continue command> |
            <list command> |
            <automatic line numbering command> |
            <renumber line command> |
            <delete line command> |
            <store program command> |
            <retrieve program command> |
            <workspace size request>

<new command> ::=
            NEW <eol>

<run command> ::=
            RUN <eol>

<continue command> ::=
            CON <eol>

<list command> ::=
            LIST [<line number range>] <eol>

<line number range> ::=
            <line number> [-[<line number>]] |
            -<line number>

<line number> ::=
            implementation dependent positive integer

<automatic line numbering command> ::=
            AUTO [<line numbering specifier>] <eol>

<line numbering specifier> ::=
            [<start>] [,<increment>]

<start> ::=
            <line number>

<increment> ::=
            <integer>

<renumber line command> ::=
            RENUM [<line numbering specifier>] <eol>

<delete line command> ::=
            DEL <line number range> <eol>

<store program command> ::=
            SAVE <file name> [,<dev info>] <eol> |
            LIST [<line number range>] <file name> [,<dev info>] <eol>
```

```
<retrieve program command> ::=
            LOAD <file name> [,<dev info>] <eol> |
            ENTER <file name> [,<dev info>] <eol>

<workspace size request> ::=
            SIZE <eol>
```

SUGGESTED EXTENSIONS TO COMAL KERNAL.

Extensions to:

```
<numeric assignment> ::=
            <numeric variable> :+ <logical expression> |
            <numeric variable> :- <logical expression>

<string assignment> ::=
            <string variable> :+ <string expression>

<stop statement> ::=
            STOP <string expression>

<simple statement> ::=
            <chain statement> |
            <null statement> |
            <exit statement> |
            <cursor control statement> |
            <clear screen statement> |
            <form feed statement>

<chain statement> ::=
            CHAIN <file name> [,<dev info>]

<null statement> ::=
            NULL

<exit statement> ::=
            EXIT

<cursor control statement> ::=
            CURSOR <row>,<column>

<row> ::=
            <logical expression>

<column> ::=
            <logical expression>

<unstructured declaration statement> ::=
            <local statement> |
            <use statement> |
            <discard statement>

<local statement> ::=
            LOCAL <local declaration> {,<local declaration>} <eol>
```

```
<local declaration> ::=
            <declaration> |
            <numeric identifier>

<use statement> ::=
            USE <package identifier> <eol>

<discard statement> ::=
            DISCARD <package identifier> <eol>

<package identifier> ::=
            <identifier>

<repetitive statement> ::=
            <loop statement> |
            <short repeat statement>

<loop statement> ::=
            LOOP <eol>
              <statement list>}
            ENDLOOP <eol>

<short repeat statement> ::=
            REPEAT <simple statement>
            UNTIL <logical expression> <eol>

<short if statement> ::=
            IF <logical expression> THEN <simple statement>
            ELSE <simple statement> <eol>

<procedure declaration> ::=
            PROC <procedure identifier> <external head appendix> <eo

<function declaration> ::=
            FUNC <function identifier> <external head appendix> <eol

<external head appendix> ::=
            [(<formal parameter list>)] <external specifier>

<external specifier> ::=
            EXTERNAL <string constant> [,<dev info>]

<clear screen statement> ::=
            PAGE

<form feed statement> ::=
            PAGE


Tonder, Denmark, 20 May 1982                Borge R. Christensen
```

APPENDIX I

ADDITIONAL INFORMATION

BOOKS and PAPERS:

COMAL PROBLEMLOSNING OG PROGRAMMERING
by Borge Christensen, Published in Danish
Gerean version published by R. Oldenbourg Verlag, titled:
COMAL 80, DIE STRUKTURIERTE SPRACHE AUF DER BASIS VON BASIC
English version: BEGINNING COMAL
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, England
CAL Research Group Technical Report No. 3

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)
Curriculum Implications of Micro-Electronics Conference, March 1981
by Roy Atherton

MAGAZINES and NEWSLETTERS:

COMAL BULLETIN
Published by Ellis Horwood Limited, England

COMAL CATALYST
Published by COMAL Users Group, Madison, Wisconsin, USA

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)
30 Brancaster Road
Newbury Park
Essex
Ilford IG2 7EP England

INDEX

If you have a Commodore computer
—this book is your complete introduction to COMAL!

COMAL HANDBOOK

Len Lindsay

At last—a book that explains this new structured form of BASIC. Invented in Denmark, this language is for use on the Commodore PET, 8096 and 8032 computers. An easy-to-use reference book—complete with a cross-reference—as well as a source for example programs and procedures (over 100 in all).

Added 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
- And, all other related keywords are cross-referenced.

The CBM COMAL interpreters (both 0.12 and 1.02 versions), 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 $15.00 from Reston Publishing Company, Software Division, 11480 Sunset Hills Road, Reston, VA 22090 — or call (703) 437-8900.

Also for the Commodore Computer . . .

PET BASIC: Training your PET Computer, by Ramon Zamora, William Scarvie and Bob Albrecht

The complete guide to hassle-free PET programming, this book is filled with examples, do-it-yourself exercises, and fun-filled explorations.

PET Games and Recreations, by Mac Oglesby, Len Lindsay, and Dorothy Kunkin

A variety of challenging and entertaining diversions for you and your PET computer. The games are arranged by level of difficulty so that there is something for everyone—adults and children, beginners and computer veterans.

cover design by debbie balboni

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

0-8359-0878-X