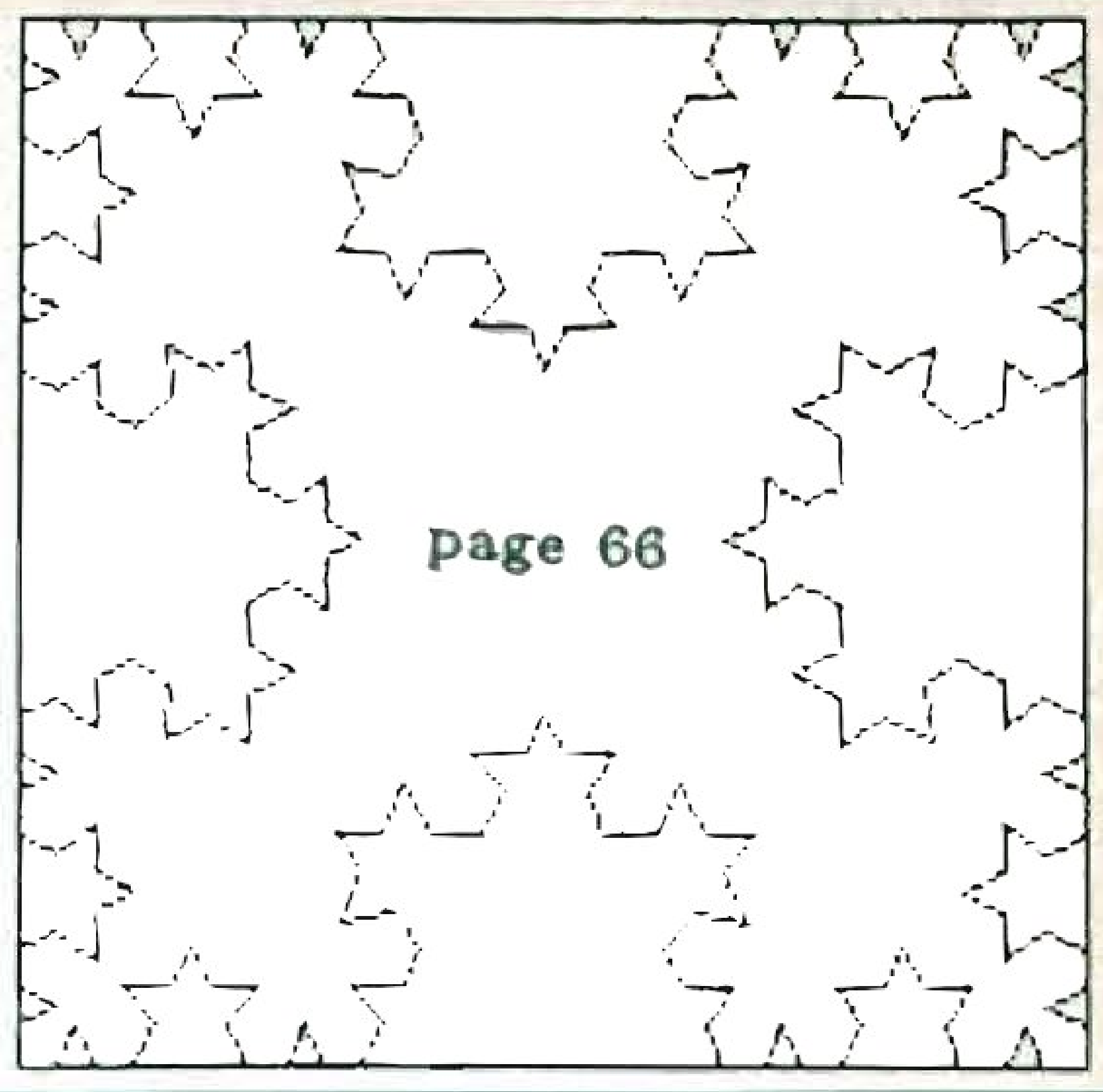
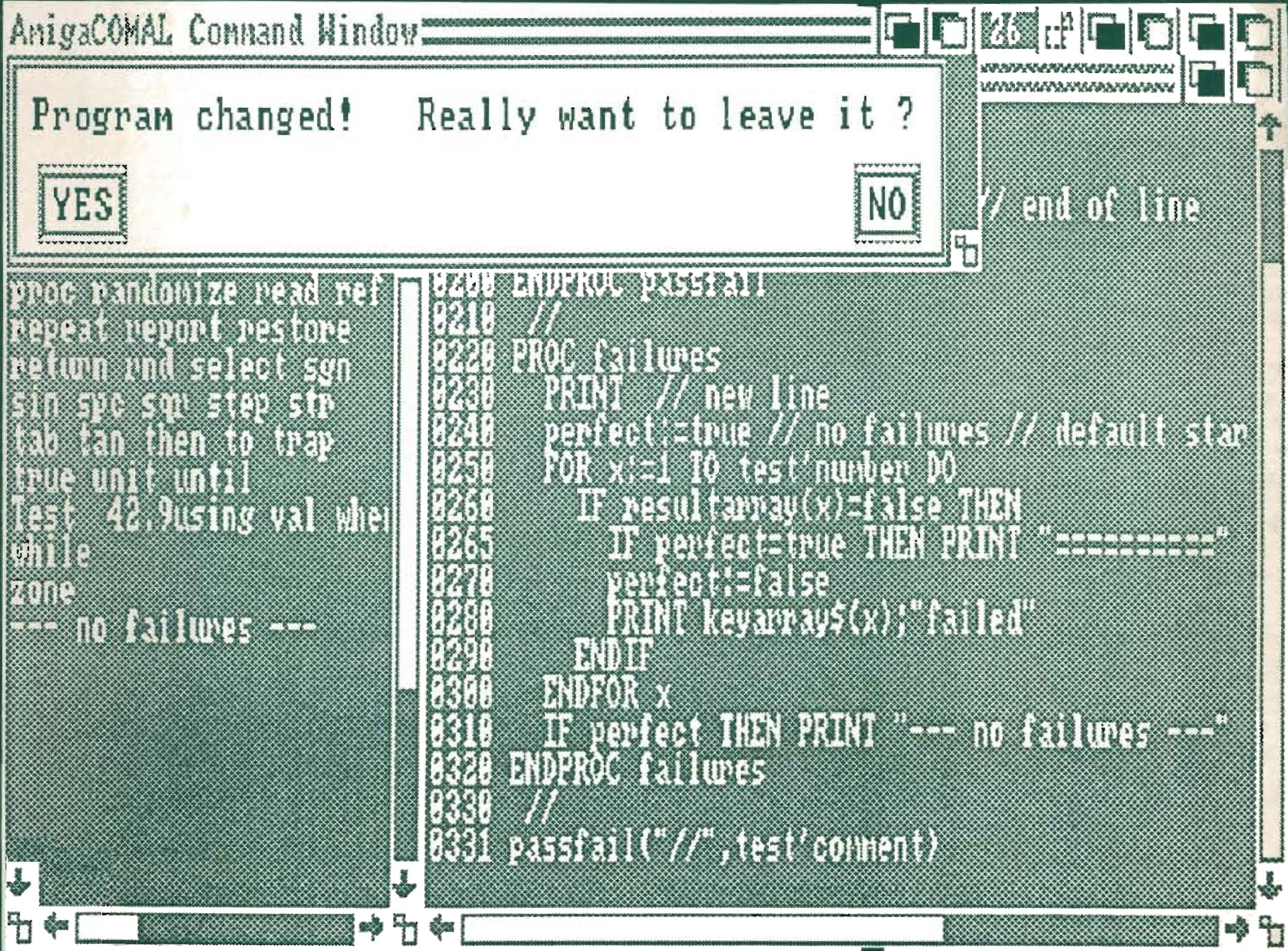
COMAL TODAY 24





```
Program changed!   Really want to leave it ?
```

```
proc randomize read ref
repeat report restore
return rnd select sgn
sin spc sqr step str
tab tan then to trap
true unit until
Test  42.9using val whe
while
zone
--- no failures ---
```

```
                    // end of line
0200 ENDPROC passfail
0210  //
0220 PROC failures
0230   PRINT  // new line
0240   perfect:=true // no failures // default star
0250   FOR x:=1 TO test'number DO
0260     IF resultarray(x)=false THEN
0265       IF perfect=true THEN PRINT "=========="
0270       perfect:=false
0280       PRINT keyarray$(x);"failed"
0290     ENDIF
0300   ENDFOR x
0310   IF perfect THEN PRINT "--- no failures ---"
0320 ENDPROC failures
0330  //
0331 passfail("//",test'comment)
```

AmigaCOMAL
by ComWare

COMAL Today
5501 Groveland Ter
Madison, WI 53716

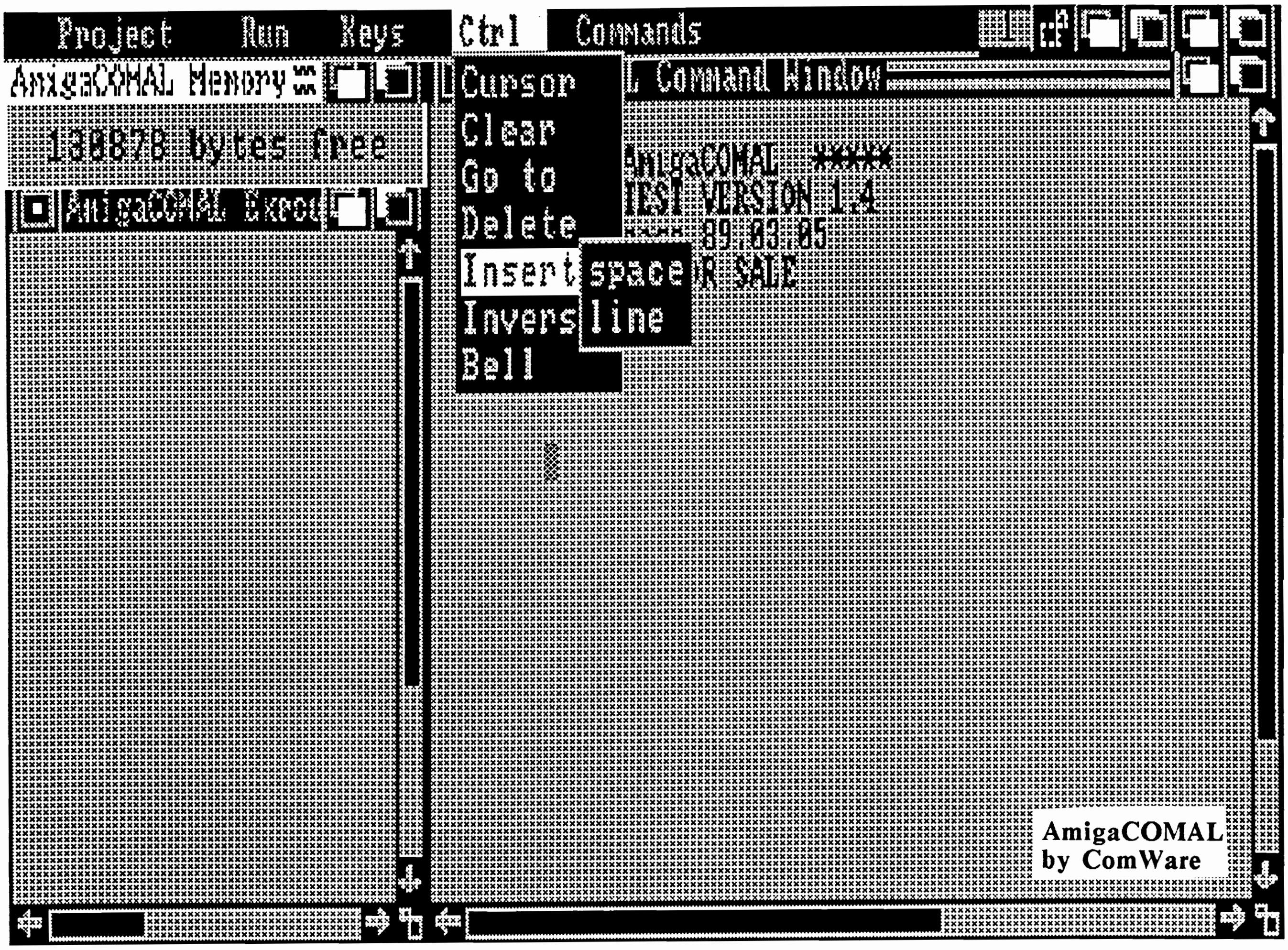
If the label says
Last Issue 24
You must renew now.
Use the order form inside.
No other notice is sent.

Bulk Rate
U.S.Postage
Paid
Madison, WI
Permit 2981

COMMON COMAL™ TEST SYSTEM
page 4

PICTURE PACKAGE
page 61

CHAOS
page 74



AmigaCOMAL: COMAL with the Works.

AmigaCOMAL from ComWare is looking good. I am enjoying testing it. The screen dump above shows how I arranged the startup default screens (plus I pulled down a menu as an example). The <u>install</u> program lets you set up AmigaCOMAL preferences just the way <u>**you**</u> want. I set mine to have the Command Window twice the size of the Execute Window. The windows may overlap if you wish, or even be combined into one window (just like on the C64 and IBM). You also can resize both windows while using COMAL. AmigaCOMAL can use as much free memory as you wish to allocate to it (I chose 128K for the example above). It re-lists your program lines as you enter them (properly indented, UPPER and lower case correct etc).

AmigaCOMAL passed all the Common COMAL tests, and, as you should expect, AmigaCOMAL is very helpful. Use the mouse to place the cursor anywhere on the command screen, or use the scroll bars to move the contents of the screen. Program listings can scroll both up and down on the screen using the cursor keys. It automatically creates the ICON picture for each program you save. Better yet, when you delete a program file, it also deletes the matching ICON. Best of all, when you save a program with a filename that already exists, it renames the existing file with the extension <u>**.backup**</u>, then saves the program.

AmigaCOMAL was written in machine code for maximum speed. A run-time compiler allows you to make your COMAL programs stand-alone. Packages can be written in machine code, C or even COMAL. Some packages written in COMAL were included on my disk, so I could see how they did it. Packages are included for the Intuition, DOS, and EXEC library commands. The authors have been very co-operative with suggestions I have sent them. They even showed me how easy it was to create interesting and useful packages written in COMAL. I did not like one aspect of their built in PASS command. So, they created, in COMAL, a package with the command: CLIPASS. Now I had both types!

Another company, Alder, is working on COMAL for the Amiga too. Their implementation is written in C. They say they have a new release that is faster than the one I tested (see inside cover of last issue), but I have not seen it. Neither have I received an article or notes on it for this issue, as I was expecting. Best I can do is give a screen dump from their previous release (see page 65) showing their edit screen (which is not re-sizable). Note that their command window is line oriented and does not include a full screen editor. A separate program editor must be used (which is integrated into their system). Neither command or edit screen allowed me to use the mouse. However, they have packages for menus and mouse control.

General

2 - Editor's Disk - Len Lindsay
58 - ViewPort - Paul Keck
76 - How To Submit Programs/Articles
76 - How To Type in a Program
77 - Order Form

Amiga COMAL Previews

cover - AmigaCOMAL verifying QUIT option
ifc - Amiga COMAL preview notes
65 - Alder COMAL edit screen/drop menus
75 - AmigaCOMAL screen: CHAOS program
81 - Sample Amiga Startup-Sequence
82 - AmigaCOMAL icons on WorkBench
82 - Sample WorkBench disk contents

Fun & Graphics

59 - Mastermind - Christine & Ray Carter
62 - Power Driver Turtle Parms - R. Hughes
66 - Fractals and Recursion - Bill Inhelder
70 - Graphing a Function and its Derivative Bill Inhelder
74 - CHAOS - Lewis Brown
74 - Cartridge Puzzle Program - Robert Ross

2.0 Packages & Programming

60 - IBM COMAL & Printers- James Landis
61 - C64 Picture Package - Paul Keck
64 - LISTERINE - Will Bow/R. Hughes
67 - Change Drive - R. Hughes
73 - Set Printer Device - R. Hughes

Applications

63 - COMAL to Fortran Translator- Sol Katz
68 - Disk Drive Direct Access - R. Hughes
73 - File Master - Bobby Wallen

Reference

58 - Power Driver Memory Locs - R. Hughes
81 - Filename Conventions

Common COMAL

4 - Test System - Len Lindsay
5 - Problem Areas - Len Lindsay
8 - String Handling Test - Len Lindsay
9 - File Access Test - Richard Bain
11 - Print Using Test - Len Lindsay
12 - Using the Test System - Len Lindsay
21 - Common COMAL Definition and Tests Richard Bain & Len Lindsay
47 - Proposed Multilevel Standard- Joel Rea

Editor	Contributors	Contributors
Len Lindsay	Richard Bain	Paul Keck
	Will Bow	James Landis
Assistant	Lewis Brown	Len Lindsay
Maria Lindsay	Christine &	Joel Rea
	Ray Carter	Robert Ross
Artwork	R. Hughes	David Stidolph
Rhianon Lindsay	Bill Inhelder	Bobby Wallen
	Sol Katz	

Our NEW Address is:

COMAL Users Group, U.S.A., Ltd.
5501 Groveland Terrace
Madison, WI 53716

(608) 222-4432

Editor's Disk

by Len Lindsay

COMAL Today is now produced using Word Perfect 5.0 on our Zenith IBM PC compatible system with our HP LaserJet printer. It has some impressive new features. It does text inside boxes (see page 4) as well as grey scale backgrounds (see page 5). However, we lost easy access to special characters that I like to use. They previously were one key stroke (user defined). Now it takes many keystrokes, plus memorizing a set of two numbers. For example, ■ is produced with control-V, then replying to the prompt with 6,94 followed by RETURN. The « and » are also affected.

Meanwhile, the term Perfect in the name Word Perfect seems to be misleading. We have had (and are still having) unending problems. It caused a delay of two months in our production schedule due to fatal flaws.

For example, I was working on the 40 page Common COMAL Test article, and all of a sudden Word Perfect could only read a small part of the article. No problem. I have a backup. Then it wouldn't read the backup file! But it was a backup of a file that worked fine yesterday! Now I am worried. I then pulled out the week ago backup... same problem. Panic. After a month of mysteriously "*losing*" files and having to start over, in spite of keeping backups and weekly backups, I was very frustrated.

COMAL to the rescue ... well kind of. I wrote a quick COMAL program (COMAL is good for this kind of thing) to read a Word Perfect file. It read and displayed the problem files just fine. Then I modified the COMAL program to duplicate the file, character by character. Fine. However, Word Perfect would not read the new file either.

I contacted Word Perfect. They had me try various things. No luck. So I bought their interim update (it seems they fixed lots of bugs in the 5.0 release, but still called it 5.0). Meanwhile, I could not progress much with my newsletter work. Finally after over 2 weeks my update disks arrived.

I spent a day installing the new 5.0 release, re-creating my hard drive directories, copying files, etc. Then I tried it ... same problems! Actually worse, now I even lost two fonts on my LaserJet. It no longer could print in lineprinter or in `Courier` ... and these are the only two fonts that are not proportional on my LaserJet! I need them for program listings and charts! (for example, see the keyword AND on page 21).

So I went back to using the original 5.0 release. I then split my articles into 1 or 2 page sections, so that if the files mysteriously became unreadable, I only had to redo a couple pages at most. I always have kept lots of backups, and that continues, even though many backups somehow are unreadable by Word Perfect 5.0.

Amiga & LaserJet

Within the past few days I finally got my LaserJet printer hooked up to my Amiga. I have to study it further, but did get a couple graphic screen dumps that are fairly readable. Anyone have a way to do better dumps on a LaserJet? Let me know.

Amiga Desktop Publishing?

Resolving problems with Word Perfect 5.0 were placed on hold until this newsletter went to press. Then I will also look into producing *COMAL Today* on my Amiga, perhaps with PageStream or such. Anyone with ideas on this please let me know.

Power Driver Update Postponed

David Stidolph spent a lot of time updating Power Driver. The new release was called C64 COMAL 1.0 (or 1.11 in its final stage). We had preliminary releases uploaded to QLink for COMALites to try. A few bugs were reported, such as with the INPUT statement. All were corrected and it was just about ready for release, when the NEW command bug was discovered.

David spent months analyzing the thousands of lines of source code and could not find nor fix the NEW bug. Richard Bain even made a special trip to Madison to help David, but the bug was not found. Therefore, the project has been postponed. David decided it would be better to start over from scratch (ie, the Power Driver source code).

Due to my many problems with Word Perfect, it would take too long to re-do all the pages in this issue that refer to C64 COMAL 1.0. Please note that it is not available yet.

ComWare AmigaCOMAL

Correction: German Amiga COMAL now has a real name! It is **AmigaCOMAL** from **ComWare**, a Danish company. Its main programmer lives near Copenhagen in Denmark. Borge Christensen, father of COMAL, also is a Dane and assisting in the project. The incorrect German reference came about due to one of the people helping with the project living in Germany (near the Denmark border). ComWare AmigaCOMAL is supposedly being distributed in Scandinavia and Germany. However, they wish to delay releasing it in the

USA and Canada until it has been in use a few months in Europe. They told me "autumn 1989" as the expected USA/Canada release.

If you want to be informed when it becomes possible to order AmigaCOMAL, just send me a Self Addressed 25¢ Stamped Envelope with a note to save it for the AmigaCOMAL announcement. I have a special folder I am keeping these envelopes in. Thanks to all who previously sent envelopes in: they are in the folder patiently waiting final news.

Mytech is now Alder

The USA branch of Mytech has undergone changes. Its principal owner apparently has moved to Mexico. The others have renamed the company to Alder. Their original release of COMAL for the Amiga was slow and had some problems. They said they have a new release now that is faster and corrects some of the problems (like initializing arrays when they are DIMmed). They said they would try to send a copy to us here, but as of this date we have not received it. Note, they have a temporary address while they are re-organizing: Alder COMAL, c/o Mark Evans, 108 Hiram Ave, Newbury Park, CA 91320. Phone 805-498-6533 (noon-7pm California time).

Apple

Apple COMAL 1.0 should be released by the time you read this. This first release will run on 64K Apples and will not have a graphics capability. A 2.0 version is being planned for 128K Apples. It will have more features. For information about Apple COMAL contact David Stidolph, COMALites United, 1670 Simpson #102, Madison, WI 53713.

UniComal IBM PC COMAL

No recent news from UniComal. They rumored that they are working on a COMAL 3.0 for the IBM that will have records and pointers.

ComWare AmigaCOMAL

Speaking of records and pointers. ComWare AmigaCOMAL has records and pointers! It looks like they may have beat UniComal to the punch, in Europe at least. AmigaCOMAL is looking very professional. Being able to write packages in COMAL is a fantastic capability. Some of the packages included on the disk were written in COMAL! This allowed me to see how they were written. Then, the RUNTIME COMPILER allows a COMAL program to be converted to a standalone file. This COMAL seems very promising.

QLink

We continue to have two national meetings each month on QLink at 10pm Eastern Time. They are held every <u>first</u> Sunday and every <u>second</u> Thursday of the month. We have our very own COMAL section on QLink. It includes a conference room for our meetings as well as a message base for announcements and questions (usually answered within a couple days). We also have two libraries of COMAL programs you can download. Here is how to get to our COMAL section on QLink:

```
CIN (Commodore Information Network)
    Computing Support Groups
        Programmers Workshop
            COMAL
                Message Board & Libraries
                    Q & A Message Board
                    COMAL 0.14 files library
                    COMAL 2.0 files library
                Conference Room
```

Once you choose COMAL, you can either go to the Message Board / Libraries or to the Conference Room. Note that our conference room is <u>not</u> in the Conference section of QLink ... but is inside our own section, exclusively for our use!

Amiga

Warning! If you are switching to the Amiga from another computer, be prepared. To be used seriously, the Amiga <u>requires</u> 2 drives. One drive is used for the Work Bench disk. With the C64, the computer operating system was in ROM, and the disk operating system was in ROM inside the disk drive. However, the Amiga relies on a disk for much of its operation, and that can lead to unending frustration. It can't even display a disk directory without accessing its WorkBench disk! So, get the second drive right at the start and save the aggravation. And while you are at it, make sure you have at least 1 meg of memory. Many programs are requiring it now ... and you will have far fewer GURU crashes if you have more memory. Next I think I may get FACC II to speed up disk access. If you have suggestions to share with others, drop me a line.

Amiga BOOT disk

Finally, make sure you have a good "boot" disk. It took me months to get a fairly acceptable startup sequence! Now, I'm not saying that my startup disk is perfect by any means, but if you get a new Amiga, change the <u>startup-sequence</u> file inside the <u>S</u> subdirectory! I think enough of you will be using Amiga computers to make it worth while for me to print some info on getting it set up right. Rather than put it here, see page 81 for more info.

COMMON COMAL Test System

Article by Len Lindsay
Test procedures & functions by Richard Bain
Syntax style by Borge Christensen

One advantage of COMAL is that the many implementations try to be compatible with each other. This is possible due to an existing COMAL standard, agreed on by members of the companies producing COMAL systems. Along this same line, we wish to refine an up to date common ground, a COMMON COMAL.

As new COMAL systems appear, they are extending COMAL. This good, if current standards are not changed in the process. A current standard COMAL program should run on the new COMAL systems. A temptation to the COMAL system developers is to do something "better". If better means not according to the standard, it is worse. If we loose compatibility between systems, we loose the one thing that can eventually boost COMAL into the limelight.

There are a few trouble areas in COMAL and the various implementations. They are covered in detail in a separate article.

Likewise, a separate article explains the beginnings of our COMMON COMAL Test Program, a collection of procedures and functions to test if a COMAL implementation meets the COMAL standard. We want to extend this idea, so please send us your input! Send us your routines that you think test a specific aspect of COMAL. We also need a few more test programs for specific areas of COMAL.

Finally, Joel Rea proposes a three level COMAL standard. Joel has done a lot of work for COMAL, and I hope you seriously consider his ideas.

After going over all this, please consider how we should keep COMAL standardized. Help us define a COMMON COMAL and methods of testing implementations.

NOTE: The new C64 COMAL 1.0 has been postponed till Fall. Months of debugging could not fix a fatal flaw in the NEW command. It now will have to be rewritten from scratch.

NOTE: AmigaCOMAL is now being released in Scandanavia and Germany. We previously called it German Amiga COMAL. It really is a ComWare product written mainly in Denmark. It should be available in the USA by Fall.

Common COMAL is a trademark of COMAL Users Group, U.S.A., Limited.

References for COMMON COMAL

Common COMAL Reference by Len Lindsay
This book goes over the COMMON COMAL keywords, noting syntax with examples, plus providing sample programs showing the keyword in use. A brief summary of each keyword is given along with notes about various COMAL implementations.

COMAL Handbook by Len Lindsay
This is one of the main references for COMAL. It is specifically covering both C64 COMAL systems (0.14 and 2.0). The manuscript was used as the manual for UniComal IBM PC COMAL. Amiga COMAL used it as its point of reference in maintaining a compatible COMAL. However, it is now out of print (withdrawn from the market by its publisher, Reston Publishing). It is replaced by *Common COMAL Reference*.

COMAL From A To Z by Borge Christensen
This is a mini-reference book for the original C64 COMAL 0.14 by the founder of COMAL himself.

COMAL Standard - COMAL Kernal
Printed in full in *COMAL Today #17*, this gives the full text of the COMAL Standard, along with associated notes and even a special program.

COMAL Info Booklet
This is the 24 page booklet we send to people who ask for information about COMAL. It tells about COMAL and includes a four page COMMON COMAL chart. Syntax and example is given for each keyword. It also has one page showing the common structures and a one page chart showing which COMAL implementations follow each of the COMMON COMAL keywords.

Note: Mytech COMAL for the Amiga is now available as Alder COMAL for Amiga. They told us that a new revision was just released that was faster. However, we have not yet received a copy to verify this. Their temporary address is: Alder COMAL, c/o Mark Evans, 108 Hiram Ave, Newbury Park, CA 91320. We also were told that order and tech help would be available by phone at 805-498-6533 (prefer noon-7pm California time).

COMMON COMAL - Problem Areas

by Len Lindsay

There are a few problem areas within the COMAL Standard. I will present some of them here. Then, in a separate article, I will define the COMMON COMAL standard. Your comments are welcome. If you see other areas of conflict that need clarification by addition to the COMMON COMAL definition, please send us your notes.

KEY$ - no key pressed

KEY$ looks for the next key in the keyboard buffer, but does not wait for a key to be pressed (INKEY$ does that).

If there is one keystroke in the keyboard buffer, that key is returned. If more than one key is waiting, the first one is returned. If keys cde are pressed, only c would be returned by KEY$.

What should COMAL do if no key has been pressed? KEY$ is not in the Kernal, but is part of COMMON COMAL and is included in all implementations (see *CT#18* back cover). The KEY$ example in our *COMAL Info Booklet* is:

```
WHILE KEY$="" DO NULL
```

This example waits for a key to be pressed, and implies that if no key is pressed, a null string "" is returned. However, the *COMAL Handbook* states that C64 COMALs both return CHR$(0) if no key has been pressed. The *COMMON COMAL Reference* says this: "*If no key is pressed Commodore COMALs return a CHR$(0). The other COMAL's return a null string.*"

The new Apple and Amiga COMALs return a null string "" if no key is pressed. The Commodore COMALs were the only ones that did not return the null string, but returned a CHR$(0) instead. However, the latest disk loaded COMAL for the C64 (COMAL 1.0) now has KEY$ return the null string like other systems. So only C64 Power Driver and C64 2.0 cartridge do not follow the COMMON COMAL standard:

If no key is pressed, KEY$ returns the null string "".

"" IN "abc" - null with IN

IN is a handy operator. It searches one string to see if it contains another string. If it finds the other string, it returns the position that it starts at. If it is not found, it returns 0. But, what does COMAL do when the first string is the null string? (For being a nothing, the null string sure can cause problems).

IN is part of the COMAL Kernal, which says: "*If «string1» is evaluated to the null string, then the value returned is 1.*" By this definition, "" IN "abc" is 1. Alder does this.

The *COMAL Handbook* says: "*If the length of «string1» is 0 (null string) then the value returned will be the length of «string2» plus 1.*" By this definition "" IN "abc" is 4. CP/M COMAL, C64 Power Driver and C64 2.0 cart do this.

The *Common COMAL Reference* agrees with the *COMAL Handbook*. There is a note included with the Kernal, on page 46 of CT#17 that says: "*A proposal is currently submitted that specifies that the returned value should be false (zero) [if «string1» is the null string].*"

So, to end the confusion, COMMON COMAL specifies that 0 is returned if «string1» is the null string. IBM PC COMAL, Amiga COMAL, Apple COMAL, C128 COMAL cart and the new C64 COMAL 1.0 do this.

Why? Because if the first string is nothing, it can't be in the other! Therefor, 0 must be returned. For example, if your program asks a question, and then uses IN to see if the reply was part of the valid responses, just hitting «return» (ie, null string) should not pass the test of a valid response, otherwise programming problems are introduced:

```
PROC file'action
  REPEAT
    PRINT "Should we delete the file?"
    INPUT "Your choice (Y or N): ": reply$
  UNTIL reply$ IN "YyNn"
  IF reply$ IN "Yy" THEN
    delete'file
  ELSE
    archive'file
  ENDIF
ENDPROC file'action
```

This example procedure would be called after a file was chosen. It would determine what action to take with the file (either delete or archive). A valid reply to the DELETE prompt is only Y, y, N, or n. We want to keep asking until we get a valid reply (a nice REPEAT UNTIL loop). Just hitting «return» is not a valid choice, and should be ignored. In our example, if COMAL returns a 1 or 5 (length of "YyNn"+1) and then 1 or 3 (length of "Yy" plus 1) for a null string reply, it is disastrous. Our file is deleted! If 0 were returned when reply$ was the null string, the

REPEAT loop would once again ask the question, as expected.

Now, mathematicians would say that the null string is part of every set. However, a reply to a question is not a set. It is a text string.

Actually, the argument could go on for years (actually, it already has). What we need to do is just set a common sense standard (return 0 if «string1» is the null string) and ask that COMAL systems follow it. Right now, Amiga COMAL, Apple COMAL, C64 COMAL 1.0, C128 COMAL and IBM PC COMAL already meet this COMMON COMAL standard:

"" IN text$ always returns 0.

DIV and negative numbers

DIV is part of the COMAL Kernal, which defines x DIV z as INT(x/z). The *COMAL Handbook* agrees with this definition. However, CP/M COMAL and C64 COMAL 2.0 do not follow this standard with negative numbers. For example:

7 DIV (-3) should be -3
(-7) DIV 3 should be -3

But CP/M COMAL and the C64 COMAL 2.0 cart give a result of -2. Amiga COMALs, Apple COMAL, C64 Power Driver, IBM PC COMAL and the new C64 COMAL 1.0 give the proper result.

COMMON COMAL maintains the same definition as the COMAL Kernal:

x DIV z means INT(x/z).

MOD and negative numbers

MOD is part of the COMAL Kernal, which defines x MOD z as x-(x DIV z)*z. Since this definition uses DIV, problems with negative numbers with DIV carry over into MOD.

COMMON COMAL abides by the Kernal definition, which can have the DIV section expanded to yield: x-INT(x/z)*z. Note that the Kernal states that z must be positive or the result is undefined (may vary with implementation). However, the definition still works and yields an answer, in all cases, though it may be irrelevant when z is negative.

Actually, MOD means modulo, not remainder. UniComal apparently is considering implementing REM as a remainder function, in addition to MOD (which would explain why they do not convert REM into // for comment). To understand modulo, and how it adheres to our definition, we can use a clock analogy.

For example, the problem 7 MOD 3 gives the answer 1. This can be demonstrated on a clock. MOD 3 means our clock will have only 3 numbers around it. Start with 0 as the first digit (same as you would for decimal, 0 through 9). 0 is on top of the clock, go clockwise to add the numbers 1 and 2 equally spaced around the clock. Now to find the answer of 7 MOD 3 just start at the 0, then count clockwise 7 digits (positive numbers mean clockwise): 1 .. 2 .. 0 .. 1 .. 2 .. 0 .. 1. Notice that you end up on the 1, which is your answer.

The other three variations with MOD involve making either 7 or 3 or both negative. In our clock analogy, negative means counterclockwise.

For -7 MOD 3 use the same clock as the previous example, but count seven times counterclockwise, since the seven is negative. You should end up on the 2, which is your answer.

For 7 MOD -3 put numbers around the clock like this: 0 at the top, then go counterclockwise and put in the numbers -1 and -2. Next, start at the top, then count clockwise (since 7 is positive) seven times. You should end up on the -2, which is the answer.

For -7 MOD -3 use the same clock as the previous example, but since the seven is negative, count counterclockwise seven times. You should end up on the -1, which is the answer.

x MOD z means x-(x DIV z)*z or x-INT(x/z)*z.

PRINT separators ; and ,

Several items may be printed on one line. Print separators are used to separate the items. The separator tells COMAL what to do after printing one item before going on to the next item. COMAL allows two types of separators, a semicolon (;) and a comma (,).

COMMON COMAL - Problem Areas

In COMAL, a print line is divided into zones of equal width. The start of each zone is sort of a tab stop (as on a typewriter). The keyword ZONE is used to set the zone width.

Originally, the default zone was 0 for no zones, and the comma was used to mean space to the next zone (therefor no spaces by default). The semicolon (the other separator) was defined as causing a one space separator. This was handy for printing a line of numbers so that there were spaces between numbers. This was followed by early COMAL implementations, including C64 COMAL 2.0 cartridge, C64 Power Driver, CP/M COMAL, the original IBM PC COMAL and Alder COMAL.

The problem was how to issue several characters with no spaces between them, regardless of the zone setting (null separator)? This is necessary to issue commands to printers, modems, etc. With the original definition, a program had to first save the original zone setting, change it to 0, print the items, then reset the zone back to its original setting. Examples of this coding commotion is seen in sample procedures and functions in *COMAL Handbook*, Appendix D.

Modifications to the standard were needed so that a null separator could be guaranteed. We proposed that the ! be used as a null separator. Granted, the PRINT line would look funny, but it would not affect any current programs. Others did not seem to like our proposal.

Another solution was implemented by UniComal. It was to flip the meaning of the ; and , . The comma was defined as a null separator, unaffected by the zone setting. The semicolon gave spaces to the next zone ... and the default zone was set to 1, thus by default the semicolon gave a one space separator. So, in all programs that did not set their own zone with the ZONE statement, there was no change. However, programs that had user defined zones now must use a semicolon rather than a comma for the separator. This was implemented in the C128 COMAL 2.0 cartridge and the latest UniComal IBM PC COMAL.

Granted, the change is a major improvement ... putting things the way they should have been at the start. But it made some existing programs run incorrectly. For this reason we were against it.

Now, the new Amiga COMAL is following UniComals print separator definition. Thus, three implementations (Amiga, IBM and C128), now follow the new separator definition. Meanwhile, both Apple COMAL and C64 COMAL 1.0 were being finalized. The developer decided to use the new definition. Thus there now are five systems following the new definition. And it definitely is the better definition. Therefor, our COMMON COMAL definition now supports the comma as a null separator and the semicolon as the zone separator.

Regrettably, this means some early implementations now do not meet the COMMON COMAL standard. However, both early C64 COMALs were produced by UniComal, who instigated this change. And most programs do not specify a zone, and will run unchanged under the new definition:

**Comma is a null print separator.
Semicolon is a zone separator.
The default zone is 1.**

Summary

Four popular implementations meet all the above COMMON COMAL standards! They are: Amiga COMAL, IBM PC COMAL, C64 COMAL 1.0, and Apple COMAL. Special thanks to Steve Kortendick for help with the DIV and MOD sections. COMMON COMAL is a trademark of COMAL Users Group, U.S.A., Limited.

Change CAT / DIR Device#

This procedure allows you to change the default device used by CAT and DIR.

```
// change device# used by cat/dir
PROC cat'dev(dv)
  IF PEEK(27013)=101 THEN // Power Driver
    POKE 28446,dv
  ELSE  // original comal 0.14
    POKE 27013,dv
  ENDIF
ENDPROC cat'dev
```

COMMON COMAL - String Handling Tests

by Len Lindsay

This program tests COMAL string handling. It is not an exhaustive test, but briefly checks various COMAL methods. Please submit any tests you think should be added, or a better version of these.

I'll mix my comments with the program. However, you must type it in as one program, not as several small programs.

First we dimension two string variables, s$ to a maximum of 10 characters, and t$ to a maximum of 4 characters.

```
DIM s$ OF 10, t$ OF 4
```

Next we test substring assignment. Here we specify that we wish to assign characters 1 through 7 of the variable s$. However, we only provide 5 characters. COMAL should pad the rest with spaces.

```
PRINT "=====testing substring assignment:"
s$(1:7):="abcde"
IF LEN(s$)=7 THEN
  PRINT "correct length"
  IF s$(7:7)=" " THEN
    PRINT "correct padding with spaces"
  ELSE
    PRINT "failed padding with spaces"
 ENDIF
ELSE
  PRINT "failed - wrong length"
ENDIF
```

Now we will try to assign 5 characters to t$, which has previously been dimensioned to hold a maximum of 4 characters. COMAL should truncate our assignment, accepting only the first 4 characters, since that is the maximum set for t$.

```
PRINT "=====testing auto truncating assignment"
t$:="abcde"
IF t$="abcd" THEN
  PRINT "passed"
ELSE
  PRINT "failed"
ENDIF
```

COMAL substrings are specified by stating the character to start at and the character to end at, separated by a colon. This specification is included in parentheses after the string name.

```
IF t$(2:3)<>"bc" THEN PRINT "Failed"
```

Now we will see if the short notation for substrings is accepted. (2:) means start at character 2 and go through to the end of the string. The long form of this is (2:LEN(t$)) where t$ is the name of the string.

```
PRINT "=====testing substrings"
t$:="abcd"
s$:=t$(2:)
IF s$="bcd" THEN
  PRINT "passed (2:)"
ELSE
  PRINT "failed (2:)"
ENDIF
```

Now we will check another short notation form for substrings. (:3) means start at the first character and go through the third character. The long form for this would be (1:3)

```
s$:=t$(:3)
IF s$="abc" THEN
  PRINT "passed (:3)"
ELSE
  PRINT "failed (:3)"
ENDIF
```

Now we will test that we can take a one character substring. To do this we must specify the start and end character as the same character number.

```
s$:=t$(2:2)
IF s$="b" THEN
  PRINT "passed (2:2)"
ELSE
  PRINT "failed (2:2)"
ENDIF
```

Now we test that COMAL will insert a character into a current string, without affecting the rest of the string. This can be done with more than one character as well. Remember that t$ is still equal to "abcd" as set previously in the program.

```
PRINT "=====testing substring inserting"
t$(2:2):="x"
IF t$="axcd" THEN
  PRINT "passed"
ELSE
  PRINT "failed"
ENDIF
```

Now we check that strings can be concatenated (added together). We check both methods:

```
PRINT "=====testing string concatenating"
t$:="abcd"
s$:=t$+t$
IF s$="abcdabcd" THEN
  PRINT "passed"
ELSE
  PRINT "failed"
ENDIF
s$:="z"
s$:+t$
IF s$="zabcd" THEN
  PRINT "passed"
ELSE
  PRINT "failed"
ENDIF
```

COMMON COMAL - File Access Tests

Original program by Richard Bain
Modified by Len Lindsay

COMAL has a set of File I/O keywords. They provide various capabilities to COMAL for reading and writing sequential and random files. This program will test some of these features. Please let us know if you have further modifications to this program so that it will be a better test.

I will mix my comments about the program throughout the listing. The program is intended to be typed in as one large program. Don't run just one section, even though it appears to be broken up due to my comments.

This program will use two files. Their names are set up as variables to make it easy for you to change if needed. I have assigned very strange names to the two filenames, so they should not conflict with any of your files! Another thing the program will do is select a file as the output location, then return the output to the screen later. I have used a variable (screen$) to hold the id for the screen, so it can be changed for IBM PC COMAL. Temp$ and temp2$ are used to hold text strings for various tests. Reply$ is used for questions to get a one character reply, or just to wait for the user to tell the program they are ready. Text$ and text2$ are used later in the program for passing large blocks of bytes from one file to another. (C64 Power Driver and C64 COMAL 1.0 must add a drive specification to the beginning of each file name, example: "0:uqtestzp.dat")

```
DIM filename$ OF 20, filename2$ OF 20, reply$ OF 1
DIM temp$ OF 40, temp2$ OF 40, screen$ OF 5
DIM text$ OF 999, text2$ OF 999 //for compare test
filename$:="uqtestzp.dat"; filename2$:="uqtestzu.dat"
screen$="ds:" //<==<== all COMALs but IBM
//screen$:="con:" //<==<== IBM only
```

Next we clear the screen and warn the user about the filenames we will be using. The INPUT statement will use the current row for its prompt starting at column 1 and only allow one character to be typed.

```
PAGE
PRINT "This program tests several COMAL"
PRINT "commands relating to file access."
PRINT "It uses temporary disk files"
PRINT filename$;"and";filename2$
PRINT "Place a NON-write protected"
PRINT "disk in the current disk drive."
PRINT
INPUT "<return> to start:": reply$
```

Before we start, make sure the files are not on the disk. COMAL shouldn't complain if we try to delete a file that does not exist.

```
PAGE
PRINT "==> making sure files";filename$;
PRINT "and";filename2$;"are deleted...";
DELETE filename$
DELETE filename2$
PRINT "OK"
```

Next we create a random file with 10 records, each with a maximum size of 40 bytes (or characters). With many disk operating systems, it is advisable to create your random file first, and then fill in the records. This also is a faster method in many systems.

```
PRINT "==> creating";filename$;"as random file..."
CREATE filename$,10,40
```

Next we open the random file for use. The record length (40) must be the same as the one used by the CREATE statement. Any time the file is accessed, the OPEN statement must specify a record length of 40. Also note that this means 40 bytes of data. If you write a string of 40 characters to a record it would not fit, because with WRITE FILE the binary representation of a string includes a length counter byte(s) and with PRINT FILE there are delimiters that must also be written.

```
PRINT "==> opening";filename$;"as random file..."
OPEN FILE 7,filename$,RANDOM 40
```

Next we will write ten records to the random file, but write them in reverse order. We use the WRITE FILE statement here. Later, to check on them, we will have to use the READ FILE statement. Each record is a string followed by a number.

```
PRINT "==> writing 10 records to";filename$;"...";
FOR x:=10 TO 1 STEP -1 DO
  PRINT x;
  WRITE FILE 7,x: "line",x
ENDFOR x
PRINT "OK"
```

Now we read back the records, in sequential order this time, making sure they are the same as what we wrote.

```
PRINT "==> Reading the 10 lines back..."
FOR x:=1 TO 10 DO
  READ FILE 7,x: temp$,line
  PRINT temp$;line;
  IF temp$<>"line" OR line<>x THEN
    CLOSE
    END "error in RANDOM file"
  ENDIF
ENDFOR x
PRINT "OK"
```

Now we close the file. A simple CLOSE with no file number specified should close all open files, and not complain if there aren't any files to close.

```
PRINT "==> closing the file..."
CLOSE
```

Now we delete the file.

```
PRINT "==> deleting file";filename$
DELETE filename$
```

Now we select a file as the output location. Anything that normally would print on the screen will now be directed to the file. (C64 Power Driver and C64 COMAL 1.0 do not allow SELECT to a file, so lines marked *** must be changed, see special listing following the one below)

```
PRINT "==> redirecting output to file";filename$
SELECT OUTPUT filename$ // ***
```

Print a few strings; all should go to the file. Then the PAGE command should output a CHR$(12).

```
WHILE NOT EOD DO
  READ temp$ //from data statements
  PRINT temp$ //goes to the file now    ***
ENDWHILE
PAGE //should be chr$(12) for form feed now ***
```

Switch the output back to the screen.

```
SELECT OUTPUT screen$ //     ***
PRINT "==> output back to screen"
```

*** Special program lines for C64 Power Driver and C64 COMAL 1.0:

```
PRINT "==> file redirection not possible"
OPEN FILE 7,filename$,WRITE // ***
WHILE NOT EOD DO
  READ temp$
  PRINT FILE 7: temp$ // ***
ENDWHILE
PRINT CHR$(12) // ***
CLOSE FILE 7 // ***
```

Next we will open the file we just wrote from the previous section in APPEND mode. Then we will write a number at the end of the file.

```
PRINT "==> opening file";filename$;"for append..."
OPEN FILE 4,filename$,APPEND
PRINT "==> writing to file..."
number#:=7
PRINT FILE 4: number#
```

Now we close the file, specifying its file number.

```
PRINT "==> closing file..."
CLOSE FILE 4
```

Next we restore the data pointer back to the beginning, so we can read the same data that we previously used when writing the file. Then we open the file to read it back.

```
RESTORE
PRINT "==> opening file";filename$;"to read..."
OPEN FILE 2,filename$,READ
```

Now we check that the items in the file are the same as in the data statements.

```
PRINT "==> reading from file..."
WHILE NOT EOD DO
  READ temp$ //from data statements
  INPUT FILE 2: temp2$
  PRINT temp2$;"...";
  IF temp$<>temp2$ THEN
    CLOSE
    END "PRINT to file or INPUT FILE failed"
  ENDIF
ENDWHILE
PRINT //cr at line end
```

Now we check that the PAGE command put a CHR$(12) into the file.

```
PRINT "==> GET$ checking Form Feed from PAGE...";
IF GET$(2,1)<>CHR$(12) THEN //form feed
  CLOSE
  END "PAGE or GET$ failed."
ENDIF
PRINT "OK"
```

Now we make sure that the number we appended to the file is there. Since it was output to the file with PRINT FILE, it is in ASCII form. Therefore, we can input the number as a text string now.

```
PRINT "==> reading back integer as a string...";
INPUT FILE 2: temp2$
PRINT temp2$
IF temp2$<>"7" THEN
  CLOSE
  END "PRINT number#/INPUT FILE as string FAILed"
ENDIF
```

Now we make sure that the EOF has been set for our file, since we just read the last item in the file. Then we close the file.

```
PRINT "==> checking for EOF flag set...";
IF NOT EOF(2) THEN
  CLOSE
  END "End Of File not found"
ENDIF
PRINT "OK"
CLOSE FILE 2
```

Now we check that GET$ can get several characters at once. To do this we restore the data pointer back to the beginning, and open the file to read again. Then we read a data item and compare it to a GET$ from the file. Then close the file.

File Access - conclusion

```
RESTORE
PRINT "==> checking GET$...";
OPEN FILE 2,filename$,READ
READ temp$ //from data statements again
temp2$:=GET$(2,10)
CLOSE
IF temp2$<>temp$(1:10) THEN
  END "GET$ failed"
ENDIF
PRINT "OK"
```

Now we will try two files open at once by copying one file over to another. First we open both files.

```
PRINT "==> Open 2 files at once:";
PRINT "One READ - One WRITE - (file copy)"
OPEN FILE 1,filename$,READ
OPEN FILE 2,filename2$,WRITE
```

Now, with just one statement we copy the file. PRINT FILE must be used (not WRITE FILE), and the statement must end with a comma (null separator) so that an extra CHR$(13) is not added to the end of the file. Then we close the files.

```
PRINT FILE 2: GET$(1,999), //copy whole file
CLOSE
```

To see if the copy worked, we open both files again. This time we use GET$ to get the contents of both files. Then we close the files.

```
PRINT "==> checking if file copy worked...";
OPEN FILE 1,filename$,READ
OPEN FILE 2,filename2$,READ
text$:=GET$(1,999); text2$:=GET$(2,999)
CLOSE
IF text$<>text2$ THEN
  END "Failed."
ENDIF
PRINT "OK"
```

Now we can delete the files off the disk.

```
DELETE filename$ //done with it
DELETE filename2$ //done with it
```

Print the I/O commands we just tested. Advise that all tests passed. If a test had failed, the program would have ended at that part of the program.

```
PRINT "APPEND, READ, WRITE, RANDOM,";
PRINT "OPEN, CLOSE, SELECT, CREATE,"
PRINT "PRINT FILE, INPUT FILE, GET$,";
PRINT "WRITE FILE, READ FILE, EOF"
INPUT "==> all tests passed. press <return>:": reply$
```

Here are the data statements used.

```
DATA "Mary had a little lamb"
DATA "Jack and Jill"
DATA "Happily ever after"
```

PRINT USING Test

by Len Lindsay

This program tests some aspects of PRINT USING. In order to do this automatically, I use files. In the first file I write what the output from the USING is supposed to be. The second file is used for the output from the PRINT USING statement itself. Later, I compare both files to make sure that they are identical. Please submit any further tests you think should be added to this test, or correct any flaws you find in this program.

```
DIM filename1$ OF 20, filename2$ OF 20
DIM reply$ OF 1
DIM text1$ OF 40, text2$ OF 40
filename1$:="uqtestzp.dat"; filename2$:="uqtestzu.dat"
PRINT "PRINT USING TEST"
PRINT "This program uses two disk files"
PRINT "that it creates, uses, then deletes."
PRINT filename1$;"and";filename2$
PRINT "place blank formatted disk into"
INPUT "current drive. Hit return when ready:": reply$
//
DELETE filename1$
DELETE filename2$
//
OPEN FILE 1,filename1$,WRITE // correct answer goes here
OPEN FILE 2,filename2$,WRITE // print usings go here
//
PRINT FILE 1: "test 120.0 test"
PRINT FILE 2: USING "test ###.# test": 120
//
PRINT FILE 1: "test   5.47 test"
PRINT FILE 2: USING "test -##.## test": 5.467
//
PRINT FILE 1: "test  -5.47 test"
PRINT FILE 2: USING "test -##.## test": -5.467
//
PRINT FILE 1: "test ******* test"
PRINT FILE 2: USING "test ###.### test": 12345
//
PRINT FILE 1: "test   3  $ 55.00 test"
PRINT FILE 2: USING "test ###  $###.## test": 3,55
//
CLOSE
//
PRINT "comparing the files now"
PRINT
OPEN FILE 1,filename1$,READ
OPEN FILE 2,filename2$,READ
passed:=TRUE //init
WHILE NOT (EOF(1) OR EOF(2)) DO
  INPUT FILE 1: text1$
  INPUT FILE 2: text2$
  PRINT text1$
  PRINT text2$
  PRINT "=========="
  IF text1$<>text2$ THEN passed:=FALSE
ENDWHILE
CLOSE
DELETE filename1$
DELETE filename2$
IF passed THEN
  PRINT "====> All passed"
ELSE
  PRINT "====> Failed"
ENDIF
```

Using the COMAL Test System Functions

by Len Lindsay

The COMAL Test System is a set of functions that test how well a COMAL implementation meets the COMMON COMAL standards. The test functions presented in this issue were originally the idea and work of Richard Bain. I added to them and enhanced them. **You are welcome to submit test functions that you devise to add to this test system. Send them in with a short note explaining what each one is testing. We will try to expand the test system so that it becomes more complete.**

Another way to test COMAL is with separate test programs. Examples are in this issue. One program tests PRINT USING in more detail, and another program tests the area of string handling. A third program tests COMAL file handling. You are invited to submit small test programs like these to add to our set of COMAL tests (such as parameters, recursion, local/global, assignment).

A final note: let us know if you find any flaws in our tests, or if any of them can be improved. Thank you.

Now, back to the topic: how to use the test functions that are presented immediately following this article.

Especially For Beginners

This article is directed mainly at beginning COMAL programmers. The more advanced can skim over most of it.

Skip a few pages ahead to find the article COMMON COMAL - Definition and Test Functions. That article lists each keyword in our Common COMAL standard along with its syntax and an example of it in use. It then briefly describes the keyword after categorizing it (Statement, Function, Operator, etc). Finally, many of the keywords have a test function presented inside a box, along with a caption that explains some of what it is testing.

To get you started, let's go over a couple of the test functions. You may need two book marks. One to mark your place in this article. The other to mark your place in the Test Functions article.

I will try to place special emphasis in this article on the beginners point of view. I will make mistakes in this article, so you can see how to recover from them, or prevent them altogether. Most articles present the correct way to accomplish something, but do not show the many mistaken methods tried before coming up with the correct way. Don't feel bad if you make mistakes. Everyone does. The perfect articles and programs cover up all the mistakes made during their preparation.

To start off, make sure COMAL is running in your computer. This usually involves booting up the COMAL system from disk. C64 and C128 cartridge owners only need to plug in the cartridge and turn on the computer. (In this article I will often refer to the C64 cartridge; the C128 cartridge is usually identical in these tests).

Many of the COMAL boot disks also load in a "HI" program automatically at start up. Sometimes this will be a menu or a welcome message. Before you start now, make sure that any such program is erased from the programming memory. Type this:

```
new
```

You can check to make sure that you have an empty programming area. Type this:

```
list
```

Your cursor should return with nothing listed. Oh, by the way, you can use UPPER or lower case letters when typing your COMAL commands and programs (except with C64 COMAL 0.14 and PET COMAL 0.14 which require unshifted letters ... Power Driver removed the restriction).

Let's start with the very first test function. It is for // (how do you pronounce //?). // is used in COMAL to signal the start of a comment. BASIC uses REM for this purpose (REMark), and some COMALs will convert REM into // for you (to help you in transition from BASIC to COMAL). Other BASICs use ! to signify the start of a comment, and some COMALs convert ! into // for you as well.

Testing // seems quite futile. The only way to test it is to use it. If COMAL accepts the comments, fine. If not, an error should come up. Use AUTO for automatic line numbers. Type:

```
auto
```

Now, COMAL will provide line numbers for you, beginning at 10, in increments of 10. You just type in the program lines, hitting <return> at the end of each line. Actually, you do not have to be at the end of a line to hit <return>. As long as your cursor is anywhere on a line, hitting <return> asks COMAL to accept the line. COMAL first checks if the line is a direct command or part of a program. If it begins with a line number (1-9999), it is considered a program line. Otherwise COMAL assumes it is a direct command (like the AUTO command you originally typed).

After you type AUTO, COMAL responds by printing the first line number for you (0010), followed by one blank space, and a blinking cursor, waiting for you to type. You don't need to capitalize the keywords, even though they are capitalized in our listings. COMAL will do that for you later.

Just for fun, let's make a few mistakes to see how COMAL handles it. First of all, let's use the name **test** rather than **test'comment** as the function name. It is shorter and will make the following exercises easier to follow. Also, try typing **fun** rather than **func** (we will SEE what happens when you drop the C ... what fun!). What you type is underlined:

```
0010 fun test closed
```

As soon as you hit <return>, COMAL checks the line just entered. It sees that it begins with a line number, so it tries to accept it as a program line.

It finds the first word, **fun**. This is not recognized as a COMAL keyword, so it must be a variable or procedure name. So far so good.

Next COMAL sees the word **test**. Now it has a problem. If **fun** is a variable, an equal sign "=" should come next for an assignment. If **fun** is a procedure name, parameters can come next, but must be preceded by a parentheses "(". Or, it is possible to have two procedure calls on one line separated by a semicolon ";" but that is not the case here either. **fun test** does not meet any of COMALs syntax rules.

Since COMAL does not understand **fun test** it cannot accept the line. Precisely what happens now will vary from COMAL to COMAL. But all COMAL systems should reject a faulty line! This is one of COMALs strong points... real time error checking on line entry.

COMAL will provide an error message, and put the cursor back on the line you typed at the point where it thinks the problem starts.

Amiga COMAL will pop up an error message window with the message. After you fix the error, the error message window will disappear. (Sorry, but my Amiga is in the repair shop, so I can't provide details on the Amiga COMALs.) Other COMAL systems will print an error message directly below the line being questioned. After the line is corrected they erase the error message, and restore to the screen what was originally there. (Now that I have experienced it, I rather like the error window popping up and disappearing in AmigaCOMAL.)

If you are using C64 Power Driver or C64 COMAL 1.0, your screen would look like this:

```
auto
0010 fun test closed
syntax error
```

The cursor is blinking on the first **t** in the word test. That is the location where the line ceased to follow proper COMAL syntax. The message given is **syntax error**. This is a pretty generic message. Most often, this will mean that you misspelled a word, forgot a parentheses or space, or some other typo.

C64 cartridge users get a better set of error messages built into the cartridge. The cursor still blinks on top of the first **t** in the word **test**, but the message under the line is:

```
":=" or "(" expected, not name
```

Of course, neither of the suggestions apply in our case, because we merely misspelled the word **FUNC** as **fun**.

IBM PC COMAL responds with the same message as the C64 cartridge (not surprising, since UniComal wrote both implementations).

CP/M COMAL has a similar message:

```
Error number 38: ":=" or ":+" or ":-" expected
```

Hey, that's right. COMAL allows a variable to be incremented or decremented. **count:+1** would add one to the variable **count**. So, if **fun** were a variable, it could also have :+ or :- come next on the line.

Phew! You are lucky. You just have one COMAL to follow. I have been running around several rooms, each with computers running different implementations of COMAL (my Amiga 1000 is in the repair shop, so I only have an empty spot on one desk where it would have been). And in addition to all the COMALs running, I also have my Word Perfect up and running, allowing me to type in this article as I actually type in the COMAL lines on the other computers.

Now, fix the word **fun** so it correctly is **func** and make sure that a space comes after it before the word **test** ... like this:

Using the COMAL Test System Functions

On the C64, move the cursor to the space after the word fun and press <shift>+<inst> to insert a space. Then type c and the word func is corrected.

With CP/M COMAL, place the cursor on the space after fun and press <ctrl>+<inst>. This opens up one space just like on the C64.

With IBM PC COMAL, press the <ins> key to go into *"insert mode"*. Then put the cursor on the space after fun and type c. Notice the rest of the line moved over to make room for the c, and the word func is correct. You remain in "insert mode" until you hit <return> or press the <ins> key again.

With Amiga COMAL <shift>+<cursor right> puts you into *"insert mode"*.

Now, don't hit <return> yet. Go to the end of the line and erase the d at the end of the word closed. Some COMALs have a quick way to go to the end of a line. IBM PC COMAL... press the <end> key. C64 cartridge... type <ctrl>-L. Amiga COMAL allows you to use your mouse... point at the d in closed and press the select mouse button... your cursor is placed at the d.

Now, after erasing the d, hit <enter>. The line is still incorrect, and all versions of COMAL will detect this. Here is what the screen looks like in Power Driver or C64 COMAL 1.0:

```
auto
0010 func test close
syntax error
```

The cursor is blinking on the c in close. COMAL did make sense out of the first two words (to define a function named test). Now it has a problem when it sees the keyword close, which is used to close a disk file. It does not belong at the end of a function definition header line!

The C64 cartridge once again is a bit more specific (even helpful this time):

```
"CLOSED" or "EXTERNAL" expected, not "CLOSE"
```

IBM PC COMAL gave a message just like the cartridge (as we expected), but did not include the "" marks around the keywords:

```
CLOSED or EXTERNAL expected, not CLOSE
```

Now see what happens if you type a space on top of the c that your cursor is blinking over and then hit <return>. Try it (turn close into lose).

The syntax error message remains with Power Driver and C64 COMAL 1.0. But the others change messages to adapt to new possibililties (before CLOSE was a keyword in the wrong place, now lose is a name out of place).

The C64 cartridge changes its message to:

```
"CLOSED" or "EXTERNAL" expected, not name
```

IBM PC COMAL again is the same (without the "" marks):

```
CLOSED or EXTERNAL expected, not name
```

CP/M COMAL gives this:

```
error number 13: "(" expected
```

In our case, the C64 cartridge and IBM PC COMAL provided us with the best clue. They asked us to check if we really wanted the word CLOSED at the end of the line. And that is indeed what we want.

Fix the word closed at the end of the line, and hit <return>.

In the C64 your screen should now look like this:

```
auto
0010 func test closed
0020 _
```

The error messages are removed from the screen and the cursor is blinking just underneath the f.

AmigaCOMAL, CP/M COMAL and IBM PC COMAL provide another service for you. They "re-list" the line on the screen for you with keywords capitalized, variables and procedure/function names in lower case, and other improvements. IBM and CP/M versions also put the cursor on the next line at the correct indentation level, ie under the n in func:

```
auto
0010 FUNC test CLOSED
0020   _
```

(This is for the latest IBM PC COMAL release, the previous ones did not re-list the lines). With Power Driver and C64 COMAL 1.0 you can type rem or ! (exclamation point) in place of // for beginning a comment. Later when you list the line, the rem or ! will be gone, and // will be in their place.

IBM PC COMAL will instantly convert ! into // for you as it relists the line, but will not accept rem. The C64 cartridge will accept ! but not rem, and will convert it when you list the line later (this makes it easy to make a current program line a comment ... just put the cursor on teh space immediately after the line number, and type a !) CP/M COMAL will not accept rem, and allows you to type ! but gives a "*not implemented*" error when the line is executed.

Now type in the rest of this four line function ... but take a shortcut on the final line. Just type endfunc and hit <return>. No need to type in the function name, COMAL inserts it for you later!

After you finish typing in the four line function, stop COMALs auto mode. To do this from Power Driver or C64 COMAL 1.0, just hit <return> on the blank line 0050 prompt. Amiga and CP/M COMAL hit the <esc> key. C64 cartridge hit the <stop> key. IBM COMAL type <ctrl>+<break>.

Now list your function to make sure it is right:

```
list
0010 FUNC test CLOSED
0020   // this is a comment
0030   RETURN TRUE // comments allowed
0040 ENDFUNC
```

Notice, ENDFUNC with no name following it. However, it will be capitalized even if you typed it in lower case, since it is a keyword. COMAL implementations automatically capitalize keywords for you. COMAL also will automatically indent the lines in any structure. Thus you see the lines inside our function definition are indented for us (two space indent on systems with 80 column screens, and one space indent on 40 column screens such as Power Driver). You do not have to type in the leading spaces you see in our listings. COMAL will insert them automatically for you when listing your program lines.

Now, you can utilize another feature of COMAL. It can scan over an entire program to make sure that all your structures are proper. For example: it checks that each FUNC later in the program has a proper ENDFUNC to match it. Any structure errors will be reported. By the time you actually RUN a program, COMAL will have checked it over pretty well! Try it. Type:

```
scan
```

If you corrrectly typed everything in, all will be fine, and the cursor returns right away. During its scan of the program, COMAL will insert the function names after each ENDFUNC, procedure names after each ENDPROC, and FOR variable names after each ENDFOR. List your program and see:

```
list
0010 FUNC test CLOSED
0020   // this is a comment
0030   RETURN TRUE // comments allowed
0040 ENDFUNC test
```

Before going on, quickly test the COMAL SCAN command. Delete line 40. Then try a SCAN:

```
del 40
scan
```

An error message appears. With Power Driver or C64 COMAL 1.0, this is the message:

at 0030: error in structured statement

As you might expect, the C64 cartridge is a bit more specific:

at 0030: "ENDFUNC" missing

IBM PC COMAL is similar (minus the "" marks of course), but does not give the line number:

ENDFUNC missing

CP/M COMAL gives this message:

Error number 55: Uncomplete structure

In any case, each COMAL informed you that you have a problem with your program structures. So, SCAN was useful.

In fact, all COMALs do a SCAN of every program before actually beginning execution. Try it. Type:

```
run
```

You got the same error as with SCAN didn't you! COMAL will not run a program that is not structured properly. This is another big plus in COMAL programming!

Now, let's put back line 40 the easy way. It is still visible on your screen. If you can see it you can enter it (except with Alder COMAL which is line oriented unless you are inside their program editor ED). Just cursor up to the line 40 as listed on the screen and hit <return>. Your cursor can be anywhere on the line! With Amiga COMAL you also my use your mouse to point to any point on line 40 and then press the select mouse button. The cursor appears at the location you pointed to. (I am getting to like this mouse!)

We finally are ready to try out our first test function. Move your cursor back down to where we were. Type: list and you will see that all is back to normal. Type: scan and that should be fine now too. Ready? Type:

```
run
```

And there you have it. Nothing. Power Driver, C64 COMAL 1.0 and the C64 cartridge just say:

```
end at 0040
```

IBM PC COMAL says:

```
Ready
```

CP/M COMAL says:

```
End of program
```

They all tell us that the program is finished. So how do we know if the function passed? Shouldn't it tell is something?

Look at our program. We have defined a function that we named test. However, we never actually used that function in the program... only defined it. So, it is good that nothing happened, since we did not ask for anything. (In BASIC, if your subroutines were at the end of your program, and you did not have an END statement before them, the program would "fall through" and execute them even if they were not called. That was bad.)

So, what do we do to call our test function? There are many things we can do. With COMAL, we can call it from direct mode. Just type this:

```
test
```

Ooops. That must not be it. That is how you call a procedure from direct mode. Not a function. COMAL didn't know what to do. Maybe you misspelled a command or procedure name? The message you get varies between systems. Power Driver and C64 COMAL 1.0 give this message:

```
command, array, substring, or procedure error
```

Both the C64 cartridge and IBM PC COMAL are more specific:

```
test: Not a procedure
```

CP/M COMAL simply says:

```
Error number 40: Unknown procedure/function
```

Actually, what we did is call a defined function incorrectly. A function is designed to return a value to the calling statement. The calling statement must do something with that value. We did not specify what to do with the value.

The easiest thing to do is to print the value. Type this:

```
print test
1
```

All the COMALs respond identically. They each print 1 as the value returned. This is correct, since line 30 in the function says to return TRUE, and true is equal to 1 (later on in the COMMON COMAL definition article, TRUE is defined).

Here is another way to test out our test function. Type this:

```
if test print "true
true
```

COMAL is very helpful. We left off the end " on the print statement and it took care of it for us. We also left out the word THEN, but COMAL took care of that for us as well. The result now is the word true is printed on the screen. All COMALs do this. The proper line would be:

```
IF test THEN PRINT "true"
```

Or, even further:

```
IF test=TRUE THEN PRINT "true"
```

Or, more precisely:

```
IF test<>FALSE THEN PRINT "true"
```

OK, // passed the test. Now we go on to the next keyword in our COMMON COMAL definition article: ABS. ABS stands for Absolute and returns the absolute value of a number.

Let's create a few more error situations now. Our first 4 lines are still in the computer (lines 10, 20, 30 and 40). To type in the next function using COMALs automatic numbering, type:

```
auto
```

Power Driver, Apple COMAL and C64 COMAL 1.0 start by prompting with line 10 if a specific line is not given. At this time, this is not good, since if we enter a line 10 it will overwrite our existing line 10. We want to add to the other function, not overwrite it. Fortunately, the 0010 line number is highlighted in reverse video to draw our attention to the fact that there already is a line 10 defined in memory. Just hit <return>

to stop auto mode. Start it at line 50 (the next line in our program sequence): auto 50

All the other COMALs automaticaly start the auto line numbers just following the last number currently in memory. Thus they would properly start at 0050.

Now, we will provide another error situation for COMAL to deal with. Type in the next test function also using the name test rather than test'abs (note: you may omit the keyword THEN as COMAL will add it for you later):

```
0050 func test closed
0060 if abs(1)<>1 return false
0070 if abs(-1)<>1 return false
0080 if abs(0)<>0 return false
0090 if abs(-3.9)<>3.9 return false
0100 return true
0110 endfunc
0120      <stop auto mode here>
```

Now try out a test on this new function. Type:

```
print test
```

C64 Power Driver and C64 COMAL 1.0 tell you:

```
program not prepassed
```

IBM PC COMAL and the C64 cartridge say:

```
program has been modified
```

CP/M COMAL says:

```
Error number 61: SCAN necessary
```

CP/M COMAL hit it on the head. We must SCAN (or RUN) a program before we can call any procedures or functions from direct mode. Each time you add lines, delete lines, or change lines in a program you must then SCAN it before you can call procs/funcs from direct mode.

OK, so do it:

```
scan
```

Now COMAL finds a problem with the program, since we have defined two different functions with the same name! C64 Power Driver and C64 COMAL 1.0 say:

```
at 0050: error in structured statement
```

IBM PC COMAL and the C64 cartridge say:

```
at 0050: test: name already defined
```

CP/M COMAL gives this message:

```
Error number 44: name already exists in 0050
```

OK, we get the picture. We can't have two functions with the same name. Well, actually, with EXTERNAL procedures we can. But we will get to that later. Now, let's see what we can do with our two functions so they work together.

First, lets store them on disk. No use retyping them later. Find a formatted disk with some empty space on it. Or you can format a disk directly from COMAL (the way to do this will vary from system to system since COMAL tries to follow the computers operating system).

First, refresh our memory by listing the program in the computer now:

```
list
0010 FUNC test CLOSED
0020   // this is a comment
0030   RETURN TRUE // comments allowed
0040 ENDFUNC test
0050 FUNC test CLOSED
0060   IF ABS(1)<>1 THEN RETURN FALSE
0070   IF ABS(-1)<>1 THEN RETURN FALSE
0080   IF ABS(0)<>0 THEN RETURN FALSE
0090   IF ABS(-3.9)<>3.9 THEN RETURN FALSE
0100   RETURN TRUE
0110 ENDFUNC test
```

Now you can save the whole program if you like:

```
save "testprog"
```

Next, list each function to disk so it can be recalled later... merged in with other programs:

```
list 10-40 "comment.lst"
list 50-110 "abs.lst"
```

Those two functions can be recalled later via the ENTER or MERGE commands.

Now, let's have some fun with EXTERNAL. If your version of COMAL does not have EXTERNAL capability, skip past this section (C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not have EXTERNAL).

COMAL 2.0 Only:

EXTERNAL is an interesting concept. Briefly, it means that the whole body of a function or procedure may be stored external to the main program, yet the main program may execute the external procedure or function as if it were actually part of the main program! Phew! That was a long winded sentence.

Using the COMAL Test System Functions

All that is needed to use the external function or procedure is the header for it, ending with the keyword EXTERNAL followed by the file name used to store it on disk. It is actually easier to show you it in action than to explain it. So let's do it!

First, procedures or functions that will be used as EXTERNAL later must be stored on disk via the SAVE command (not LIST). Also, to qualify for use as EXTERNAL, a procedure or function must be CLOSED and may not use IMPORT statements. Different COMAL implementations have various "extra" things that you can do with the procedure or function that you store externally, such as including lines before or after it. However, these are extensions, and we will avoid them for now.

We already LISTed each function to disk. Now we need to SAVE each individual function to disk. To do this, the function has to be the only one in memory. So, we just erase everything, then enter in each function previously stored on disk, save it to disk, then clear memory and do it again for the next one:

```
new
enter "comment.lst"
save "comment.ext"
new
enter "abs.lst"
save "abs.ext"
new
```

Actually, the first two new commands are not needed, since all COMALs that support EXTERNAL also automatically do a NEW before beginning an ENTER command.

Now you have two functions stored on disk in two ways. In ASCII from the LIST to disk command. And also in binary (tokenized) from the SAVE command just used.

LISTed to disk, functions may be merged into other programs later with the MERGE command (or the ENTER command with Power Driver, C64 COMAL 1.0 and Apple COMAL).

SAVEd to disk, the functions may be used as EXTERNAL functions. That is why we included the suffix of .ext at the end of the filename. This will remind us that the file can be used as an EXTERNAL function!

Now for the fun. We will write a short program that will utilize the two functions as EXTERNAL (this can be continued for many more). We will also show how the name of the function in the file does not have to match the name we give it in our new program that uses it externally (that is why we can have two functions named TEST both used in the same program ... if they are external functions).

Let's write a new program. Type:

```
new
auto
0010 print test'comment
0020 print test'abs
0030 //
0040 func test'comment external "comment.ext"
0050 func test'abs external "abs.ext"
0060          <stop auto mode here>
```

That was a relatively simple and short program. We just print the value returned by the two external functions (lines 10 and 20). Lines 40 and 50 are the full function definitions for our program. Remember that the body of each function is stored on disk. COMAL takes care of how it is executed.

Now, save this program to disk before you try it. It is always a good idea to save programs before running them... insurance against problems!

```
save "ext-test"
```

Ready. Try it:

```
run
1
1
```

Yay! It worked. Both functions returned a value of TRUE. They both passed the test, and EXTERNAL worked as well! Plus both functions have the same name on disk, but allowed us to call them by another name in our test program.

Amiga & UniComal 2.0 Special:

Amiga, C64 cartridge, C128 cartridge and IBM PC COMAL users can utilize a special extension of EXTERNAL. You can use a variable name as the filename with each EXTERNAL definition. This sounds like a minor point. No way. Try this:

```
new
auto
0010 while not eod
0020 read keyword$
0030 print test
0040 endwhile
0050 //
0060 func test external keyword$+".ext"
0070 //
0080 data "comment","abs"
0090          <stop auto mode here>
```

Using the COMAL Test System Functions

By using a variable name in line 60, we can substitute various functions on disk for the one definition! Yes, that one external function definition will suffice to test every one of the test functions in the COMMON COMAL definition and test article! Just put the keywords as part of the data statements. COMAL will redefine the function each time it is called using the current value of keyword$. Try it!

Now, let's get all users back to this article. EXTERNAL users know how to do external functions now. You can adapt each of the functions to be used in this manner.

Everyone Back ... All COMALs

OK, fun and games is over! Let's fix those first two functions so they are properly named (test'comment and test'abs). At the same time we will illustrate one important thing that Power Driver, C64 COMAL 1.0 and Apple COMAL users need to remember when merging program segments. Fix the first function.

COMAL 2.0 users can issue a CHANGE command to save time:

```
new
enter"comment.lst"
change "test","test'comment"
0010 FUNC test CLOSED
 <<line 10 is displayed, test is highlighted>>
```

Hit <return> and test is changed into test'comment. Hit <n> and it is not changed. (In CP/M COMAL the found text is not highlighted and you hit <space> to not change it).

Other COMAL users, fix it like this:

```
new
enter"comment.lst"
list
0010 FUNC test CLOSED
0020   // this is a comment
0030   RETURN TRUE // comments allowed
0040 ENDFUNC test
```

Now cursor up the screen and insert 'comment in both places test appears. Or do this just for the first one. After the ENDFUNC just erase the word test (type spaces over it). COMAL will insert the correct name when you do a SCAN.

```
scan
list
0010 FUNC test'comment CLOSED
0020   // this is a comment
0030   RETURN TRUE // comments allowed
0040 ENDFUNC test'comment
```

Everyone now ... store the function to disk:

```
delete "0:comment.lst"
// ibm & cp/m use a: or b: in place of 0:
// Amiga use DF0: or DF1: in place of 0:
list "comment.lst"
```

Notice that specifying a drive will vary from computer to computer, since COMAL generally abides by the way the computer identifies the drives. Commodore 64 uses 0: and 1: while IBM and CP/M use a: and b:. Amiga uses DF0: and DF1:. Apple COMAL will have a table of drive ids, and will be able to convert from those of the other systems into Apple computer format.

Now do the same thing for the other function, changing test into test'abs. Remember to LIST it back to disk too.

OK. Now, let's merge the two functions from disk together into one program.

With COMAL 2.0, use the MERGE command:

```
new
enter "comment.lst"
merge "abs.lst"
```

COMAL automatically renumbers the abs.lst function as it is entered. Thus you could have each of the test functions on disk in LISTed format, and they could be merged into one program easily.

Other COMAL users, use the ENTER command to merge segments. Note that COMAL does not renumber lines as they are merged with the current program. If lines coming in from disk have the same line number as those already in the computer, they will overwrite the lines in the computer. To see this happen type a simple program line, number it line 10. Then ENTER the first test function you LISTed to disk. LIST the results and notice that your line 10 is gone... replaced by the line 10 in the test function:

```
new
10 print "test line"
enter "comment.lst"
list
0010 FUNC test'comment CLOSED
0020   // this is a comment
0030   RETURN TRUE // comments allowed
0040 ENDFUNC test'comment
```

Notice that line 10 from disk replaced the line 10 you typed in just prior to the ENTER command.

To avoid this problem, C64 Power Driver, C64 COMAL 1.0 and Apple COMAL users should make sure procedures and functions LISTed to disk have high line numbers (line 9000 or higher). Then when ENTERed with another program, the line numbers will not conflict (unless the other program is very very big ... and even then, if it is first renumbered by 1, there will not be a problem).

Use the RENUM command to renumber the test function:

RENUM 9000

Then delete the old file, and LIST it back to disk with the high line numbers (remember to use disk drive specifications to match your computer system):

delete "0:comment.lst"
list "0:comment.lst"

Now, issue a new command and do the same thing with the abs test function.

Everyone now:

Now you have seen how the 2.0 COMALs have a few luxury features not available to C64 Power Driver, C64 COMAL 1.0 and Apple COMAL (MERGE, CHANGE, and EXTERNAL). All three of these are part of the COMMON COMAL standard, so you can see that without them, some COMALs are only a subset of the full COMMON COMAL. However, their programs will be upward compatible with the 2.0 COMALs.

Now you are ready to type in each test function. You can type them in individually and store them on disk to merge together later. Or 2.0 users can store them as external type functions and write a short program that calls them as needed. Or, type them all as part of one large program.

All the individual test functions can be put together into one large test program in many ways. In the next column I illustrate one way to do it. A full program with all the test functions accessed in this manner is on Today Disk 24, and available by special request on Amiga or IBM PC disks.

COMMON COMAL is a trademark of COMAL Users Group, U.S.A., Limited.

```
max'tests:=255
DIM keyarray$(1:max'tests) OF 10
DIM resultarray(1:max'tests)
test'number:=0
//
PROC passfail(keyword$,test'result)
  test'number:+1 //increment number of tests
  keyarray$(test'number):=keyword$
  resultarray(test'number):=test'result
  IF test'result=FALSE THEN
    PRINT // new line
    PRINT keyword$;"failed"
  ELSE
    PRINT keyword$;
    IF CURCOL>70 THEN PRINT // end of line
  ENDIF
ENDPROC passfail
//
PROC failures // print all keywords that failed
  PRINT // new line
  perfect:=TRUE // no failures // default start
  FOR x:=1 TO test'number DO
    IF resultarray(x)=FALSE THEN
      IF perfect THEN PRINT "=========="
      perfect:=FALSE
      PRINT keyarray$(x);"failed"
    ENDIF
  ENDFOR x
  IF perfect THEN PRINT "--- no failures ---"
ENDPROC failures
//
passfail("//",test'comment)
FUNC test'comment CLOSED
  // this is a comment
  RETURN TRUE // comments are allowed
ENDFUNC test'comment
//
passfail("abs",test'abs)
FUNC test'abs CLOSED
  IF ABS(1)<>1 THEN RETURN FALSE
  IF ABS(-1)<>1 THEN RETURN FALSE
  IF ABS(0)<>0 THEN RETURN FALSE
  IF ABS(-3.9)<>3.9 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'abs
//
passfail("and",test'and)
FUNC test'and CLOSED
  IF (TRUE AND TRUE)<>TRUE THEN RETURN FALSE
  IF (TRUE AND FALSE)<>FALSE THEN RETURN FALSE
  IF (FALSE AND TRUE)<>FALSE THEN RETURN FALSE
  IF (FALSE AND FALSE)<>FALSE THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'and
//
passfail("at",test'at)
FUNC test'at CLOSED
  row:=CURROW; col:=CURCOL
  PRINT AT 10,30: "",
  IF CURROW<>10 THEN RETURN FALSE
  IF CURCOL<>30 THEN RETURN FALSE
  PRINT AT 0,20: "",
  IF CURROW<>10 THEN RETURN FALSE
  PRINT AT 4,0: "",
  IF CURCOL<>20 THEN RETURN FALSE
  CURSOR row,col
  RETURN TRUE
ENDFUNC test'at
// etc. etc. ...
failures
```

//

// anything typed here

Statement - Anything after // is ignored, allowing comments in programs. In direct mode, // allows you to overtype on a full screen line. After your command, just type // and hit return; the rest of the line is ignored, and your command executed.

```
FUNC test'comment CLOSED
  // this is a comment
  RETURN TRUE // comments allowed
ENDFUNC test'comment
```

1. Tests single line comment and comment at the end of a statement.

ABS

ABS(«numeric expression»)
PRINT ABS(standard'number)

Function - Gives the absolute value of the number. Positive numbers and zero are unaffected, while negative numbers become positive.

```
FUNC test'abs CLOSED
  IF ABS(1)<>1 THEN RETURN FALSE
  IF ABS(-1)<>1 THEN RETURN FALSE
  IF ABS(0)<>0 THEN RETURN FALSE
  IF ABS(-3.9)<>3.9 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'abs
```

2. Tests positive, negative, zero and non-integer values.

AND

«expression» AND «expression»
IF number>0 AND number<100 THEN

Operator - Gives the result of a logical AND of two expressions, as shown by the following table:

```
AND   || TRUE  | FALSE
======|===============
TRUE  || TRUE  | FALSE
------|+-------+------
FALSE || FALSE | FALSE
```

This is different than most BASICs in which AND is a bitwise operator. For bitwise AND in COMAL see BITAND.

```
FUNC test'and CLOSED
  IF (TRUE AND TRUE)<>TRUE THEN RETURN FALSE
  IF (TRUE AND FALSE)<>FALSE THEN RETURN FALSE
  IF (FALSE AND TRUE)<>FALSE THEN RETURN FALSE
  IF (FALSE AND FALSE)<>FALSE THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'and
```

3. Tests all four AND operation possibilities.

APPEND

OPEN [FILE] «file#»,«filename»,APPEND
OPEN FILE 2,"test",APPEND

File Type - Part of the OPEN statement. The sequential file must already exist on disk, and is opened in APPEND mode. New data is written to the file immediately after the existing data.

AT

PRINT AT «row»,«col»: [«print list»[«mark»]]
INPUT AT «row»,«col»[,«len»]: [«prompt»:] [«vars»[«mark»]]
PRINT AT 1,1: "Section number:"; num;
INPUT AT 10,1,1:"Yes or No? ":reply$

Special - Part of INPUT or PRINT statements, specifying a specific location to start at, similar to having a CURSOR statement immediately before a PRINT or INPUT statement. PRINT AT may also be combined with a USING format. Remember, the cursor location is specified row then column, similar to finding your seat at a theater. Including a comma or semicolon at the end of the statement causes the cursor to remain on the current line, and not go down to the next line. A comma means stay where it is, while a semicolon means space to the next zone then stay there (default is one space).

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not accept a comma at the end of an INPUT AT statement. They do accept a semi-colon.

```
FUNC test'at CLOSED
  row:=CURROW; col:=CURCOL
  PRINT AT 10,30: "",
  IF CURROW<>10 THEN RETURN FALSE
  IF CURCOL<>30 THEN RETURN FALSE
  PRINT AT 0,20: "",
  IF CURROW<>10 THEN RETURN FALSE
  PRINT AT 4,0: "",
  IF CURCOL<>20 THEN RETURN FALSE
  CURSOR row,col
  RETURN TRUE
ENDFUNC test'at
```

4. Tests row and column placement, then column only, then row only.

ATN

ATN(«numeric expression»)
PRINT ATN(num1+num2)

Function - Returns the arctangent in radians of the number. *CP/M COMAL allows you to choose degrees rather than radians, but has radians as the default.*

```
FUNC test'atn CLOSED
  IF ATN(0)<>0 THEN RETURN FALSE
  IF ABS(ATN(1)-PI/4)>0.000001 THEN RETURN FALSE
  IF ABS(ATN(TAN(0.5))-0.5)>0.000001 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'atn
```

5. Tests the arctangent trig function.

AUTO

AUTO [«start line»][,«increment»]
AUTO 9000
AUTO 100,100
AUTO ,5

Command - Makes the COMAL system generate line numbers automatically as a program is typed in. Valid line numbers are 1 - 9999. Each line number is always four characters. Some systems pad with leading 0's (0030), others with leading spaces (30). AUTO begins with the next available line number if you do not specify a starting line. If you don't specify an increment, 10 is used. If you don't specify a starting line number, 10 is used, unless program lines already exist. Then the line number to start at is the last line plus the increment.

Alder COMAL will not allow you to cursor up to correct a previous line while in AUTO mode, other COMALs do.

Just hitting «return» after a line number inserts a blank line in the program, except in C64 Power Driver, C64 COMAL 1.0 and Apple COMAL, which use a blank line to terminate AUTO mode. C64 Power Driver and C64 COMAL 1.0 always start at line 10 unless otherwise specified.

BASIC : exit COMAL to BASIC, see BYE

BITAND

«argument» BITAND «argument»
show(bnum BITAND %00001000)

Operator - Returns the bitwise AND of the two numbers, similar to most BASICs AND. Binary constants are prefixed by a %. The following table shows how BITAND works:

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

```
FUNC test'bitand CLOSED
  IF (3 BITAND 3)<>3 THEN RETURN FALSE
  // (%11 BITAND %11)<>%11  <==binary
  IF (3 BITAND 0)<>0 THEN RETURN FALSE
  // (%11 BITAND %0)<>%0  <==binary
  IF (5 BITAND 6)<>4 THEN RETURN FALSE
  // (%101 BITAND %110)<>%100  <==binary
  RETURN TRUE
ENDFUNC test'bitand
```

6. Tests three types of BITAND operations: both bits on, both bits off, and one bit on - one bit off.

BITOR

«argument» BITOR «argument»
PRINT (bnum BITOR flag)

Operator - Returns the bitwise OR of the two numbers, similar to most BASICs OR. Binary constants are prefixed by a %. The following table shows how BITOR works:

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

```
FUNC test'bitor CLOSED
  IF (3 BITOR 3)<>3 THEN RETURN FALSE
  // (%11 BITOR %11)<>%11  <==binary
  IF (3 BITOR 0)<>3 THEN RETURN FALSE
  // (%11 BITOR %0)<>%11  <==binary
  IF (5 BITOR 6)<>7 THEN RETURN FALSE
  // %101 BITOR %110)<>%111  <==binary
  IF (0 BITOR 0)<>0 THEN RETURN FALSE
  // (%0 BITOR %0)<>%0  <==binary
  RETURN TRUE
ENDFUNC test'bitor
```

7. Tests three types of BITOR operations: both bits on, both bits off, and one bit on - one bit off.

BITXOR

«argument» BITXOR «argument»
bnum=(num1+num2) BITXOR %10000000

__Operator__ - Returns the bitwise exclusive OR of the two numbers. BITXOR performs the bitwise XOR operation bit by bit on the two numbers. Binary constants are prefixed by a %. The following table shows how BITXOR works:

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

```
FUNC test'bitxor CLOSED
  IF (3 BITXOR 3)<>0 THEN RETURN FALSE
  // (%11 BITXOR %11)<>%0  <==binary
  IF (3 BITXOR 0)<>3 THEN RETURN FALSE
  // (%11 BITXOR %0)<>%11  <==binary
  IF (5 BITXOR 6)<>3 THEN RETURN FALSE
  // (%101 BITXOR %110)<>%11  <==binary
  IF (0 BITXOR 0)<>0 THEN RETURN FALSE
  // (%0 BITXOR %0)<>%0  <==binary
  RETURN TRUE
ENDFUNC test'bitxor
```

8. Tests the three types of BITXOR operations: both bits on, both bits off, and one bit on - one bit off.

BYE

BYE

__Command__ - Returns you to the computer's operating system.

Both Amiga COMALs ask you to confirm the exit from COMAL as a safeguard. Amiga COMAL also lets you click on the close window gadget in the top left corner of the COMMAND window to exit COMAL. C64 and C128 carts use the keyword BASIC instead of BYE, since BASIC is the main operating system.

CASE

CASE «control expression» [OF]
CASE reply$ OF
CASE choice OF

__Statement__ - Begins a CASE structure, allowing a multiple choice decision with as many specific WHEN sections as needed. A default OTHERWISE section may be included that is executed if none of the WHEN sections match the condition (which can be either string or numeric). Statement blocks following each WHEN are indented when listed, but the CASE, WHEN and OTHERWISE statements are not). The system will insert the word OF for you if you don't type it.

```
FUNC test'case CLOSED
  DIM reply$ OF 4, choice$ OF 1
  reply$="abcd"
  FOR x:=1 TO 4 DO
    CASE x OF
    WHEN 1
      IF x<>1 THEN RETURN FALSE
    WHEN 2
      IF x<>2 THEN RETURN FALSE
    OTHERWISE
      IF x<3 THEN RETURN FALSE
    ENDCASE
    choice$=reply$(x:x)
    CASE choice$ OF
    WHEN "a"
      IF x<>1 THEN RETURN FALSE
    WHEN "b"
      IF x<>2 THEN RETURN FALSE
    OTHERWISE
      IF x<3 THEN RETURN FALSE
    ENDCASE
  ENDFOR x
  RETURN TRUE
ENDFUNC test'case
```

9. Tests numeric and string CASE matching, with multiple WHEN statements and an OTHERWISE.

CAT

CAT [«filename»]
CAT

__Command__ - Gives a catalog (directory) of the files on a disk. It uses the default drive if none is specified. Pattern and wild card matching is allowed and should match the way the operating system works. For example, with Commodore and IBM computers, the following is true:
? matches any one character
* matches any string of characters

With the Amiga, the * means the current window, and other characters (such as #) are used for pattern matching. On IBM and Amiga, a period (.) is a significant part of a filename, while on Commodore it is just another character. These and other differences are to be expected, since the COMAL system must adapt to the computers operating system methods.

Most COMALs include both CAT and DIR even though they may be identical in purpose, to allow users to use the one they are used to. C64 Power Driver and C64 COMAL 1.0 allow DIR in programs, but not CAT.

CHAIN

CHAIN «filename»
CHAIN "menu"

Command/Statement - Loads and runs a program from a disk file. The program must have previously been SAVEd to disk.

CHANGE

CHANGE "«old text»","«new text»"
CHANGE "zz","print'report"

Command - Changes parts of a program line into another string of characters, just like in word processors.

Both Amiga COMALs do not have this command, but have their own methods of accomplishing it. C64 Power Driver and Apple COMAL do not have the CHANGE command.

CHR$

CHR$(«numeric expression»)
PRINT CHR$(num)

Function - Returns the character with the specified numeric (ASCII) code. ORD is the complimentary function to CHR$.

Note that ASCII codes may vary from system to system (Commodore in particular).

```
FUNC test'chr CLOSED
  IF CHR$(53)<>"5" THEN RETURN FALSE
  IF ORD(CHR$(65))<>65 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'chr
```

10. **Tests for ASCII character "5" and verifies that ORD is the complement of CHR$.**

CLOSE

CLOSE [[FILE] «filenum»]
CLOSE FILE 2

Command/Statement - Closes the file specified. If no specific file is specified, all files are closed (but does not affect files opened via the SELECT OUTPUT statement). No error should occur if you attempt to close a file that is not open.

CP/M COMAL gives an error if you try to close a file that is not open.

CLOSED

PROC «procname»[(params)] [CLOSED]
FUNC «funcname»[(params)] [CLOSED]
PROC newpage(header$) CLOSED
FUNC gcd(n1,n2) CLOSED

Procedure/Function Type - Declares that all variables and arrays inside the procedure or function are to be local - hidden from the main program. Likewise, all variables and arrays in the main program are not known inside a CLOSED procedure or function. However, specific variables and arrays may become known inside a CLOSED procedure or function by use of parameters or the IMPORT statement. Data statements inside a CLOSED procedure or function are considered local (*except for C64 Power Driver, C64 COMAL 1.0 and Apple COMAL). Many COMALs allow a CLOSED procedure or function to be made EXTERNAL. See EXTERNAL.*

```
FUNC test'closed CLOSED
  temp:=TRUE
  closed'proc
  RETURN temp
  //
  PROC closed'proc CLOSED
    temp:=FALSE
  ENDPROC closed'proc
  //
ENDFUNC test'closed
```

11. **Tests that a variable changed inside a closed procedure does not affect a variable with the same name outside that procedure.**

CON

CON

Command - Restarts a program that was previously stopped by a STOP statement or the stop/break key.

Due to the internal linking system used by most COMALs, if lines are added, deleted, or modified, or if new variables are introduced, the program may not be able to be continued.

COS

COS(«numeric expression»)
PRINT COS(number)

Function - Returns the cosine of the number in radians. *CP/M COMAL allows you to choose degrees rather than radians, but has radians as the default.*

```
FUNC test'cos CLOSED
  IF ABS(COS(0)-1)>0.000001 THEN RETURN FALSE
  IF ABS(COS(PI/3)-0.5)>0.000001 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'cos
```

12. Tests the cosine trig function.

CREATE

CREATE «filename»,«# records»,«record size»
CREATE "names",128,200

Command/Statement - Creates a random access file of the specified size. Most COMALs count records in a random file beginning with record number 1. (*CP/M COMAL begins with record number 0.*)

CURCOL

CURCOL
column:=CURCOL

Function - Returns the current column position of the cursor on the text screen. Columns are counted from left to right. The leftmost column is 1.

CURCOL is not implemented by Alder. CURCOL is part of the SYSTEM package in C64 & C128 2.0 cartridge implementations, and requires a USE SYSTEM command prior to use, and is not available within CLOSED procedures or functions without being imported (or having a separate USE SYSTEM within the procedure or function).

```
FUNC test'curcol CLOSED
  col:=CURCOL
  CURSOR 0,30
  IF CURCOL<>30 THEN RETURN FALSE
  CURSOR 0,col
  RETURN TRUE
ENDFUNC test'curcol
```

13. Tests that the correct column is returned by setting only the column position, checking and then replacing the cursor where it originally was.

CURROW

CURROW
row:=CURROW

Function - Returns the current row of the text screen that the cursor is on. Rows are counted top to bottom. The top row is 1.

CURROW is not implemented by Alder. CURROW is part of the SYSTEM package in C64 & C128 2.0 cartridge implementations, and requires a USE SYSTEM command prior to use, and is not available within CLOSED procedures or functions without being imported (or having a separate USE SYSTEM within the procedure or function).

```
FUNC test'currow CLOSED
  row:=CURROW
  CURSOR 10,0
  IF CURROW<>10 THEN RETURN FALSE
  CURSOR row,0
  RETURN TRUE
ENDFUNC test'currow
```

14. Tests that the correct value is returned by setting the cursor location by row, checking the value, then returning the cursor to its original row location.

CURSOR

CURSOR «line»,«position»
CURSOR 1,1

Command/Statement - Positions the cursor to the specified row and column. Rows are counted from top to bottom; columns from left to right. The top row is line 1. The leftmost column is column 1. Cursor positioning is similar to finding your seat at a theater. First find the row, then the position in that row.

Specifying 0 as a row or column means not to change it, thus CURSOR 0,9 would move to position 9 on the current row.

If used as a direct command, CURSOR correctly positions the cursor, but then moves to the first position on the next line for the next command (which surprises many beginners).

```
FUNC test'cursor CLOSED
  row:=CURROW; col:=CURCOL
  CURSOR 10,30
  IF CURROW<>10 THEN RETURN FALSE
  IF CURCOL<>30 THEN RETURN FALSE
  CURSOR 0,20
  IF CURROW<>10 THEN RETURN FALSE
  CURSOR 15,0
  IF CURCOL<>20 THEN RETURN FALSE
  CURSOR row,col
  RETURN TRUE
ENDFUNC test'cursor
```

15. Saves the current cursor position, then moves it to a specific location and checks the values. Next it moves just the column and checks. Then it moves just the row and checks. Finally the cursor is returned to its original location.

DATA

DATA «value»{,«value»}
DATA "Sam",134,"Fred",22,"Gloria",46

Statement - Declares data constants that may be assigned to variables via a READ statement. Data may be text strings within quotes, or numbers.

Multiple items may follow a DATA keyword, separated by commas. When the last DATA item is read, EOD is set to TRUE. Data can be reused following a RESTORE command.

To include a quote mark as part of the string data, use two consecutive quote marks (<u>"abc""defg"</u> is read as <u>abc"defg</u>).

Data inside a CLOSED procedure or function is regarded as local data. Likewise, a READ statement inside a CLOSED procedure or function may only read data inside that procedure or function.

In C64 Power Driver, C64 COMAL 1.0 and Apple COMAL data is always global.

```
FUNC test'data CLOSED
  DIM s$ OF 6
  READ s$
  IF s$<>"passed" THEN RETURN FALSE
  READ x
  IF x<>13 THEN RETURN FALSE
  RETURN TRUE
  DATA "passed",13
ENDFUNC test'data
```

16. **Tests reading string and numeric data from DATA statements.**

DEL

DEL «range»
DEL 460
DEL pause

<u>Command</u> - Removes (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.

<u>DEL 10</u> deletes line 10. <u>DEL pause</u> deletes the procedure or function named pause. <u>DEL 10-30</u> deletes all the lines in the range of 10 through 30. <u>DEL -90</u> deletes all program lines up to and including line 90. <u>DEL 9000-</u> deletes all program lines after and including line 9000.

CP/M COMAL requires that a specified line must exist.

DELETE

DELETE «filename»
DELETE "test5"

<u>Command/Statement</u> - Removes files from disk. Several files may be deleted at once by using pattern matching and wildcard characters. These will match the methods used by the computers operating system and will vary between systems. For example, the following are used by Commodore 64 and 128:
? matches any one character
* matches any string of characters

IBM PC COMAL can only delete one file at a time. C64 Power Driver and C64 COMAL 1.0 require that a drive be specified as part of the filename.

DIM

DIM «string var» OF «max char»
DIM «str array»(«index») OF «max char»
DIM «array name»(«index»)
DIM name$ of 30
DIM players$(1:4) OF 10
DIM scores(min:max)

<u>Command/Statement</u> - Allocates (dimensions) space for strings and arrays. Arrays begin with element 1 unless otherwise specified. Multiple DIMs may be in one statement, separated by commas. Redimensioning is not allowed.

Each element of a numeric array is initially set to 0 when dimensioned (*except for Alder, but they may have this corrected now)*). Arrays may have multiple dimensions, with whatever top and bottom limits you wish, within memory limitations.

```
FUNC test'dim CLOSED
  DIM s$ OF 4, x(1), z(-1:2)
  s$:="passed"
  IF s$<>"pass" THEN RETURN FALSE
  IF x(1)<>0 THEN RETURN FALSE
  x(1):=TRUE; z(-1):=5
  IF x(1)<>TRUE THEN RETURN FALSE
  IF z(-1)<>5 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'dim
```

17. **Tests dimensioning string and numeric arrays, including one with a negative element index. It also checks that strings are truncated if they are longer than the limit and that an array is initialized to 0 when it is dimensioned.**

DIR

DIR [«filename»]
DIR

<u>Command</u> - Gives a directory of the files on a disk. It uses the default drive if no other is specified. Pattern matching and wild card characters are allowed and should match the way the operating system works. For example, with IBM and Commodore 64 computers, the following is true:
? matches any one character
* matches any string of characters

With the Amiga, the * means the current window, and other characters (such as #) are used for pattern matching. On IBM and Amiga, a period (.) is a significant part of a filename, while on Commodore it is just another character. These and other differences are to be expected, since the COMAL system must adapt to the computers operating system methods.

Most COMALs include both CAT and DIR even though they may be identical in purpose, to allow users to use the one they are used to. C64 Power Driver and C64 COMAL 1.0 allow DIR in programs, but not CAT.

DISCARD

DISCARD

__Command__ - Discards all previously linked packages and libraries.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support packages and thus do not have a DISCARD command.

DISPLAY

DISPLAY [«range»] [TO] [«filename»]
DISPLAY "names.lst"
DISPLAY init

__Command__ - Lists a program without line numbers. Ranges of lines may be specified, as with LIST. Program lines may be displayed to disk, allowing them to be inserted into word processing documents and such. Some systems allow program lines displayed to disk to be re-entered with the ENTER or MERGE commands.

In CP/M COMAL, if there are more lines to be displayed than fit on the screen, it automatically pauses after each screenful. Hit <space> for the next screen. In Alder COMAL, <CTRL>+S is used to pause a display. Other COMALs use <space> to pause and restart a listing.

If you cursor up a program listing on the screen, when you hit the top line, Amiga, IBM and C128 COMALs will re-list the previous line, and scroll the listing down. This allows you to backtrack up a listing, and is very handy.

DIV

«dividend» DIV «divisor»
result=guess DIV count

__Operator__ - Provides division with an integer answer. It can be used in conjunction with the MOD operator. DIV defines __x DIV z__ as __INT(x/z)__.

C64 cart and CP/M COMAL do not follow the definition when one of the numbers is negative.

```
FUNC test'div CLOSED
  IF 500 DIV 256<>1 THEN RETURN FALSE
  IF 1500 DIV 5<>300 THEN RETURN FALSE
  IF (-7) DIV 3<>(-3) THEN RETURN FALSE
  IF 7 DIV (-3)<>(-3) THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'div
```

18. Tests two types of division, with and without remainder, followed by two types of division with negative numbers.

DO : see FOR and WHILE

EDIT

EDIT [«range»]
EDIT pause
EDIT

__Command__ - Lists program lines to the screen, similar to LIST, but without indentations, one line at a time, for you to edit. Line ranges may be specified as with LIST. *C64 Power Driver, C64 COMAL 1.0 and Apple COMAL list lines continuously, not one at a time.*

ELIF

ELIF «expression» [THEN]
ELIF reply$ IN "YyNn" THEN

__Statement__ - Allows conditional statement execution. ELIF is short for ELSE IF and is part of the IF structure. The statement block following the ELIF is executed only if the condition is TRUE, otherwise it is skipped (the statement block is automatically indented in listings). If you omit the word THEN, the system will insert it for you.

```
FUNC test'elif CLOSED
  IF FALSE THEN
    RETURN FALSE
  ELIF TRUE THEN
    NULL
  ELSE
    RETURN FALSE
  ENDIF
  RETURN TRUE
ENDFUNC test'elif
```

19. Tests that a TRUE ELIF condition has its statements executed, and that execution then continues after the ENDIF, skipping the ELSE section statements.

ELSE

ELSE

__Statement__ - Provides alternative statements to execute when all IF and ELIF conditions in the IF structure evaluate to FALSE (the statement block is automatically indented in listings).

```
FUNC test'else CLOSED
  IF FALSE THEN
    RETURN FALSE
  ELSE
    RETURN TRUE
  ENDIF
ENDFUNC test'else
```

20. Tests that the ELSE section is executed by default.

END

END [«message»]
END "All Done."

Statement - Terminates program execution. END is optional. Without an END statement, a program ends automatically after its last line is executed. There may be more than one END statement in a program. Programs ending at an END statement may not be restarted via CON (use STOP for this capability). A message may be included to replace the system default end message (usually End At Line 0100 or something similar).

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not allow the optional message after END. CP/M COMAL seems more flexible with the optional message, and thus ending messages can be slightly different. Amiga COMAL's end message is printed in the COMMAND Window.

ENDCASE

ENDCASE

Statement - Marks the end of a CASE structure.

```
FUNC test'endcase CLOSED
  CASE 1 OF
  WHEN 1
    NULL
  OTHERWISE
    RETURN FALSE
  ENDCASE
  RETURN TRUE
ENDFUNC test'endcase
```

21. Tests that program execution skips to after the ENDCASE following a successful WHEN match.

ENDFOR

ENDFOR [«control variable»]
ENDFOR sides#
ENDFOR increment

Statement - Marks the end of a FOR loop. The system will insert the variable name after ENDFOR for you if you omit it (after a SCAN or RUN). Single line FOR statements do not use ENDFOR.

Some systems convert NEXT into ENDFOR for you (making the transition from BASIC easier).

The control variable is considered local to the FOR structure by Amiga, IBM, CP/M, C128 and C64 cartridge COMALs. Thus a for loop variable will not conflict with a variable of the same name in the main program.

```
FUNC test'endfor CLOSED
  FOR x#:=1 TO 2 DO
    y#:=x#
  ENDFOR x#
  IF y#<>2 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'endfor
```

22. Tests that ENDFOR marks the end of a FOR loop.

ENDFUNC

ENDFUNC [«function name»]
ENDFUNC even

Statement - Marks the end of a user defined function. The system will insert the function name after the ENDFUNC for you if you do not type it (after a SCAN or RUN). ENDFUNC is not used with EXTERNAL function header lines. See EXTERNAL for more information.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support nested functions or EXTERNAL functions.

```
FUNC test'endfunc CLOSED
  IF int'func#<>5 THEN RETURN FALSE
  RETURN TRUE
  //
  FUNC int'func# CLOSED
    RETURN 5
  ENDFUNC int'func#
  //
ENDFUNC test'endfunc
```

23. Tests that ENDFUNC marks the end of a function.

ENDIF

ENDIF

Statement - Marks the end of a multi-line IF structure. One line IF statements do not use an ENDIF.

```
FUNC test'endif CLOSED
  IF FALSE THEN
    RETURN FALSE
  ENDIF
  RETURN TRUE
ENDFUNC test'endif
```

24. Tests that program execution skips to after the ENDIF for a failed IF structure.

ENDLOOP

ENDLOOP

<u>Statement</u> - Marks the end of a multi-line LOOP structure.

```
FUNC test'endloop CLOSED
  LOOP
    EXIT
    RETURN FALSE
  ENDLOOP
  RETURN TRUE
ENDFUNC test'endloop
```

25. Tests that program execution continues after the ENDLOOP after an EXIT statement is executed.

ENDPROC

ENDPROC [«procedure name»]
ENDPROC show'item

<u>Statement</u> - Marks the end of a procedure. The system will insert the procedure name after the ENDPROC for you if you do not type it (after a SCAN or RUN). ENDPROC is not used for EXTERNAL procedure header statements. See EXTERNAL for more information.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support nested procedures or EXTERNAL procedures.

```
FUNC test'endproc CLOSED
  setvar(x)
  RETURN x
  //
  PROC setvar(REF var) CLOSED
    var:=TRUE
  ENDPROC setvar
  //
ENDFUNC test'endproc
```

26. Tests that ENDPROC marks the end of a procedure definition.

ENDTRAP

ENDTRAP

<u>Statement</u> - Marks the end of the error handler TRAP structure.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support the error handler structure and thus do not have ENDTRAP.

```
FUNC test'endtrap CLOSED
  TRAP
    div0=5/0
    RETURN FALSE
  HANDLER
    NULL
  ENDTRAP
  RETURN TRUE
ENDFUNC test'endtrap
```

27. Tests that execution of the error handler continues after the ENDTRAP following completion of the handler section.

ENDWHILE

ENDWHILE

<u>Statement</u> - Marks the end of a multi-line WHILE structure. ENDWHILE is not used with single line WHILE statements.

```
FUNC test'endwhile CLOSED
  WHILE FALSE DO
    RETURN FALSE
  ENDWHILE
  RETURN TRUE
ENDFUNC test'endwhile
```

28. Tests that statements are not executed if the WHILE condition fails, and that program execution skips to immediately after the ENDWHILE statement.

ENTER

ENTER «filename»
ENTER "testing.lst"

<u>Command</u> - Enters program lines from an ASCII format file (such as a file of a program previously LISTed to disk). Any current program is cleared from memory prior to entering the new lines (use MERGE to preserve the current program).

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not clear a program first, allowing ENTER to also merge program segments into the current program. CP/M and IBM PC COMALs do not require line numbers in the file being entered.

When transferring a COMAL program from one system to another, LIST the program to disk, then

ENTER or MERGE it into the other system. This may also be done via modem or networks.

EOD

EOD
WHILE NOT EOD DO

Function - Boolean function that returns TRUE if End Of Data has been reached. If there are no DATA statements in the program, EOD is always TRUE.

```
FUNC test'eod CLOSED
  IF EOD THEN RETURN FALSE
  WHILE NOT EOD DO READ x
  IF NOT EOD THEN RETURN FALSE
  RETURN TRUE
  DATA 13,14
ENDFUNC test'eod
```

29. Tests that EOD returns FALSE if there is data to be read, that EOD can be used to read all data, and that after all the data is read, EOD is TRUE.

EOF

EOF(«filenum»)
WHILE NOT EOF(infile) DO

Function - Boolean function that returns TRUE when End Of File has been reached. Since several files may be open at one time, you must specify the file number.

ERR

ERR
CASE err OF

Function - Returns the error number when an error occurs within an error handler structure. Error numbers are implementation specific.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support an error handler and thus do not have ERR.

```
FUNC test'err CLOSED
  TRAP
    REPORT 13
  HANDLER
    IF ERR<>13 THEN RETURN FALSE
  ENDTRAP
  RETURN TRUE
ENDFUNC test'err
```

30. Tests that the error number reported by ERR is the same one that has been trapped.

ERRTEXT$

ERRTEXT$
PRINT ERRTEXT$

Function - Returns the error message for the current error inside an error handler. Error messages are implementation specific, and methods of dealing with them varies between COMALs.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support an error handler and thus do not have ERRTEXT$.

```
FUNC test'errtext CLOSED
  TRAP
    IF LEN(ERRTEXT$)<>0 THEN RETURN FALSE
    div0:=5/0  //division by 0 error
  HANDLER
    IF LEN(ERRTEXT$)>0 THEN RETURN TRUE
  ENDTRAP
  RETURN FALSE
ENDFUNC test'errtext
```

31. Tests that there is no text in ERRTEXT$ prior to an error, and that after an error occurs, ERRTEXT$ contains the message.

ESC

ESC
EXIT WHEN ESC

Function - Returns TRUE if the stop/break key has been pressed. This is only useful if the stop/break key is disabled (see TRAP).

ESC seems unreliable in Alder COMAL.

```
FUNC test'esc CLOSED
  TRAP ESC-
  TRAP ESC+
  IF ESC THEN NULL
  IF ESC THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'esc
```

32. Tests that the stop/break key can be disabled and enabled. It then tests that once set, the ESC flag can be cleared back to its FALSE setting.

EXEC

[EXEC] «procname»[(«parameter list»)]
show'item(number)

Command/statement - Executes a procedure. May be used from direct mode. The word EXEC is optional and rarely typed. Multiple EXEC statements may be on one line, separated by semicolons. Only the procedure name needs to be typed to execute a procedure.

```
FUNC test'exec CLOSED
  x:=FALSE
  setx(x)
  RETURN x
  //
  PROC setx(REF y)
    y:=TRUE
  ENDPROC setx
  //
ENDFUNC test'exec
```

33. Tests that the original variable passed as a REF parameter is properly updated after the procedure call.

EXIT

EXIT [WHEN «condition»]
EXIT WHEN errors>3

Statement - Provides the method for leaving a LOOP structure. It can be conditional with the optional WHEN extension.

CP/M COMAL and Amiga COMAL use EXIT to exit from FOR, REPEAT and WHILE structures as well as LOOP. This can cause potential problems and is viewed as a flaw rather than an enhancement.

```
FUNC test'exit CLOSED
  LOOP
    EXIT
    RETURN FALSE
  ENDLOOP
  RETURN TRUE
ENDFUNC test'exit
```

34. Tests that EXIT leaves the loop structure and begins execution following the ENDLOOP statement.

EXP

EXP(«numeric expression»)
PRINT EXP(number)

Function - Returns the natural logarithm's base value e raised to the power specified. A good representation of e is 2.718281828.

```
FUNC test'exp CLOSED
  IF EXP(0)<>1 THEN RETURN FALSE
  IF ABS(EXP(1)-2.71828)>0.00001 THEN RETURN FALSE
  IF ABS(EXP(LOG(10))-10)>0.000001 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'exp
```

35. Tests a few conditions of the natural logarithms base value e raised to the specified power.

EXTERNAL

PROC «name»[(«parms»)][EXTERNAL «file»]
FUNC «name»[(«parms»)][EXTERNAL «file»]
FUNC rec'size(name$) EXTERNAL "rec.ext"
PROC set'up EXTERNAL "setup.ext"

Special - Identifies a procedure or function as an external one. This means that the body of the procedure or function is stored on disk, and is not part of the program itself. Thus ENDFUNC or ENDPROC are not used. An external procedure or function is considered CLOSED. To be used as external, a procedure or function must be CLOSED and previously SAVEd to disk.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support EXTERNAL procedure or functions. Amiga, IBM, C64 and C128 cartridge COMALs allow the filename to be a variable.

```
FUNC test'external CLOSED
  // this test'external is only valid if
  // called as an external func
  RETURN TRUE
ENDFUNC test'external
```

36. Test that a function can be external. Note that this test is only valid if the function is saved to be an external function.

FALSE

FALSE
ok:=FALSE

System Constant - Always equals 0. It can be used in comparisons or as a numeric expression. Example, test:=FALSE is the same as test:=0.

```
FUNC test'false CLOSED
  IF FALSE<>0 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'false
```

37. Tests that FALSE is equal to 0.

FILE

: see CLOSE, INPUT, OPEN, PRINT, READ, WRITE

FIND

FIND "«text string»"
FIND " PROC "

Command - Searches the program for specified text. It is case sensitive in most COMALs, so that repeat would not match REPEAT.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not have the FIND command.

FOR

FOR «var»:=«#» TO «#» [STEP «#»] DO [«statement»]
FOR x:=10 TO 1 STEP -1 DO PRINT x
FOR delay:=1 TO amount DO NULL
FOR num:=1 TO total DO

Statement - Marks the start of a FOR structure or one line FOR statement. The variable is initialized to the start value before loop execution begins. A check is made that the variable value does not exceed the end value before executing the loop statements (it is possible for the loop to be skipped entirely if the start value exceeds the end value to begin with). If the step value is negative, the variable is decremented with each loop, rather than incremented. The variable may be an integer variable which yields faster execution in many systems. The statement block within a multi-line FOR structure is automatically indented when listed. The system will insert the word DO for you if you omit it.

The FOR loop variable is considered LOCAL to the FOR structure in Amiga, IBM, CP/M, C64 and C128 cartridge COMALs. Thus a for loop variable will not conflict with a variable of the same name in the main program.

```
FUNC test'for CLOSED
  FOR w:=5 TO 5 DO
    IF w<>5 THEN RETURN FALSE
  ENDFOR w
  FOR z:=1 TO 0 DO
    RETURN FALSE
  ENDFOR z
  FOR x:=10 TO 7 STEP -2 DO
    t:=x
  ENDFOR x
  IF t<>8 THEN RETURN FALSE
  FOR y#:=1 TO 1 DO
    RETURN TRUE
  ENDFOR y#
  RETURN FALSE
ENDFUNC test'for
```

38. Tests a FOR loop that is executed only once, then one that is not executed at all. Next it test a loop that is decremented, and finally an integer loop.

FUNC

FUNC «name»[(«parm»)] [CLOSED]
FUNC «name»[(«parm»)] EXTERNAL «filename»
FUNC call'answered EXTERNAL "call"
FUNC but'first$(text$) CLOSED
FUNC occurances#(text$,c$)

Statement - Marks the start of a user defined function. Parameter passing is allowed; parameters used are considered local to the function unless preceded by the REF keyword. If the statement ends with CLOSED, the function is considered a closed function, and all variables and arrays in it are unknown to the main program. Likewise, all variables and arrays in the main program are then unknown to the closed function. Use IMPORT or parameters to bring main program variables or arrays into a closed function. The block of statements inside the function definition are automatically indented when listed. Functions may be recursive.

String, external and nested functions are not supported by Apple COMAL, C64 COMAL 1.0 or C64 Power Driver. A closed function is more tightly closed in CP/M COMAL than other COMALs.

```
FUNC test'func CLOSED
  IF hope$<>"hope" THEN RETURN FALSE
  RETURN TRUE
  //
  FUNC hope$ CLOSED
    RETURN "hope"
  ENDFUNC hope$
  //
ENDFUNC test'func
```

39. Tests that ENDFUNC ends the function definition.

GET$

GET$(«filenum»,«# of characters»)
text$=GET$(2,16)

Function - Returns the specified number of characters from the specified file. The file must previously have been opened as a read type file. If the end of file is reached before the specified number of characters are retrieved, only those retrieved prior to EOF will be returned (there is no padding of spaces and no error occurs unless you attempt to read from the file again).

CP/M COMAL has difficulties with EOF while executing GET$ due to the manner that CP/M is structured.

GOTO

GOTO «label name»
GOTO jail

Statement - Transfers program execution to the line with the specified label name. Since COMAL has many structures and loop methods, GOTO is not required to be used other than in advanced programs. It is being considered to remove GOTO from the COMAL standard.

```
FUNC test'goto CLOSED
  GOTO this'line'is'making'me'ill
  RETURN FALSE
this'line'is'making'me'ill:
  RETURN TRUE
ENDFUNC test'goto
```

40. Tests that GOTO can jump over lines.

HANDLER

HANDLER

Statement - Marks the beginning of the error handling section of the TRAP HANDLER structure. The block of statements in the trapped section and the error handling section are automatically indented when listed.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support an error handler and thus do not have HANDLER.

```
FUNC test'handler CLOSED
  TRAP
    div0=5/0
    RETURN FALSE
  HANDLER
    RETURN TRUE
  ENDTRAP
  RETURN FALSE
ENDFUNC test'handler
```

41. Tests that the handler section is executed after an error.

IF

IF «condition» THEN [«statement»]
IF reply$ IN "yYnN" THEN

Statement - The start of a multi-line IF structure. May also be a one line IF statement (no ENDIF is used). IF allows conditional statement execution. The block of statements following the IF are only executed if the condition is TRUE. The block of statements are automatically indented when listed. The system will insert the word THEN for you if it is omitted.

```
FUNC test'if CLOSED
  t:=FALSE
  IF TRUE THEN
    t:=TRUE
  ENDIF
  IF t THEN RETURN TRUE
  RETURN FALSE
ENDFUNC test'if
```

42. Tests that statements are executed following a TRUE condition with a single line IF or multi-line IF structure.

IMPORT

IMPORT «identifier» {,«identifier»}
IMPORT running'total

Statement - Allows a closed procedure or function to use variables, arrays, procedures and functions from the main program. There may be more than one IMPORT statement in a procedure or function, and all IMPORT statements should come prior to any executable statement in that procedure or function.

IMPORT is not supported by C64 Power Driver, C64 COMAL 1.0 or Apple COMAL. In those systems all procedures and functions in the main program are automatically available inside closed procedures and functions; variables and arrays can be shared via parameter passing.

```
FUNC test'import CLOSED
  t:=FALSE
  p
  RETURN t
  //
  PROC p CLOSED
    IMPORT t
    t:=TRUE
  ENDPROC p
  //
ENDFUNC test'import
```

43. Tests that a variable outside a CLOSED procedure is accessible after an IMPORT statement.

IN

«string1» IN «string2»
IF guess$ IN word$ THEN winner

Operator - Returns the position of «string1» within «string2» (or 0 if not found). If «string1» is the null string ("") 0 is returned.

If «string1» is the null string, Alder returns 1; C64 cart, CP/M COMAL and C64 Power Driver return the length of the string plus 1.

```
FUNC test'in CLOSED
  DIM s$ OF 3
  s$:="abc"
  IF ("b" IN s$)<>2 THEN RETURN FALSE
  IF ("d" IN s$) THEN RETURN FALSE
  IF ("" IN s$) THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'in
```

44. Tests that substrings are found when possible; are not found when not possible; and that the null string is not found in any string.

INPUT

INPUT FILE «file#»[,«rec#»]: «var list»
INPUT [AT «row»,«col»[,«len»]:] [«prompt»:]«vars»[«mark»]
INPUT FILE 2: text$
INPUT AT 5,2,10:"ZIP CODE: ": zip'code,
INPUT "Age? ":age
INPUT reply$

Statement - INPUT allows the user to enter data into a running program from the keyboard (the AT section is optional). INPUT FILE gets the data from the file specified, which must have been previously opened for reading. INPUT FILE reads ASCII files, such as those created by PRINT FILE or a Word Processor with ASCII file output (does not read files created by WRITE FILE statements). The prompt is optional and may be a variable.

During the INPUT from keyboard request, the input area is a protected field extending to the end of the line (unless the length part of the AT section is specified). A 0 length means only a carriage return will be accepted (*except in CP/M COMAL*). A 0 for the row or column means not to change it (stay in the same row or column). If the «mark» is a comma, the cursor remains where it is after the reply. If it is a semicolon, spaces are printed to the next zone (one space by default if ZONE is not specified), then the cursor remains at that position.

CP/M COMAL requires that the prompt be a constant if used. Alder, C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not allow a comma to be used as the mark at the end of the statement (only allow a semicolon).

INT

INT(«numeric expression»)
tally:+INT(number)

Function - Returns the nearest integer less than or equal to the specified number. Both positive and negative numbers are rounded down (-8.3 becomes integer -9).

```
FUNC test'int CLOSED
  IF INT(3)<>3 THEN RETURN FALSE
  IF INT(3.8)<>3 THEN RETURN FALSE
  IF INT(-3)<>-3 THEN RETURN FALSE
  IF INT(-3.3)<>-4 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'int
```

45. Tests that the value returned is the same for integers; and that the value returned is the nearest integer that is less than a non-integer.

KEY$

KEY$
WHILE KEY$="" DO NULL

Function - Returns the first character in the keyboard buffer. If no key has been pressed, the null string ("") is returned.

C64 cart and C64 Power Driver return CHR$(0) if no key has been pressed.

```
FUNC test'key CLOSED
  WHILE KEY$>CHR$(0) DO NULL
  IF KEY$>"" THEN RETURN FALSE
  IF LEN(KEY$)>0 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'key
```

46. Tests clearing the keyboard buffer, then checks that no key is left in the buffer. Finally it checks that the null string (length 0) is returned when there is no key pressed.

LABEL

[LABEL] «label name»:
months:

Identifier - Assigns a label name to the line. This label is only referenced by RESTORE or GOTO. It is non-executable and may be placed anywhere within a program as a one line statement. The line is not indented (*except in Amiga and CP/M COMAL*). You do not have to type the word label, and most systems do not list it either (similar to how EXEC is treated).

```
FUNC test'label CLOSED
  RESTORE right'here
  READ x
  IF x<> 5 THEN RETURN FALSE
  RETURN TRUE
  DATA 4
right'here:
  DATA 5
ENDFUNC test'label
```

47. Tests that a label can be used with RESTORE to reset the next data item pointer.

LEN

LEN(«string expression»)
length=LEN(text$)

Function - Returns the length of the specified string. All characters, even non-printing characters, are counted. The length of the null string "" is 0.

```
FUNC test'len CLOSED
  DIM s$ OF 5
  s$:=""
  IF LEN(s$)<>0 THEN RETURN FALSE
  s$:="12345"
  IF LEN(s$)<>5 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'len
```

48. **Tests that the length of the null string is 0 and that the length of a string is equal to the number of characters assigned to it.**

LINK

LINK «filename»
LINK "mouse"

Command - Loads a disk based package and links it to the program. A USE command is required in the program before the package commands are available to the program.

Amiga COMAL links a package into the system with the USE command.

LIST

LIST [«range»] [TO] [«filename»]
LIST header
LIST "myprog.lst"
LIST pause "pause.lst"

Command - Lists the specified program lines. If no lines are specified, all lines are listed. If a filename is specified, the lines are listed to that file (in ASCII form), otherwise they are listed to the current output location (screen by default). A procedure or function name can be used to specify a line range.

```
LIST 30     lists only line 30
LIST -30    lists all lines up to and including line 30
LIST 9000-  lists all lines after and including line 9000
LIST 100-200  lists lines 100 through 200 inclusive
LIST pause  lists all lines in procedure name pause
```

Lines LISTed to disk may be merged into the programs with the MERGE command (*C64 Power Driver, C64 COMAL 1.0 and Apple COMAL use the ENTER command for merges*). Statement blocks within structures are automatically indented when listed.

LIST and ENTER commands are useful when transferring programs from one system to another. LIST the program to disk. Then ENTER it into the other system. Modems and networks may also be used to aid the transfer.

If more lines than fit on one screen are to be listed, CP/M COMAL pauses after each screen... press space for the next screen. With Alder COMAL, <CTRL>+S is used to pause a listing. The other COMALs have <space> pause and restart a listing.

CP/M COMAL requires the filename to come before the line range (reverse of the other COMALs). CP/M COMAL requires that a line number used to specify a range must exist. Alder requires the keyword TO before a filename; without the word TO, the LIST becomes a global search for the text in quotes.

Amiga, IBM and C128 COMALs will re-list preceding lines if you cursor up on the top line of the screen. This is handy to see lines that have just scrolled off the top.

LOAD

LOAD «filename»
LOAD "menu"

Command - Loads a program from disk into the computers memory. Program memory is cleared before loading the program. The program being loaded must have previously been SAVEd to disk. You cannot LOAD a program SAVEd by a different COMAL implementation. To transfer programs between implementations, use LIST to disk, and ENTER to retrieve them.

LOG

LOG(«numeric expression»)
PRINT LOG(number);

Function - Returns the natural logarithm of the number specified. This is log to the base e. A good representation of e is 2.718281828.

```
FUNC test'log CLOSED
  IF LOG(1)<>0 THEN RETURN FALSE
  IF ABS(LOG(2.71828)-1)>0.00001 THEN RETURN FALSE
  IF ABS(EXP(LOG(10))-10)>0.000001 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'log
```

49. **Tests the natural logarithm function.**

LOOP

LOOP

Statement - Starts a multi-line loop structure that uses the EXIT statement as the exit method. The statement block within a LOOP structure are automatically indented when listed.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not have the LOOP structure.

```
FUNC test'loop CLOSED
  count:=0
  LOOP
    EXIT WHEN count=2
    count:+1
  ENDLOOP
  IF count<>2 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'loop
```

50. Tests exiting a loop after a certain number of passes.

MAIN

MAIN

Command - Returns to the main program section when a program is stopped while an external procedure or function is being executed.
Alder, C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not have a MAIN command.

MERGE

MERGE [«line#»[,«increment»]] «filename»
MERGE "readrec.lst"

Command - Merges program lines from a disk file (ASCII format). The lines are renumbered as they are merged into the current program.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL use the ENTER command to merge program segments. Alder, Amiga COMAL, and C64/C128 2.0 cartridge COMALs require line numbers in the file being merged, even though those line numbers are ignored.

MOD

«dividend» MOD «divisor»
color=number mod 16

Operator - Returns the modulo of the numbers. It can be used in conjunction with DIV. It defines x MOD z as x-(x DIV z)*z which expands into x-INT(x/z)*z. If z is negative, the result may be irrelevant, but should follow the definition.

When one of the numbers is negative, CP/M COMAL and C64 cartridge COMAL do not follow the definition.

```
FUNC test'mod CLOSED
  IF 500 MOD 256<>244 THEN RETURN FALSE
  IF 1500 MOD 5<>0 THEN RETURN FALSE
  IF (-7) MOD 3<>2 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'mod
```

51. Tests that MOD in cases where the result should be 0 as well as when the result should be a positive value. Then it tests MOD with a negative number.

NEW

NEW

Command - Erases the program currently in memory and clears all variables. Linked packages are also cleared.

Amiga COMALs request confirmation if a NEW command is issued and the program in memory has been modified and not saved.

NEXT : converted to ENDFOR, see ENDFOR

NOT

NOT «condition»
IF NOT ok THEN

Operator - Returns to reverse of the TRUE / FALSE evaluation:
NOT TRUE = FALSE
NOT FALSE = TRUE

```
FUNC test'not CLOSED
  IF (NOT TRUE)=TRUE THEN RETURN FALSE
  IF (NOT FALSE)<>TRUE THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'not
```

52. Tests the two possibilities of NOT: NOT TRUE and NOT FALSE.

NULL

NULL
WHILE KEY$="" DO NULL

Statement - Does nothing. It can be used as an empty statement, such as in a pause loop.

```
FUNC test'null CLOSED
  NULL
  RETURN TRUE
ENDFUNC test'null
```

53. Tests that a NULL statement is accepted.

OF : see DIM and CASE

OPEN

OPEN [FILE] «file#»,«filename»,«type»
OPEN FILE 2,"scores",READ

Command/Statement - Opens a file and assigns it a file number (that is used later with file operation statements). A file may be opened to the screen, printer and serial port as well as disk.

OR

«condition» OR «condition»
IF reply$<"a" OR reply$>"z" THEN

Operator - Returns the result of the logical OR of the two expressions. This is different than most BASICs in which OR is a bitwise operator. For bitwise OR in COMAL see BITOR.

```
 OR   || TRUE  | FALSE
======|===============
TRUE  || TRUE  | TRUE
------|+-------+------
FALSE || TRUE  | FALSE
```

```
FUNC test'or CLOSED
  IF (TRUE OR TRUE)<>TRUE THEN RETURN FALSE
  IF (TRUE OR FALSE)<>TRUE THEN RETURN FALSE
  IF (FALSE OR TRUE)<>TRUE THEN RETURN FALSE
  IF (FALSE OR FALSE)<>FALSE THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'or
```

54. Tests the four possibilities of OR: both TRUE, both FALSE, and one TRUE - one FALSE.

ORD

ORD(«string expression»)
a=ORD("a")

Function - Returns an integer representing the ASCII code (ordinal number) of the specified string. If the string is longer than one character, ORD only looks at the first character. An error results if the null string is used. ORD is system dependent and may vary between systems (especially Commodore).

```
FUNC test'ord CLOSED
  IF ORD("5")<>53 THEN RETURN FALSE
  IF ORD(CHR$(65))<>65 THEN RETURN FALSE
  IF ORD("c")-ORD("a")<>2 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'ord
```

55. Tests that the ASCII value for "5" is correct, then that CHR$ is a correct complement to ORD, and that the letter c is 2 more than the letter a in ORD values.

OTHERWISE

OTHERWISE

Statement - Marks the start of the default case in the CASE structure. The block of statements after the OTHERWISE are executed if no WHEN case condition is met. The block of statements are indented automatically when listed.

```
FUNC test'otherwise CLOSED
  x:=3
  CASE x OF
  WHEN 1,2,4
    RETURN FALSE
  OTHERWISE
    RETURN TRUE
  ENDCASE
  RETURN FALSE
ENDFUNC test'otherwise
```

56. Tests that if no WHEN case is matched, the OTHERWISE section is executed.

PAGE

PAGE

Statement/Command - Clears the screen and puts the cursor at the top left corner (1,1). If output is to another device, a CHR$(12) is sent (form feed).

```
FUNC test'page CLOSED
  PAGE
  IF CURCOL<>1 THEN RETURN FALSE
  IF CURROW<>1 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'page
```

57. Tests that PAGE clears the screen and leaves the cursor in position 1,1 (top left corner).

PEEK

PEEK(«memory address»)
device=PEEK(4839)

Function - Returns the decimal value of the contents in the specified memory location. PEEK is very machine dependent.

PEEK is a package command in Amiga, Alder and IBM PC COMAL and requires a USE command prior to using it.

PI

PI
PRINT "Value of PI is";PI

Function - Returns the value of pi. The number of digits varies between systems. Generally, pi is equal to 3.14159266.

```
FUNC test'pi CLOSED
  IF PI<3.141592 THEN RETURN FALSE
  IF PI>3.141593 THEN RETURN FALSE
  RETURN ABS(PI-4*ATN(1))<1e-06
ENDFUNC test'pi
```

58. Tests that PI returns a close representation of the value of PI.

POKE

POKE «memory address»,«contents»
POKE 4839,13

Function - Places the specified decimal value into the memory indicated memory location. POKEing the wrong value into some memory locations may "lock out" your machine.

POKE is a package command in Alder, Amiga and IBM PC COMAL, and requires a USE statement prior to using it.

PRINT

```
PRINT [AT «row»,«col»:] [USING «form»:] «list»[«mark»]
PRINT [FILE «#»[,«rec»]:][USING «form»:]«list»[«mark»]
```
PRINT FILE 2: text$
PRINT AT 9,1: USING "$###.##": amount

Statement/Command - Prints items as specified. More than one item may be specified in one PRINT statement, separated by a , or ;. A comma is a null separator (no spaces between items). A semicolon ; prints spaces to the next zone (one space by default if ZONE has not been specified).

PRINT FILE statements write items in ASCII to the file (it may be preferable to use WRITE FILE instead for data files). The AT and USING sections are optional parts of a PRINT statement, and are part of COMMON COMAL. They provide added flexibility.

CP/M COMAL issues a CHR$(9) for each comma (as a tab mark). C64 Power Driver and C64 cartridge COMAL use the original meanings of the comma and semicolon separators. Unless you use a ZONE statement, the results are the same. See ZONE.

```
FUNC test'print CLOSED
  cc=CURCOL
  PRINT "",
  IF CURCOL<>cc THEN RETURN FALSE
  zz=ZONE
  ZONE 5
  PRINT "";
  cc=CURCOL
  PRINT "";
  IF cc+5<>CURCOL THEN RETURN FALSE
  ZONE zz //reset zone
  RETURN TRUE
ENDFUNC test'print'print
```

59. Tests that printing a null string with a comma endmark does not move the cursor. Then it checks that zone works with the semicolon separator. The zone is reset after the tests.

PROC

PROC «name»[(«parm»)] [CLOSED]
PROC «name»[(«parm»)] [EXTERNAL «file»]
PROC readrec(number)
PROC compare(t1$,t2$) EXTERNAL "comp.ext"

Statement - Marks the start of a multi-line procedure definition, including parameter passing (parameters are considered local unless preceded by the REF keyword). A procedure may recursive. The CLOSED keyword is included at the end of the statement to make a procedure closed. A closed procedure does not know about variables or arrays in the main program (unless they are IMPORTed). Likewise, variables and arrays inside a closed procedure are local, and remain unknown to the main program.

A closed procedure can be used as an EXTERNAL procedure by SAVEing it to disk. You can define a procedure within another procedure (nested).

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not allow nested procedures or EXTERNAL procedures.

```
FUNC test'proc CLOSED
  x:=FALSE
  setx
  RETURN x
  //
  PROC setx
    x:=TRUE
  ENDPROC setx
  //
ENDFUNC test'proc
```

60. Tests that variables changed in an open procedure are also changed in the main program.

RANDOM

OPEN FILE «file#»,«filename»,RANDOM «len»
OPEN FILE 2,"subs",RANDOM 88

__File Type__ - Identifies a file as random access, for both reading and writing.

Commodore COMALs have a 254 byte limit on record length. Alder COMAL also allows READONLY to be specified after the record length, in which case the file may only be read and not written to.

RANDOMIZE

RANDOMIZE [«seed»]
RANDOMIZE
RANDOMIZE num

__Statement/Command__ - Randomizes the random number generator. This generates a series of pseudo random numbers. You only need to use the RANDOMIZE command once in your program (such as right the the very beginning).

Specify a "seed" number after RANDOMIZE and you cause a specific series of random numbers to be generated. The series of numbers will be the same each time that specific seed is used. This is helpful while testing a program that uses random numbers.

```
FUNC test'randomize CLOSED
  RANDOMIZE
  FOR x:=1 TO 50 DO
    r:=RND(1,20)
    IF (r<1) OR (r>20) THEN RETURN FALSE
  ENDFOR x
  DIM rarray(1:9)
  RANDOMIZE 9 //specific set of numbers
  FOR x=1 TO 9 DO rarray(x)=RND(1,99)
  RANDOMIZE 9 //reset to specific set
  FOR x=1 TO 9 DO
    IF RND(1,99)<>rarray(x) THEN RETURN FALSE
  ENDFOR x
  RETURN TRUE
ENDFUNC test'randomize
```

61. Tests that no seed is needed with RANDOMIZE. Then checks that a random number is in its correct range. Next it checks that a specified seed gives the same set of random numbers each time.

READ

READ [FILE «file#»[,«rec#»]:] «var list»
OPEN [FILE] «filenum»,«filename»,READ
READ name$,age
READ FILE 2,record: name$,adr$,city$,st$
OPEN FILE 3,filename$,READ

__File Type__ or __Statement__ - In an OPEN statement, specifies a sequential file to be read. READ also can be used as a statement to read data from DATA statements. Finally, READ FILE statements read data from sequential or random files that were created with WRITE FILE statements (these are binary files, not ASCII).

```
FUNC test'read CLOSED
  DIM s$ OF 6
  READ s$
  IF s$<>"passed" THEN RETURN FALSE
  READ x
  IF x<>13 THEN RETURN FALSE
  RETURN TRUE
  DATA "passed",13
ENDFUNC test'read
```

62. Tests that string and numeric data are properly read.

REF

REF «var»
PROC alter(REF text$) CLOSED
FUNC slide(REF text$)

__Parameter Type__ - Specifies that the parameter will be an alias for the matching variable or array in the calling statement (passed by reference rather than by value). The value of the calling statement changes as its REF parameter is changed.

```
FUNC test'ref CLOSED
  setvar(x)
  RETURN x
  //
  PROC setvar(REF var) CLOSED
    var:=TRUE
  ENDPROC setvar
  //
ENDFUNC test'ref
```

63. Tests that variables used as REF parameters have their value updated as the parameter is changed within the procedure.

RENAME

RENAME «old filename»,«new filename»
RENAME "temp","final"

__Statement/Command__ - Renames a disk file. Takes an existing file and gives it a new name.

RENUM

RENUM [«target start»][,«increment»]
RENUM 100
RENUM ,5
RENUM 9000,1

Command - Renumbers the program in memory. Valid line numbers are 1-9999. By default, it renumbers a program to start at line 10 and increment by 10, unless you specify otherwise.

REPEAT

REPEAT

Statement - Marks the start of a multi-line REPEAT structure. The block of statements after the REPEAT are automatically indented when listed. They are continually executed until the condition after the UNTIL evaluates to FALSE. The statements will always be executed at least once.

```
FUNC test'repeat CLOSED
  REPEAT
    READ x
    IF x=4 THEN RETURN FALSE
  UNTIL x=5
  IF x=5 THEN RETURN TRUE
  RETURN FALSE
  DATA 10,3,5,4
ENDFUNC test'repeat
```

64. Tests that a REPEAT loop is repeated and exited properly.

REPORT

REPORT [«error code»]
REPORT
REPORT 256

Statement - Part of the error handler structure. REPORT causes an error (optionally you can specify what error number to generate). This is useful when using multiple nested handlers. REPORT puts you into the next outer handler. If REPORT is issued while not in a handler section, the error is reported to the system. REPORT is very system dependent. All COMALs (*except CP/M COMAL and Alder COMAL*) allow you to include an error message along with the error number.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not support the error handler structure, thus do not have the REPORT statement.

```
FUNC test'report CLOSED
  TRAP
    REPORT 13
    RETURN FALSE
  HANDLER
    IF ERR=13 THEN RETURN TRUE
  ENDTRAP
  RETURN FALSE
ENDFUNC test'report
```

65. Tests that REPORT can issue a specific error number.

RESTORE

RESTORE [«label»]
RESTORE month'names
RESTORE

Statement/Command - Allows data in DATA statements to be re-used. The pointer to the next data item is reset back to the first data item, unless a label is specified. Then the next data item pointer points at the first data item following the label.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not allow labels to be used with RESTORE, but always restore to the first data item.

```
FUNC test'restore CLOSED
  DIM l$ OF 5
  RESTORE good'language
  READ l$
  IF l$<>"COMAL" THEN RETURN FALSE
  RETURN TRUE
good'language:
  DATA "COMAL"
another'language:
  DATA "BASIC"
ENDFUNC test'restore
```

66. Tests that RESTORE will restore the pointer to the next data item following the specified label.

RETURN

RETURN [«value»]
RETURN TRUE
RETURN text$

Statement - Assigns the value specified after the RETURN to the function and returns control to the calling statement. RETURN may also be used to terminate a procedure early.

```
FUNC test'return CLOSED
  x=0
  rtest'return
  IF x=0 THEN RETURN TRUE
  RETURN FALSE
  //
  PROC rtest'return CLOSED
    RETURN
    x=5
  ENDPROC rtest'return
  //
ENDFUNC test'return
```

67. Tests that RETURN will cause an early exit to a procedure and that RETURN will return values from a function.

RND

RND [(«start num»,«end num»)]
dice=RND(1,6)+RND(1,6)

__Function__ - Returns a random number greater than or equal to zero, and less than 1. If start and end limits are specified, RND returns an integer within the specified limits, inclusive.

Alder COMAL uses RND# when specifying a random integer in a range. C64 Power Driver uses RND(0) rather than RND.

```
FUNC test'rnd CLOSED
  RANDOMIZE
  FOR x:=1 TO 50 DO
    r:=RND(1,20)
    IF (r<1) OR (r>20) THEN RETURN FALSE
    r:=RND
    IF (r<0) OR (r>1) THEN RETURN FALSE
  ENDFOR x
  RETURN TRUE
ENDFUNC test'rnd
```

68. Tests that RND can return integers within a specified range and that RND can return a random number between 0 and 1.

RUN

RUN [«filename»]
RUN
RUN "menu"

__Command__ - Begins execution of the program currently in memory. If a file is specified, the memory is cleared and the file is loaded and run.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not allow a filename to be specified.

SAVE

SAVE «filename»
SAVE "zombies"

__Command__ - Stores the program in memory to the specified file in compressed form. Comments are not removed. Later the program can be retrieved with the LOAD, RUN, or CHAIN command. Procedures or functions stored with the SAVE command can be used as EXTERNAL procedures and functions.

A SAVEd program may not be transferred to another COMAL system. To transfer a program it must be in ASCII form as with the LIST to disk command. Use the ENTER command to retrieve it to the other system. Modems or networks may also be used to transfer the file.

Some COMALs save linked packages with the program. Commodore COMALs will not save a program if one with the same name already exists. IBM COMAL overwrites any file with the same name (without warning or notification). Amiga COMAL will rename a file found with the same name (adding the .BKUP extension) and then save the program (along with an ICON image for it).

SCAN

SCAN

__Command__ - Scans the program in memory for structure errors. Once a program has been SCANned or RUN procedures and functions may be called from direct mode.

SELECT

SELECT [OUTPUT] «type»
SELECT loc$

__Statement/Command__ - Selects the output location. You do not have to type the keyword OUTPUT. If it is omitted, COMAL will insert it for you. Default location is the screen. Possible locations are system dependent, but most COMALs include a standard set of destinations among their choices. Amiga COMAL, CP/M COMAL, Apple COMAL, C64 Power Driver, C64 COMAL 1.0, C64 2.0 cartridge, C128 2.0 cartridge, PET COMAL 0.14, PET COMAL 1.02, and PET COMAL 2.0 include:

SELECT "ds:"	**Data Screen**
SELECT "lp:"	**Line Printer**

Most also include:

SELECT "sp:"	Serial Port
SELECT "filename"	

IBM PC COMAL does not recognize the standard destinations in addition to its own. It only uses the identifications set up by MS-DOS:

SELECT "con:"	*Console*
SELECT "prn:"	*Printer*
SELECT "lpt1:"	*Printer (also lpt2: and lpt3:)*
SELECT "com1:"	*Serial Port (also com2:)*

```
FUNC test'select CLOSED
  TRAP
    SELECT OUTPUT "lp:" //printer
    SELECT OUTPUT "ds:" //screen
  HANDLER
    RETURN FALSE
  ENDTRAP
  RETURN TRUE
ENDFUNC test'select'select
```

69. Tests that SELECT will accept the two values common between COMALs: "ds:" and "lp:".

SGN

SGN(«numeric expression»)
flag=SGN(number)

Function - Returns -1 if the number is negative. Returns 1 if the number is positive. Returns 0 if the number is 0.

```
FUNC test'sgn CLOSED
  IF SGN(1)<>1 THEN RETURN FALSE
  IF SGN(-1)<>-1 THEN RETURN FALSE
  IF SGN(0)<>0 THEN RETURN FALSE
  IF SGN(1000)<>1 THEN RETURN FALSE
  IF SGN(-0.01)<>-1 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'sgn
```

70. Tests that SGN returns -1, 1 or 0 as appropriate.

SIN

SIN(«numeric expression»)
plot(SIN(num),y)

Function - Returns the sine of the number in radians. *CP/M COMAL allows you to choose radians or degrees, but has radians as the default.*

```
FUNC test'sin CLOSED
  IF SIN(0)<>0 THEN RETURN FALSE
  IF ABS(SIN(PI/3)-SQR(3)/2)>0.000001 THEN RETURN
  FALSE //wrap line
  RETURN TRUE
ENDFUNC test'sin
```

71. Tests the sine trig function.

SIZE

SIZE

Command - Displays the amount of available free memory. Some COMALs display more information, such as program size and data area size. This is an informational display only. See FREE for a function that may be used in a program.

SPC$

SPC$(«number of spaces»)
PRINT SPC$(39)

Function - Returns the number of spaces specified.

```
FUNC test'spc CLOSED
  IF SPC$(5)<>"     " THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'spc
```

72. Tests that SPC$ returns the number of spaces specified.

SQR

SQR(«numeric expression»)
root=SQR(number)

Function - Returns the square root of the number.

```
FUNC test'sqr CLOSED
  IF ABS(SQR(1000)-31.62277)>0.00001 THEN RETURN FALSE
  IF ABS(SQR(8)*SQR(8)-8)>0.00001 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'sqr
```

73. Tests the square roots returned in two ways: subtracting the correct value from the SQR call, and multiplying the SQR with itself and then subtracting the original number. Both should have 0 for an answer.

STEP

STEP «numeric expression»
FOR x=1 TO max STEP 2 DO

Part of FOR statement - Sets the amount the FOR variable is incremented after each loop. If it is negative, the loop variable is decremented rather than incremented, and terminates when the variable value is less than the end amount. The step amount can be an integer or real numeric expression.

Note: a non-integer step size can lead to some "round off" problems due to the way addition is performed on the numbers.

```
FUNC test'step CLOSED
  count:=0
  FOR x:=1 TO 11 STEP 2 DO
    IF x MOD 2<>1 THEN RETURN FALSE
    count:+1
  ENDFOR x
  IF count<>6 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'step
```

74. Tests that a step of 2, starting with an odd number, will only result in odd numbers, and that the correct number of loops are made.

STOP

STOP [«message»]
STOP "now inside PROC remove'blank"

__Statement__ - terminates program execution. Execution may be continued with the CON command. Variables may be displayed or changed before continuing. Lines may also be listed. However, if any lines are added, deleted, or modified the program may not be able to be restarted (due to internal tables).

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not allow a message as part of a STOP statement.

STR$

STR$(«number»)
zip$=STR$(number)

__Function__ - Returns a string that is the equivalent of the number. The number 567 becomes "567". The VAL function does the reverse, converting a string into a number.

```
FUNC test'str CLOSED
  IF STR$(56)<>"56" THEN RETURN FALSE
  IF STR$(1.61803)<>"1.61803" THEN RETURN FALSE
  IF STR$(-13)<>"-13" THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'str
```

75. Tests the STR$ function with an integer, real number, and negative number.

TAB

TAB(«column number»)
PRINT TAB(col), name$

__Function__ - Prints spaces up to the column specified. If that position is already exceeded, it goes to the specified position on the next line. TAB is always part of a PRINT statement.

CP/M COMAL converts the TAB function into: TAB «column number»

```
FUNC test'tab CLOSED
  PRINT TAB(25),"",
  IF CURCOL<>25 THEN RETURN FALSE
  CURSOR 0,1
  RETURN TRUE
ENDFUNC test'tab
```

76. Tests that TAB locates the cursor properly.

TAN

TAN(«numeric expression»)
PRINT TAN(number)

__Function__ - Returns the tangent of the number in radians. *CP/M COMAL allows you to choose radians or degrees, but radians is the default.*

```
FUNC test'tan CLOSED
  IF TAN(0)<>0 THEN RETURN FALSE
  IF ABS(TAN(PI/4)-1)>0.000001 THEN RETURN FALSE
  IF  ABS(TAN(2*PI/3)+SQR(3))>0.000001  THEN  RETURN
FALSE
  RETURN TRUE
ENDFUNC test'tan
```

77. Tests the tangent trig function.

THEN

THEN
IF NOT ok THEN RETURN FALSE
ELIF errors>3 THEN

__Part of IF__ - Part of the IF and ELIF statements. The system will insert the word THEN for you if you omit it.

```
FUNC test'then CLOSED
  IF FALSE THEN RETURN FALSE
  IF TRUE THEN
    RETURN TRUE
  ENDIF
  RETURN FALSE
ENDFUNC test'then
```

78. Tests that THEN is accepted in a single line and multi-line IF statement.

TO

«start num» TO «end num»
FOR x:=1 TO 4 DO

__Part of FOR__ - Part of the FOR statement, separating the start and end numbers.

```
FUNC test'to CLOSED
  FOR x:=1 TO 0 DO
    RETURN FALSE
  ENDFOR x
  RETURN TRUE
ENDFUNC test'to
```

79. Tests that TO is accepted in a FOR statement, and that if the start value excedes the end value the loop is skipped.

TRAP

TRAP // start of error handler
TRAP ESC- // disable stop/break key
TRAP ESC+ // enable stop/break key

Statement - Marks the start of the error trap structure. Also is used to disable/enable the stop/break key.

C64 Power Driver, C64 COMAL 1.0 and Apple COMAL do not have the error handler structure. Alder uses TRAPESC (one word) to set the stop/ break key disable, and 0 or 1 instead of + or -.

```
FUNC test'trap CLOSED
  t:=FALSE
  TRAP
    PRINT 2/0 // division by 0
    t:=FALSE
  HANDLER
    TRAP
      TRAP ESC- //disabled
      TRAP ESC+ //enabled
      t:=TRUE
    HANDLER
      NULL
    ENDTRAP
  ENDTRAP
  RETURN t
ENDFUNC test'trap
```

80. Tests the error trap structure. Then tests that the stop/break key can be disabled and enabled.

TRUE

TRUE
RETURN TRUE

System constant - Always equal to 1 when used as assignment. Other times it means not FALSE (a value that is not equal to 0).

```
FUNC test'true CLOSED
  IF TRUE<>1 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'true
```

81. Tests that TRUE equals 1 when used as an assignment.

UNIT

UNIT [«string expression»]
UNIT data'drive$
drive$:=UNIT$

Statement/Function - Sets the default unit when used as a statement. As a function, it returns the current unit name.

```
FUNC test'unit CLOSED
  RETURN ":" IN UNIT$
ENDFUNC test'unit
```

82. Tests that a proper unit is returned: it must contain a colon (:).

UNTIL

UNTIL «condition»
UNTIL reply$="q"

Statement - Marks the end of the REPEAT structure. Statements inside the structure are executed until the condition is TRUE.

```
FUNC test'until CLOSED
  x:=256
  REPEAT
    x:=x/2
  UNTIL x<1
  RETURN x<1
ENDFUNC test'until
```

83. Test that UNTIL exits a REPEAT loop correctly.

USE

USE «package name»
USE system

Statement - Activates a package that is linked to the program, making all its procedures and function accessible. Package "commands" are not available to CLOSED procedures and functions unless they are imported or another USE command is included in the CLOSED procedure or function.

Amiga COMAL also automatically does a LINK if needed. CP/M COMAL requires quotes around the package name: USE "mouse"

USING

PRINT USING «format»: «var list»
PRINT USING "##> $###.##": x, cash(x)
PRINT AT 8,5: USING "##": item

Special - Part of a PRINT statement allowing formatted output. Within the format string a # reserves a position for each possible digit of the number; a period "." marks the decimal point location; a minus sign "-" is optional reserving a position for the negative sign. On the right of the decimal point, zeroes are padded where necessary. On the left of the decimal point, spaces are padded. All other characters (other than # . -) are printed as supplied. If the number has more digits than reserved, a * is printed in each reserved position.

```
FUNC test'using CLOSED
  PRINT // move to next line
  x=42.8923
  PRINT USING "Test'using -##.#": x,
  IF CURCOL<>11 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'using'using
```

84. Tests that the cursor is at the correct location after a PRINT USING statement is executed.

VAL

VAL(«numeric string»)
age=VAL(reply$)

Function - Returns the numeric value of a numeric string. This allows you to input data as strings, check them for errors, then convert them into numbers. VAL accepts the digits, + and - signs, decimal point, and exponential notation. Leading spaces are ignored by VAL.

Requesting a VAL of a non-numeric string results in an error in most systems. UniComal ignores the rest of the string beginning at the first non-numeric character (but gives an error if it's the first character). C64 Power Driver, C64 COMAL 1.0 and Apple COMAL return 0 as the VAL of non-numeric strings (because they have no error handling system to trap a failed VAL).

```
FUNC test'val CLOSED
  IF VAL("56")<>56 THEN RETURN FALSE
  IF VAL("1.125")<>1.125 THEN RETURN FALSE
  IF VAL("-13")<>-13 THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'val
```

85. Tests that the proper value is returned for an integer, real number and negative number.

WHEN

WHEN «list of values»
WHEN "Jan","jan"
WHEN 1,2

Statement - Provides a specific case within a CASE structure. One or more values are listed after the WHEN. If any match the current case value, its following statements are executed. Statement blocks are indented automatically when listed.

Some COMALs also allow WHEN to be used as a conditional EXIT from the LOOP structure:
EXIT WHEN «condition»

In CP/M COMAL and Amiga COMAL, EXIT and EXIT WHEN statements will exit all loops, not just the LOOP structure. Since this may cause COMMON programs to run incorrectly, we consider it a flaw.

```
FUNC test'when CLOSED
  x:=5
  CASE x OF
  WHEN 5
    x:=6
  WHEN 6
    RETURN FALSE
  OTHERWISE
    RETURN FALSE
  ENDCASE
  RETURN TRUE
ENDFUNC test'when
```

86. Tests that only one WHEN case statements are executed by changing the case variable in one section to a valid value on a following WHEN condition. Only the first WHEN section should be executed.

WHILE

WHILE «expression» [DO] [«statement»]
WHILE NOT EOF(infile) DO process
WHILE errors<3 DO

Statement - Marks the start of a multi-line WHILE structure. As long as its condition is true the statements are executed. If the condition is FALSE right at the start, the statements are skipped over. Some COMALs also allow a one line WHILE statement that does not need an ENDWHILE. The system will insert the word DO if you omit it.

```
FUNC test'while CLOSED
  DIM a$ OF 20
  a$:="never stop"
  WHILE a$<>"stop" DO
    a$:=a$(2:)
  ENDWHILE
  IF a$<>"stop" THEN RETURN FALSE
  RETURN TRUE
ENDFUNC test'while
```

87. Tests that a WHILE loop continues executing until its condition is met.

WRITE

WRITE FILE «file#»[,«rec#»]:«var»
OPEN [FILE] «filenum»,«filename»,WRITE
WRITE FILE 2: name$
OPEN FILE 3,"scores",WRITE

Statement/Command/File Type - Writes data to a file in binary form (not ASCII). As a file type, it specifies that the file is to be written to. Data written by a WRITE FILE statement may be read with a READ FILE statement, and is not compatible between implementations.

ZONE

ZONE [«tab interval»]
ZONE 5
z:=ZONE

Statement/Function - Returns the current zone setting as a function. As a statement it sets the zone to the number specified. The zone determines the interval in tab positions on an output line. The default zone is 1 (a zone at each column). The semicolon is the zone separator (for PRINT statements and at the end of INPUT statements). By default, a semicolon outputs one space (if a ZONE statement is previously used, it outputs spaces to the next zone). When a comma is used as a separator in a PRINT statement (or at the end of an INPUT statement) no spaces are printed, and the cursor remains where it is (null separator).

C64 Power Driver, C64 2.0 cartridge, Alder and CP/M COMAL follow the original standard: the comma as the zone separator (with default zone 0) and the semicolon always outputing one space. If no ZONE statement is used in a program, it will work with the either standard. Alder uses GETZONE as the function that returns the current zone.

```
FUNC test'zone CLOSED
  PRINT // force new line
  z:=ZONE
  ZONE 10
  PRINT "";"";
  IF CURCOL<>21 THEN RETURN FALSE
  PRINT "",
  IF CURCOL<>21 THEN RETURN FALSE
  CURSOR 0,1
  ZONE z
  RETURN TRUE
ENDFUNC test'zone
```

88. Tests that the zone settings work with the semicolon zone separator.

Common COMAL is a trademark of COMAL Users Group, U.S.A., Limited.

Special Notes

C64
Power Driver is the current disk loaded COMAL for the C64 or C128. It has more commands, more user memory, and more capabilities than COMAL 0.14. It has been upgraded to C64 COMAL 1.0, but that upgrade has a fatal flaw with the NEW command. After months of efforts to find and fix this bug, the project was postponed and it will be rewritten from scratch.

The COMAL 2.0 cartridge is more powerful, faster, and easier to use, but its programs require the cartridge to be run, while Power Driver programs can be compiled with the compiler in Power Box.

IBM PC, compatibles, and PS/2
UniComal IBM PC COMAL 2.2 is the current version of COMAL for the IBM. The Mytech COMAL project was cancelled.

CP/M
CP/M COMAL 2.10 is the original and current release of COMAL for CP/M. It will run on the C128 in CP/M mode. A demo disk is available if you wish to test it (and its installation).

Amiga
AmigaCOMAL from ComWare is being distributed in Scandanavia and Germany. It will not be released in the USA / Canada until fall according to ComWare. We previously incorrectly referred to it as German Amiga COMAL. Borge Christensen, founder of COMAL, is involved in this project. The system is written in machine code for speed.

Another company, Alder (formerly Mytech), has a COMAL for the Amiga and said that it is shipping now, however, we have not yet seen it ourselves. Alder COMAL is written in C.

When we refer to Amiga COMAL in our articles, we are usually referring to ComWare AmigaCOMAL (Germany/Denmark). The other implementation from Alder is referred to as Alder COMAL (formerly Mytech COMAL).

Apple
Apple COMAL 1.0 should be released by the time you read this. There is no graphics in this first release. A future release of version 2.0 is being planned now, and will include more features.

PET
One of the original implementations of COMAL was on the Commodore PET computer. It is still available and being used. Disk loaded PET COMAL 0.14 is current. A plug in board for version 2.0 was also available.

Proposed COMAL Multilevel Standard
by Joel Ellis Rea ("COMALite J" on QuantumLINK)

First there was the COMAL Kernal. Now there is the Common COMAL proposed standard. COMAL Kernal was considered "too small"—most versions of COMAL went far beyond the Kernal specifications, while generally remaining upwardly compatible with it. For example, even C-64 COMAL 0.14 went beyond it by adding the `KEY$` function and the `USING` clause to the `PRINT` statement, not to mention all of those graphics extensions. Common COMAL started out as a proposed "high-level" pseudo-standard to include such implementations as the COMAL 2.0 cartridge for the C-64, and UniCOMAL® IBM PC COMAL 2.0. But now, new implementations are coming out, many of which change old established aspects of the language such as what the **,** and **;** separators mean in the **PRINT** statement, etc. Thus, C-64 COMAL 2.0 is no longer Common COMAL-compatible in this and several other important respects. Also, many new keywords are added that are common enough words that some older programs may have used these words as variable, label, `PROC`edure, `FUNC`tion, or package names. For example, adding a new keyword called `RECORD` would make it hard to convert older programs which used the word `record` as a name. The good thing about standards is that they are, in a word, "standard." That means that theoretically* a program written to the standard would run as-is on any implementation which was compatible with the standard. The bad thing is that they often inhibit progress, because additions to the language would either be non-standard, or would require changing the language standard and thus making all previously standard programs and language implementations non-standard themselves!

My suggested solution to this dilemma is a *multi-level* standard! Currently, I have defined three levels. These would be called *Level I COMAL, Level II COMAL,* and *Level III COMAL.* As additions come out, they can be implemented as higher level standards! Basically, Level I COMAL is the same as the COMAL Kernal. Thus, COMAL 0.14 would be 95% compatible with the Level I standard (it lacks the `IMPORT` statement which *is* in the Kernal!), with extensions. Level II COMAL is the Common COMAL standard, perhaps with some minor changes to make it more compatible with previous "2.0"-level COMALs. Level III COMAL contains some favorite "blue-sky" enhancements I have long wanted, and is described more fully in this document. It would be nice if the new Amiga COMAL from Europe would be adapted to fit this standard.

Ideally, any program written to a lower level standard would run on that standard, *and* on any higher-level standard, without modifications. In practice, some names would may to be changed in order to protect the innocent new keywords. To implement this, I have established some ground rules that new implementations and proposed new levels should follow:

- No higher level standard should contradict a lower level standard without *good* reason!
- If a new feature can be implemented with reasonable semantics by extending or re-using an existing keyword rather than creating a new one, this should be done. For example, my entire Level III standard adds only *three* new keywords: `ENDDIM`, `ENDWITH`, and `WITH`, only one of which is at all likely to have been previously used much as names in existing programs!

To aid in using a higher-level implementation to develop programs which follow a lower-level standard, I propose that all new implementations which have *any* enhancements above Level I should add the `SETLEVEL` editor command. Like all commands (as opposed to statements, functions, clauses, structures, etc.) this is not a part of the language itself. Rather, it is a means of warning and/or forbidding features not implemented in a given lower level. Thus, a person with Level III would not need a spec sheet on Level I in order to write a program which is guaranteed to be compatible with the Kernal/Level I.

My suggested improvements for Level III are mainly concerned with greatly increasing what can be done with data. I implement structured variables and arrays (like the *records* in Pascal, the *structs* in C, the structured *data divisions* in COBOL, etc.), plus a nifty feature called "virtual arrays," which can be used in combination with the structured data. I also obviate much of the need for pointers by allowing variables, arrays, and structured variables and arrays to be dimensioned to a specific location in memory. And now, the Level III enhancements:

Non-Keyword Enhancements—These enhancements do not have keywords of their own, and thus are placed here. These include variable type identifier characters, and one new operand. They are as follows:

***Variable Type Identifiers** (all levels as well as extensions):*

«name»	Real (or non-variable *«name»*)	//All levels
«name»$	String	
«name»#	Two-byte signed Integer (-32,767:32,767)	
«name»!	Single byte (0:255)	//Level III only
«name»%	Two-byte Word (unsigned integer—0:65,535)	
«name»&	Four-byte signed Long Integer (-2,147,438,648:2,147,438,647)	
«name»@	Four-byte Longword (0:4,294,967,296—useful as pointers on 32-bit CPUs.)	
«name».	Structured variable or array (see `DIM` and `ENDDIM`)	

New Operator:

@*«var»*	Returns the address of the memory location containing the first byte of the specified variable's value. Generates `Illegal Operation` if the variable is "virtual." Mainly used with `DIM AT`, but could be handy with `PEEK`, `POKE`, and/or `SYS`.

AT—*Clause:* For all Levels, is used with a `PRINT` statement directed to the screen (or to a window) to position the cursor to a specific location before the start of the output. For Levels II and III, is used with an `INPUT` statement directed to the keyboard (or to a window) to position the cursor to a specific location before the start of the input, and to optionally limit the width of the input field. For Level III only, is used with a `DIM` or `ENDDIM` to specify that a non-"virtual" variable, array, structured variable, or structured array is to reside at a specific location in memory, usually at an absolute location or as an alias to another name.

Syntax—*Clause (for more details, see individual statement listings):*

```
PRINT [FILE «filenum»[,«recnum»[,«bytenum»]]: ][AT «row»[,«col»]: ]
    [«printlist»]                                              //All levels
INPUT [FILE «filenum»[,«recnum»[,«bytenum»]]: ][AT «row»[,«col»[,«width»]]: ]
    [«promptstring»: ]«variable»{[, «variable»]...}            //Levels II and III
DIM {«name»[[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
    |![(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
    |#[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
    |%[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
    |&[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
    |@[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
    |$[(«intexp»[:«intexp»][{ ,«intexp»[:«intexp»]}])] OF «intexp»]
    |[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])] . OF «name».
    [ AT «longwordexp»][,...]}                                  //Level III only
ENDDIM [«name»[.][ AT «longwordexp»]]                           //Level III only
```

Examples—*See individual statement listings.*

CLOSE—*Statement:* Closes a specified open file, all open files, or multiple specified open files (Level III only). If no *«file numbers»* are specified, all open files are closed. If one or more *«file numbers»* are specified and the `FILE` clause keyword is omitted, it will be supplied automatically.

Syntax—*Statement:*

```
CLOSE [[FILE] «file number»]                        //Levels I and II
CLOSE [[FILE] «file number»[, «file number»]...]    //Level III
```

Examples—*Statement:*

```
CLOSE                                  //Close all open files
CLOSE FILE 8                           //Close file 8 only
CLOSE FILE output'file#                //Close file output'file
CLOSE FILE 10, 20                      //Level III only
CLOSE FILE infile#, outfile#           //Level III only
```

DIM—***Statement:*** Creates numeric arrays, string variables, and/or arrays of strings, and allocates space for them. It can also allocate simple numeric variables (though this is unnecessary due to COMAL's implicit allocation of simple numeric variables upon first assignment, except as a field of a structured variable). In Level III only, it can define structured variables and/or arrays by declaring them to have an equivalent structure to that of a structured variable or array previously defined in a `DIM` structure, and allocate space for them. The space may be allocated in RAM (default), in a special `DIM`-type "virtual memory" block-storage file (Level III only), or at a particular memory location (Level III only)—either absolute or relative to a previously defined name of any type. By using the `AT` clause, Level III can also allocate a variable, array, structured variable, or structured array to reside at some specific location in memory.
***Structure** (Level III):* Defines a structured variable and/or array, and allocates space for it.The space may be allocated in memory (default), at some specific location in memory, or in a special `DIM`-type "virtual memory" block file.
***Clause** (Level III):* As a parameter to the **OPEN** statement, specifies that the opened file (must be on a block storage device) is a special "virtual memory" file.

Syntax—*Statement:*
```
DIM {«name»[ [(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |#[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |$[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])] OF «intexp»] [,...]} //Level I & II

DIM [FILE «filenum»:]{«name»[[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |![(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |#[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |%[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |&[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |@[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])]
     |$[(«intexp»[:«intexp»][{ ,«intexp»[:«intexp»]}])] OF «intexp»]
     |[(«intexp»[:«intexp»][{,«intexp»[:«intexp»]}])] . OF «name».
     [ AT «longwordexp»][,...]}                              //Level III only
```

Syntax—*Structure (Level III only):*
```
DIM [FILE «filenum»:]«name»[[(«intexp»[:«intexp»][{ ,«intexp»[:«intexp»]}])] . OF
   {«simple, non-file DIM statements and/or DIM structures»...}
ENDDIM [«name».][ AT «longwordexp»]
```

Syntax—*Clause of* `OPEN` *statement (Level III only):*
```
DIM [READONLY|WRITEONLY]
```

Examples *(all levels):*
```
DIM mystring$ OF 100 //Simple string allocation, 100 characters
DIM myarray#(100) //Simple integer array, 101 elements (0..100)
DIM year'sales(1980:1999) //Double-bounded real array, 1 dimension, 20 elements
DIM my'string'array$(10) OF 32 //String array
DIM mymatrix(10,10) //2-dimensional real array, 121 total elements (11x11)
DIM mymatrix#(1:10,1:20) //2-dimensional integer array, 200 total elements
DIM screen'line$ OF screen'width# //Expression instead of constant length
DIM scores(1:class'size#) //Expression instead of constant upper bound
DIM strng$ OF 64, ary#(3,2,150), stry$(100,5) OF 8//Multiple items
```

Examples *(Level III only):*
```
DIM myinteger# AT $0001FE00                          //Absolute location
DIM strng$ OF 128 AT keyboard'buffer@                //Absolute location
DIM strng$ OF 4096, length% AT @strng$               //Alias location
DIM longint&, hiword% AT @longint&, loword% AT @longint&+2
```

The last two examples above demonstrate the `AT` clause in conjunction with the `@` operator which returns the address of any following name. This is handy for creating *aliases.* The last example showed how arithmetic could be used to alias a part of a variable, etc., other than from the start of it.

The following short program demonstrates simple "virtual memory" arrays. The virtual string array "state$" has 1,000 elements of 2 bytes each, for a total of 2,000 bytes in the "virtual memory" file named "zip.codes". The file itself is assumed to have been previously created, and already containing the 2-letter state abbreviation for each 3-digit ZIP code prefix.

```
   OPEN FILE 10, "zip.codes", DIM READONLY
   DIM FILE 10: state$(0:999) OF 2
   LOOP
     INPUT "ZIP code (""-1"" to exit)? ": zip'code
     EXIT WHEN zip'code<0
     zip# := zip'code/100
     //The following statement performs the actual file reads automagically!
     PRINT zip'code;"is in the state of";state$(zip#)
   ENDLOOP
   CLOSE FILE 10
```

The following program segment defines a structured variable consisting of an integer, a real, an integer, 2 8-byte strings, and a 1-dimensional integer array of 3 elements, in that order:

```
   DIM vital'stat. OF
     DIM height'ft#, height'in
     DIM weight#
     DIM hair'color$ OF 8, eye'color$ OF 8
     DIM measurements#(1:3)
   ENDDIM vital'stat
```

The following line defines a one-dimensional structured array where each element is structured identically to the structured variable "vital.stat" above. This is great for sorting, since the single variable "vital.stat" can be used as a temporary for exchanges between array elements.

```
   DIM class'stats(1:class'size#). OF vital'stat.
```

If such a temporary variable is not needed, and the array is the only one of its structure, the array can be allocated and its structure defined in a single `DIM` structure, as follows:

```
   DIM class'stats(1:class'size#). OF
     DIM height'ft#, height'in
     DIM weight#
     DIM hair'color$ OF 8, eye'color$ OF 8
     DIM measurements#(1:3)
   ENDDIM class'stats.
```

Structured arrays can also be dimensioned to reside in a "virtual memory" file:

```
   OPEN FILE 20, "class.dat", DIM
   DIM FILE 20: class'stats(1:class'size#). OF
     DIM height'ft#, height'in
     DIM weight#
     DIM hair'color$ OF 8, eye'color$ OF 8
     DIM measurements#(1:3)
   ENDDIM class'stats.
```

The following program demonstrates using a virtual memory file with the above structure, with COMAL automagically doing both the reading and writing of the data in the file as necessary:

```
OPEN FILE 20, "class.dat", DIM
DIM FILE 20: class'size# //First item in file is single int var holding array length

DIM FILE 20: class'stats(1:class'size#). OF //Next is the array itself
   DIM height'ft#, height'in
   DIM weight#
   DIM hair'color$ OF 8, eye'color$ OF 8
   DIM measurements#(1:3)
ENDDIM class'stats.

DIM r$ OF 1

REPEAT
   INPUT AT 0,0,1: "View, Change, or Quit? ": r$
   CASE r$
   WHEN "V", "v"
      view'stats
   WHEN "C", "c"
      change'stats
   OTHERWISE
   ENDCASE
UNTIL r$ IN "Qq"

CLOSE FILE 20

PROC view'stats

   REPEAT

      INPUT "Which student # (""0"" to exit)? ";s#

      IF s#>class'size# THEN
         PRINT "Sorry, only"; class'size#; "students!"

      ELIF s#>0 THEN

         WITH class'stats(s#). DO //Actual file reads performed here automagically!
            PRINT height'ft#, "'"; height'in, """ tall."
            PRINT "Weighs"; weight#; "pounds."
            PRINT "Has";hair'color$;"hair and";eye'color$;"eyes."
            PRINT "Measurements: ";
            FOR i#:=1 TO 3 DO PRINT measurements#,"""";
         ENDWITH class'stats(s#).

         PRINT

      ENDIF

   UNTIL s#=0

ENDPROC view'stats
```

```
PROC change'stats

   REPEAT

      INPUT "Which student # (""0"" to exit)? ";s#

      IF s#>class'size# THEN
         PRINT "Sorry, only"; class'size#; "students!"

      ELIF s#>0 THEN

         INPUT "Height (in total inches): ": inches

         WITH class'stats(s#). DO //Actual file writes performed here!
            height'ft# := inches DIV 12
            height'in := inches MOD 12
            INPUT "Weight (in pounds): ": weight#
            INPUT AT 0,0,8: "Hair color: ": hair'color$
            INPUT AT 0,0,8: "Eye color:  ": eye'color$
            INPUT "Chest (in inches): ": measurements#(1)
            INPUT "Waist (in inches): ": measurements#(2)
            INPUT "Hips  (in inches): ": measurements#(3)
         ENDWITH class'stats(s#).

         PRINT

      ENDIF

   UNTIL s#=0

ENDPROC change'stats
```

The **AT** clause can be used with structures as well, either to use a structure as an alias to something else, or to use something else as an alias to a structure, or to define a structure as residing at an absolute location in memory, etc. If the structure is defined in a **DIM** structure and allocated to a set or alias location using **AT**, the **AT** clause is placed on the end of the corresponding **ENDDIM** statement, rather than on the **DIM** statement itself. Of course, **AT** cannot be used with **FILE**, nor may **AT** be specified *inside* a structure. Some examples:

```
DIM alias'record. OF my'record. AT @my'record.
DIM absolute'record. OF my'record AT $00C08000
DIM sixteenth'byte$ OF 1 AT @my'record.+$0F

DIM new'structure. OF
   DIM byte'part!
   DIM integer'part#
   DIM word'part%
   DIM real'part
   DIM longword'part@
   DIM longint'part&
   DIM string'part$ OF 32
ENDDIM new'structure. AT buffer'address@
```

ENDDIM—***Statement*** *(Level III Only):* This statement has only one purpose: to mark the end of a `DIM` structure which defines a structured variable or array. The *«name»* after the `ENDDIM` keyword is optional, but if specified it must match the name of the structured variable or array defined by the `DIM` structure. The dimensions of the array are *not* duplicated here. If either the *«name»*, the terminating period, or both are omitted, the COMAL `SCAN` compiler pass will add the omitted syntax item(s) automatically the first time the program is `RUN` or `SCAN`ned. An `AT` clause may also be used to denote that the non-virtual structured variable or array resides at some specific location in memory.

Syntax*—Statement:*

```
ENDDIM [«name»[.][ AT «longwordexp»]]
```

Examples:*—See* `DIM` *structure.*

ENDWITH—***Statement*** *(Level III Only):* This statement has only one purpose: to mark the end of a `WITH` structure which simplifies references to a structured variable or array. The *«structure reference»* after the `ENDWITH` keyword is optional, but if specified it must match the one specified by the `WITH` structure. If either the *«structure reference»*, the terminating period, or both are omitted, the COMAL SCAN compiler pass will add the omitted syntax item(s) automatically the first time the program is `RUN` or `SCAN`ned.

Syntax*—Statement:*

```
ENDWITH [«structure reference»[.]]
```

Examples:*—See* `WITH` *structure.*

FILE—***Clause:*** A required element in `CLOSE` and `OPEN` statements (the `FILE` keyword will be automatically supplied by the COMAL compiler if omitted), and an optional element in `DIM` (Level III only), `INPUT`, `PRINT`, `READ`, and `WRITE` statements. For the `CLOSE`, `DIM`, and `OPEN` statements, this clause specifies the *file number* (or *«filenum»*) of the file being closed, dimensioned to, or opened. For the `INPUT`, `PRINT`, `READ`, and `WRITE` statements, this clause not only specifies the *«filenum»* itself, but also optionally specifies a *record number* (*«recnum»*) and an optional *starting byte number* (*«bytenum»*) for the I/O operation to begin at. See the individual statement listings for more details on usage and syntax, plus examples.

Syntax—*Clause (for more details, see individual statement listings):*

```
CLOSE [[FILE] «filenum»[{, «filenum»...}]]                //Multi files, level III.
DIM [FILE «filenum»:]...                                  //Level III only.
INPUT [FILE «filenum»[,[«recnum»][,«bytenum»]]: ]...
OPEN [FILE] «filenum», «filename»[, «mode»]
PRINT [FILE «filenum»[,[«recnum»][,«bytenum»]]: ]...
READ [FILE «filenum»[,[«recnum»][,«bytenum»]]: ]...
WRITE [FILE «filenum»[,[«recnum»][,«bytenum»]]: ]...
```

Examples:*—See individual statement listings.*

OPEN—*Statement:* This statement establishes a connection between a *file* (a collection of data located externally to the program itself, to be either input into the program, or output from it, or some combination of the two) and the program itself by allowing all further references to the file to be via a simple integer value called a *file number*. Files are of two main types: *sequential* and *random access*. Sequential files may be located on any type of device, while random access files may only be located on *block-type* devices such as disk drives and RAM disks. Level III COMAL implements a third type of file, called *virtual-memory* files, which is actually a simplified form of random access, and thus must reside on a block-type device. The syntactic elements of the `OPEN` statement include the *«filenum»*, or the file number itself, which can be any integer in the range 0:255. Some implementations reserve the numbers `0`, `1`, `2`, and/or `255` for special system files. To maintain compatibility, programs which are intended to be portable should only use file numbers in the range 5:250. The *«filename»* is a system-dependent *file specification* which specifies where the file is located. Sequential files may refer to devices such as the keyboard, screen, printer, serial ports, parallel ports, MIDI ports, A/D ports, etc., depending on the system hardware available. References to block devices must include a file name, possibly with a *path* on systems with hierarchical directories.

The *«mode»* tells whether the file is to be accessed as sequential or random access, and what operations are allowed. Valid modes for Level I include *none*, `APPEND`, `RANDOM` *«reclen»*, `RANDOM` *«reclen»* `READONLY`, `READ`, and `WRITE`. Valid modes for Level II include all of those and `RANDOM` *«reclen»* `WRITEONLY`. Valid modes for Level III include all of the above plus `DIM`, `DIM READONLY`, and `DIM WRITEONLY`. The *none*, `APPEND`, `READ`, and `WRITE` modes specify sequential access, meaning that the file is read from the beginning on. If no mode is specified, the file is opened as sequential access, permitting both read and write operations: if the file does not exist, an empty one of that name is created, while if a file of that name already exists, it is opened and the *file pointer* set to point to the first byte of the file. If the `APPEND` mode is specified, the file is opened as sequential access, and only write operations (`PRINT FILE` and `WRITE FILE`) are allowed: if the file does not exist, an empty one of that name is created, while if a file of that name already exists, it is opened and the file pointer set to point to the first position after the current end of the file, so that any new data written to the file will be added ("appended") to the end of the file. If the `READ` mode is set, the file is opened as sequential access, and only read operations (`INPUT FILE` and `READ FILE`) are allowed: if no file of that name exists, a "`File Not Found`" error is returned (which may be trapped in Level II and III by use of the `TRAP/HANDLER/ENDTRAP` structure), while if a file of that name already exists, it is opened and the file pointer set to point to the first character. If the `WRITE` mode is specified, the file is opened as sequential access, and only write operations are allowed: if the file does not exist, an empty one of that name is created, while if a file of that name already exists, it is deleted and a new empty file of the same name is created.

The `RANDOM` *«reclen»* and `DIM` modes, with or without either `READONLY` or `WRITEONLY`, specify random access. `RANDOM` *«reclen»* specifies "standard" random access, which permits the `FILE` clause of the I/O statements `INPUT FILE`, `READ FILE`, `PRINT FILE`, and `WRITE FILE` to specify the *«recnum»* parameter, to allow the operation to occur at a particular *record* in the file—each such record contains precisely *«reclen»* bytes. `DIM` specifies a new "virtual memory" random access mode, which inhibits explicit I/O operations on the file entirely— instead, the `DIM FILE` statement or structure allocates one or more variables (simple, array, structured, and/or structured array) to blocks and bytes in the file such that merely referencing such a variable automagically reads from the correct portion of the file, while merely assigning a value to such a variable automagically writes to the correct portion of the file. For either, omitting `READONLY` and `WRITEONLY` specifies that both reading from and writing to the file are permitted, and that if no file of the specified name exists, one of the proper name and format is created, while if one does exist, it is opened and its data made available to the program. If `READONLY` is specified, only read operations from the file are permitted, and the `OPEN` succeeds only if the file already exists, while if no such file exists, a "`File Not Found`" error is returned. If `WRITEONLY` is specified, only write operations to the file are permitted, and any existing file of that name is deleted, and a new one of the specified name and type is created.

Some implementations add further clauses to the `OPEN` statement, but these are system- dependent and are not part of the COMAL standards. For instance, Commodore 8-bit COMAL 0.12, 0.14, and Power Driver add a `UNIT` clause for specifying devices.

Syntax—Statement:

`OPEN [FILE]` *«filenum»*`,` *«filename»*`[,` `{RANDOM` *«reclen»*`|READ|WRITE}]` //Level I

`OPEN [FILE]` *«filenum»*`,` *«filename»*`[,` `{RANDOM` *«reclen»* `[READONLY]|READ`
`|WRITE}]` //Level II

`OPEN [FILE]` *«filenum»*`,` *«filename»*`[,` `{DIM [{READONLY|WRITEONLY}]`
`|RANDOM` *«reclen»* `[{READONLY|WRITEONLY}]|READ|WRITE}]` //Level III

Examples—All levels:

`OPEN FILE 10, "lp:", WRITE` //Opens line printer for output on some systems.

`OPEN FILE 7, "error.log", APPEND` //Opens file "ERROR.LOG" on the current block device, as a sequential output file. If it already exists, new data will be added to the end.

`OPEN FILE 20, "a:cledger.dat", RANDOM 120` //Opens a file from block device "A:" named "CLEDGER.DAT" as a random access file with 120-byte records.

`OPEN FILE 8, filename$, READ` //Opens a file specified by the string variable "filename$" (possibly input from the user) as sequential read only, returning a **TRAP**pable (if Level II or III) error if the file does not already exist.

`OPEN FILE 12, "aux:"` //Open device "AUX:" (possibly a communications port or device driver) for sequential bi-directional access.

Examples—Level II & III only:

`OPEN FILE 5, "c:\system\users.dat", RANDOM 32 READONLY` //Opens an existing file "USERS.DAT" in subdirectory "SYSTEM" on block device "C:" as read-only random access with 32-byte records, returning a **TRAP**pable error if the file does not exist.

Examples—Level III only:

`OPEN FILE 50, "df1:/mystuff/datafile", RANDOM 64 WRITEONLY` //Creates a new random-access file with 64-byte records (by erasing any existing file of the same name) and opens it in a mode permitting only write accesses.

`OPEN FILE 25, "dh0:customers", DIM` //Opens a file (or creates a new one if it does not already exist) on a block device as a "virtual memory" random access file.

`OPEN FILE 8, "login:userlist", DIM READONLY` //Opens an existing "virtual memory" file (returning an error if not found), and permitting only references to the variables residing in the file.

`OPEN FILE 12, "/gl/chart_of_accounts_data", DIM WRITEONLY` //Creates a new "virtual memory" file (by erasing any existing file of the same name) and opens it in a mode permitting only assignments to the variables residing in the file.

SETLEVEL—***Command*** *(Required on Level III and up, may be implemented on any level):* This command helps overcome the fact that adding features to a language like COMAL either makes incompatibilities possible (by introducing new keywords which previous programs may have used as names), or weakens the effectiveness of the dynamic syntax error checking (by using old keyword in new ways, ways which would have been erroneous in a previous version). Also, of course, a person may wish to use a higher-level system to write programs, but may wish to be sure that they will be compatible with lower-level implementations.

SETLEVEL is an editor command, like **LIST** or **LOAD** or **SETEXEC**. It is not technically part of the COMAL language itself; rather, it is a part of the environment in which one writes COMAL programs. It takes only one parameter, which is optional. This parameter, *«level»*, is an integer which determines which level you want the system to emulate, and whether violations of that level are to be forbidden entirely (*«level»* is negative) or simply warned against (*«level»* is positive). Setting *«level»*to zero disables level checking entirely, and allows the full features of the given implementation to be used without warning, including extensions to any and all standards. If *«level»* is omitted, the value last used is displayed. If none was given previously, a default set by a configuration or installation program is used, or "0" if none has been used. **SETLEVEL** only flags or restricts the COMAL language itself and its elements, such as *statements*, *structures*, *functions*, *clauses*,*operators*, etc. It does not flag nor restrict other editor commands, or other features of the COMAL environment.

Syntax*—*Command:

SETLEVEL [*«level»*]

Examples:

The following can be considered as a session in COMAL Level III with a standard line-oriented editor. Characters typed in by the user are underscored.

```
setlevel -1   //Restricts to COMAL Kernal (Level I standard).
500 loop
"LOOP" not allowed in Level 1  //Cursor returns to line 500, user cursors down.
setlevel 1  //Warns of incompatibilities with COMAL Kernal.
500 loop
"LOOP" incompatible with Level 1  //Line 500 is accepted and compiled.
setlevel 2  //Warns of incompatibilities with Common COMAL (Level II)
500 loop
510 exit when key>""
520 endloop
530
540 open 12,filename$,dim
"DIM" clause to "OPEN" incompatible with Level 2
540 open 12,filename$,random 32
display
. . .
LOOP
   EXIT WHEN KEY$>""
ENDLOOP

OPEN FILE 12, filename$, RANDOM 32
. . .
setlevel
2

setlevel -3
550 dim button^ to myrecord.  //Possible pointer-type extension
"TO" clause to "DIM" not allowed in Level 3
setlevel 0
550 dim button^ to myrecord.
560 button^=newpointer(myrecord(4).)
```

WITH—***Structure(Level III Only):*** This is a unique structure whose purpose is different from other structures. Other structures include *branching* structures such as `IF/THEN/ELIF/ELSE/ENDIF` and `CASE/WHEN/OTHERWISE/ENDCASE`; *looping* structures such as `FOR/ENDFOR`, `LOOP/EXIT /EXIT WHEN/ENDLOOP`, `REPEAT/UNTIL`, and `WHILE/ENDWHILE`; and *subprogram* structures such as `FUNC/RETURN/ENDFUNC` and `PROC/RETURN/ENDPROC`. Other unique structures include the `DIM/ENDDIM` and the `TRAP/HANDLER/REPORT/ENDTRAP` structures, which, like `WITH/ENDWITH`, are alone in their types. But even so, `TRAP` joins the branching, looping, and subprogram structures in being able to change the flow of the program, while `DIM` joins the subprogram structures in defining a section of code into a named unit of some sort. `WITH` does none of these things. Instead, it provides a shorthand means of referring to long, complex structured variables or arrays. If a section of code is going to be using a given structure often, enclosing it in a `WITH/ENDWITH` structure makes the code much easier to type in and to read, at the expense of slowing down the `SCAN` compiler prepass stage somewhat.

The *«structure reference»* is the name of any `DIM`ensioned structured variable or array. If the structured data is nested as part of another structured variable or array, the `WITH` structure must either be enclosed in another `WITH` structure specifying the "parent" structure, or the complete structure reference must be given.

Syntax:—Structure:

```
WITH «structure reference». [DO]
   «COMAL statements»
ENDWITH [«structure reference»[.]]
```

Examples:

```
DIM english'height. OF
   DIM feet!
   DIM inches                                     //Structure reference
ENDDIM english'height.

DIM physical'stats. OF
   DIM height. OF english'height.
   DIM weight%, hair'color$ OF 8, eye'color$ OF 8
ENDDIM physical'stats.

...
inches:=5                                         //Simple variable
PRINT inches                                      //Prints "5"
...

WITH physical'stats. DO

   weight%:=185; hair'color:="Wm Black"; eye'color:="Dk Brown"
   PRINT inches                                   //Prints "5"

   WITH height. DO
      feet!:=6; inches:=1.5
      PRINT inches                                //Prints "1.5"
   ENDWITH height.

   PRINT inches;height.inches                     //Prints "5 1.5"

ENDWITH physical'stats.

PRINT inches;physical'stats.height.inches         //Prints "5 1.5"

WITH physical'stats.height. DO
   PRINT inches                                   //Prints "1.5"
ENDWITH physical'stats.'height
```

ViewPort

by Paul Keck

How many times have you seen it? That blank stare, then the spark of (false) recognition in their faces. "*COBOL? Isn't that the language for business programmers?*" No, you explain. Co-MAL. It's got the structure of Pascal and... there they go. Not listening again.

How can we COMALites tell others about our wonderful language? My experience has been that whenever you start to define COMAL in terms of other languages, people tend to remember the worst possible aspects of those other languages and throw them back at you.

"*Pascal? I hate all those semicolons. And the TYPE statements, and you know, no variable is REALLY local.*"

"*Whaddya mean, 'Familiarity of BASIC'? I started out on C. Now THERE's a language.*"

"*Logo? That's for kids.*"

"*Well, maybe it's easy to learn, but what can you do with it besides write games?*"

The only way I can ever get my message across is to say something cryptic like "*It's a really great language.*" Then: "*Here, look at this program.*" Once I get a person to look at a well written COMAL program, they get curious. "**Easy curves**", the one that first got me interested in COMAL, is great for this. You'll start hearing remarks like:

"*You mean that's the WHOLE program?*"

"*Where are the POKEs?*"

"*That's for kids.*"

Well, everybody isn't easily pleased. For the tougher cases, or for people you can't sit down in front of your computer, try a different strategy. Include a COMAL system on the back of a disk you are giving them for some other reason. My favorite for this is a VT100 emulating Kermit terminal program. Many students around my university own Commodore 64s (though they are bullied into denying it), and so are happy to have an alternative to buying an expensive VT100 or Zenith terminal. And, of course, on the flip side is COMAL. Write something like "**type LOAD"FASTBOOT",8 and RUN**" on the label. This will arouse every computer user's natural curiosity. Sooner or later, they will try it out. If you know the person's interests, put some programs on the disk that you know they would appreciate. If it's a teacher, include educational programs; if it's a workaholic, some database or graphing programs; if it's a teenager, put in a few bitmaps of scantily clad women. If you don't know anything about them, try the Auto Run Demo Disk, or The Best of COMAL. I have used this tactic to distribute twenty or so COMAL disks to unsuspecting users, and I've gotten back positive feedback from over half.

One of the best ways to track down "victims" is through bulletin boards. Whenever someone expresses dissatisfaction with whatever language they are working with, jump right in and offer them a great COMAL program that does just what they want. For people who accidentally scratched an important file, give them "**dir manipulator**" or "**disk'editor**". Tell them how easy it is to draw great graphics in COMAL. Or how the "IN" operator and substring capabilities will allow them to solve their problem with text handling. If they still seem reluctant, remind them that they have absolutely nothing to lose by trying what you offer. Mention the word FREE a lot when talking about COMAL programs. It may seem a little bit tacky, but helping somebody out of a tight spot can really make them take a good look at a new thing. Since we all believe that COMAL is the best computer language on the block, why be shy about telling people?

Finally, if you have power, use it. Teachers- have your students write their programs in COMAL. BBS operators- write a great utility in COMAL and make life hard to live without it. User group leaders (or members)- always be on the lookout for possible converts. Many novice users show up at user group meetings begging for someone to show them how to exploit their new computer to its fullest. Be the person to show them the way. As the old saying goes, don't hide the light of COMAL under a bushel! Let it shine!

Power Driver Memory Locations

by R. Hughes

	0.14	PD
Printer dev	26129	27507
Printer 2nd addr	26131	27509
Disk dev# for CAT	27013	28446
Turtle x'pos(hi)	27255	28742
Turtle x'pos(lo)	27256	28743
Turtle y'pos	27260	28747
Turtle heading(hi)	27277	28764
Turtle heading(lo)	27278	28765
Turtle visible	27295	28782
Pen-state (Up/Dwn)	27333	28820
Graphic Set 0/1	27261	28748
Sprite on/off bits	27276	28763

Student Mastermind

by Christine and Ray Carter

I. Starting Date -
   January 1, 1989

II. Problem -
   Construct a computer program to carry out the functions of the game "Mastermind".

III. Procedure -
   A. First, my father and I analyzed the steps taken in a game of Mastermind.
      1. To start the game, have the computer select a combination of four colors from the six available colors: red, green, blue, yellow, white or black.
      2. The player then selects four of the available six colors in any combination.
      3. The computer then analyzes the players choice, and puts a black dot for each correct color in the correct place, and a white dot for each correct color in the wrong place.
      4. The computer and player then repeat steps 2 and 3 until the player has run out of turns (ten chances), or correctly guessed the code.
   B. We then developed a flowchart to show the major steps in the program.
   C. My father and I then proceeded to write a computer program to implement the flowchart.

IV. Observation -
   When working on a computer program, I observed that there are several stages to completing a correct program. You must first analyze and describe the problem. Next, a flowchart is helpful to visualize the major steps. Then, the program is written and entered into the computer. Finally you must make additions, changes and other sorts of corrections until the program works properly.

```
//submitted by christine carter
//and ray carter - sub(1014)
//
dim mind$ of 4, guess$ of 4
dim colors$ of 6
dim ab$ of 3, yn$ of 1
colors$:="rgbywk"
initl //initialization routine
repeat
  page
  nguess:=0
  slct(mind$) //routine to choose random set of colors
  correct:=false
  repeat
    getguess(guess$) //get players guess
    nguess:+1
    report //grade players guess
  until (nguess=10 or correct)
  goagain(yn$) //find out if she wants to play again
until yn$="n"
stop
proc initl
  print chr$(142) //select upper/graphics
  border 0
  background 12
  pencolor 1 //set text color to white
  randomize //set random number generator
  page //print out instructions
  print "welcome to the commodore 64 version of"
  print "m a s t e r m i n d"
  print
  input "press return to see instructions: ": yn$
  print "the computer will select a four color"
  print "sequence of different colors"
  print "chosen from red, green, blue, yellow,"
  print "black, and white."
  print
  print "you then have ten chances to guess the"
  print "correct colors and sequence."
  print
  print "enter your guesses as four letters"
  print "using r for red, g for green, y for"
  print "yellow, w for white, or k for black."
  print "the computer will then grade your guess"
  print "by showing a * for each"
  print "correct color which is not in the right"
  print "place - and a # for each"
  print "color which is in the right place"
  print
  print "for example you might enter 'rgbk' for"
  print "red, green, blue, black"
  print "and the computer might show your grade"
  print "by #** which would mean that you have"
  print "guessed three correct colors, of which"
  print "only one is in the correct position"
  input "press return to start the game: ": yn$
endproc initl
proc slct(ref m$) closed
  dim colors$ of 6, a$ of 1
  colors$:="rgbywk"
  m$:=""
  for i:=1 to 4 do //get four colors
    repeat
      n:=rnd(1,6) //get next color
      a$:=colors$(n:n)
      tf:=a$ in m$
    until (tf=false) //make sure color is not already used
    m$:=m$+a$
  endfor i
endproc slct
proc getguess(ref g$) closed
  dim colors$ of 6
  colors$:="rgbywk"
  g$:=""
  repeat //get players guess
    valid:=true
    input "enter next guess:  ": g$
    for i:=1 to 4 do
      valid:=(g$(i:i) in colors$) and valid
    endfor i
    if (len(g$)<>4) then valid:=false
  until valid //make sure it is valid
endproc getguess
proc report
  if (guess$=mind$) then correct:=true
  for i:=1 to 4 do //display players guess in picture form
    case guess$(i:i) of //check each char, show right colr
    when "r"
      red
    when "g"
      green
```

Student Mastermind

```
    when "b"
      blue
    when "y"
      yellow
    when "w"
      white
    when "k"
      black
    endcase
  endfor i
  print "@";
  for i:=1 to 4 do
    //check how many colors are in right place
    if guess$(i:i)=mind$(i:i) then
      blackdot
    endif
  endfor i
  for i:=1 to 4 do
    //check how many right colors not in right place
    ab$:=""
    for j:=1 to 4 do
      if (i<>j) then ab$:=ab$+mind$(j:j)
    endfor j
    if guess$(i:i) in ab$ then
      whitedot
    endif
  endfor i
  print
endproc report
proc red
  pencolor 2
endproc red
proc green
  pencolor 5
endproc green
proc blue
  pencolor 6
endproc blue
proc yellow
  pencolor 7
endproc yellow
proc white
  pencolor 1
endproc white
proc black
  pencolor 0
endproc black
proc blackdot
  pencolor 0
  print "#",
endproc blackdot
proc whitedot
  pencolor 1
  print "*",
endproc whitedot
proc goagain(ref yn$)
  if (correct) then //if player is right, congratulate her
    for i:=0 to 15 do
      border i
      for j:=1 to 100 do null
    endfor i
    border 0
    print
    print "you got it in ";nguess;" tries."
  else  //let her know if she didn't get it
    print "you blew it!!  the answer is ";mind$
  endif
  input "would you like another game? [y/n] ": yn$
endproc goagain
```

IBM PC COMAL and Printers

by James Landis

Here are some addresses that may help with UniComal IBM PC COMAL and printers.

In the System Package 2 instructions INP and OUT go to hardware addresses and they could damage your computer.

The addresses below should not hurt your computer (but use at your own risk).

To check printer status use:

INP#($379) for LPT1
INP#($279) for LPT2

The values have to be tested for each printer before use.

On an Epson FX100+ the values returned:

233	Printer Ready
87	Printer Not Ready
119	Printer Out Of Paper
147	Printer Turned Off

Don't forget to test your printer before using it in a program.

If you would like to reset any printer using an OUT instruction:

control register for LPT1 is $37a
control register for LPT2 is $27a

To use type:

```
OUT($37a,%11000) // initializes LPT1
OUT($37a,%11100) // printer ready
```

Control register bits:

bit#	settings/meanings
0	0=normal, 1=causes output of byte of data
1	0=normal, 1=automatic line after CR
2	0=initialize printer, 1=normal
3	0=deselect printer, 1=normal
4	0=interrupt disabled, 1=interrupts enabled

```
PROC reset'printer
 USE system
 OUT($37a,%11000)
 OUT($37a,%11100)
ENDPROC reset'printer
```

With the INP instructions you can check your printer before you send any data to see if power is off or out of paper. If it is not ready you can send the OUT instructions to try to initialize the printer. I hope this information is of use to someone.

Picture Package for C64 Cart

by Paul Keck

I've been working on a graphics save/load package for C64 COMAL 2.0 I call pics.

As COMALites, we all have the capability to draw hi-res or lo-res color graphics pictures, and save them to disk for later use. Unfortunately, those without a COMAL cartridge can not see them. This prompted me to write the pics package. With it, you can save and load not only COMAL graphics pictures, but also BASIC bitmaps and Koala format color pictures! The pics package makes use of the free package memory area from $8009 to $bfff, setting up a memory '*cache*' that is used as a storage area for pictures. You may also set aside a 10003 byte string as a second storage area.

Saving and loading of pictures is done through the cache. This leaves the graphics screen undisrupted until you decide to show the picture. It also means that you may display a message on the text screen such as "*Loading picture.. please wait.*" I think people will come up with some good applications of this type of thing before long.

One surprise I got with the preliminary versions of the package was GIANT economy sized disk files when I saved programs with the package linked. Why did the program file get so big? This caused a brief but intense session of head-scratching until I realized that the package thought the memory cache was part of itself (yes, and it saved the whole picture in memory along with the program).

I thought, "*Easy enough to fix,*" but then I thought, "*Wait a minute!*" I NEWed the program in memory, and loaded the giant file. Sure enough, after USE pics and cache'into'screen(TRUE), the picture that had been in the cache when I saved the program popped up on the screen. Oho! I added a procedure so that YOU, yes you, can tell the package whether to save the picture along with the program. Some people may decide to use the package just to take advantage of this feature.

The program "pics-demo" gives a tiny taste of what you might use this package for. Run it, and also read through the package keywords, so you'll know what it can do for you. I plan to expand the range of picture files that can be loaded and saved; I hope to include compact COMAL bitmaps, Doodle pics, and the "gg" and "jj" type compressed pictures that are often seen on QLink. Further suggestions will be amiably considered. I will also make the caching and uncaching routines faster- right now, I just wanted to make sure they would work. Any updates I make will be compatible with the existing package, so go ahead and get started using it! [hopefully the package & demo will be on TD#24]

load'koala(<filename$>)

Loads a Koala-format picture file into the memory cache. Does not affect the screen.

save'koala(<filename$>)

Saves the memory cache to disk as a Koala-format picture file. Will save it as a Koala pic regardless of whether the cache has a hi-res or lo-res screen in it, so make sure you only save lo-res screens this way.

load'screen(<filename$>)

Loads a COMAL picture file into the memory cache. This procedure loads both 36 block (hi-res) and 40 block (lo-res) COMAL pictures. It can be used as a direct replacement for the "loadscreen" procedure in the graphics package, but remember that it loads into the memory cache, NOT the graphics screen.

save'screen(<filename$>)

Saves the cache contents as a COMAL picture file. Saves either the 36 block type or 40 block type correctly. Can be used as a replacement for the "savescreen" procedure in the graphics package.

load'bitmap(<filename$>)

Loads a 32 block bitmap file into the cache. The procedure automatically clears out the color area of the cache before loading a bitmap (or a COMAL screen) so that any colors from a previously cached picture are removed. It replaces them with the current graphics pencolor (the pixel colors will show up as the pencolor). So, before loading in the bitmap, issue a pencolor(<color>).

save'bitmap(<filename$>)

Saves the cache's bitmap as a 32 block bitmap file. You can do this with any type of screen that is in the cache, enabling you to extract bitmaps from other sources.

cache'into'screen(int)

Places the contents of the cache into the graphics screen. The parameter refers to whether you want an automatic "fullscreen" command issued. The "fullscreen" is necessary to let COMAL know that the background and border colors have been changed. This means that if you are viewing the graphics screen and issue a "cache'into'screen(FALSE)", the old background will stay up until a "fullscreen" is given (looks awful). However, this also means that if you are viewing the text screen you may put up a hi-res/bitmap picture while the user is reading the text screen, and pop over to graphics when ready. Since switching over to the graphics screen requires a "fullscreen", everything's hunky dory.

screen'into'cache(int)
Puts the picture currently showing on the graphics screen into the cache. The parameter refers to whether an automatic "clearscreen" should be given after the screen is cached.

string'into'screen(int,<screen$>)
As "cache'into'screen", but places the contents of the string onto the graphics screen. The string must have a current length of 10003. This is taken care of by the <something>'into'string" procedures; the requirement is just to keep you from trying to put up a filename or some other string.

screen'into'string(int,<screen$>)
As "screen'into'cache", but puts the picture into a string instead. The string must be dimensioned to 10003 before attempting to place a picture in it.

string'into'cache(<screen$>)
Swaps the string contents into the cache.

cache'into'string(<screen$>)
Swaps the cache contents into the string. Can be used right after loading in a picture so that another may be loaded into the cache.

set'file'num(<file number>)
Sets the file number used for saving and loading pictures to and from the disk. Default is 254; I don't know of any reason you'd need to change it, but hey, why not. Maybe you have a lucky number.

decolor'cache
This is the routine called by the "load'bitmap" and "load'screen" procedures prior to loading in a picture. You could use it to strip the colors from a colored picture in the cache, leaving a bitmap.

decolor'text
Sets the entire text screen's color memory to the current text color. This can be used to "clean up" the text screen after viewing a lo-res color picture whose colors "bleed over" onto the text screen. This will not preserve the colors which may have been on the text screen before the picture messed them up; if you need several colors, I suggest using the "getscreen" and "setscreen" procedures in the system package.

link'pic(int)
The parameter tells whether to include the current cached picture when saving the program the package is linked to. Using this, it is now possible to save a picture along with a COMAL program! Just load the picture into the cache, issue a "link'pic(TRUE)", and save. This will add about 50 blocks to the program length (the package plus the cache). TRUE is the default condition when "pkg.pics" is linked, so if you don't want to save the picture in the cache, type "link'pic(FALSE)" before saving your program the first time. It should stay unlinked until you tell it otherwise. A note for interested package programmers: the procedure just resets the pointer in the library module signifying the end of the package to before or after the cache area.

version'pics$
Returns the version string, so you can see what version you have. This is the only one so far.

Reference
Graphics Editor System, Colin Thompson, *COMAL Today #11*, page 57
Pic Finder, Colin Thompson, *COMAL Today #12*, page 32

Power Driver Turtle Parameters

by R. Hughes

These functions give you some of the capabilities that the C64 cartridge users enjoy.

```
FUNC x'pos  // 0-320
  RETURN 256*PEEK(28742)+PEEK(28743)
ENDFUNC x'pos
//
FUNC y'pos  // 0-199
  RETURN 199-PEEK(28747)
ENDFUNC y'pos
//
FUNC heading CLOSED // 0-360
  a:=PEEK(28764); b:=PEEK(28765)
  CASE a OF
  WHEN 0
    RETURN 90
  WHEN 129
    RETURN 89
    when130
    RETURN 88-(b DIV 64)
  WHEN 131
    RETURN 82-(b DIV 32)
  WHEN 132
    RETURN 82-(b DIV 16)
  WHEN 133
    RETURN 74-(b DIV 8)
  WHEN 134
    RETURN 58-(b DIV 4)
  WHEN 135
    IF b<53 THEN
      RETURN 26-(b DIV 2)
    ELSE
      RETURN 386-(b DIV 2)
    ENDIF
  WHEN 136
    RETURN 322-b
  WHEN 137
    RETURN (194-(b*2))-(PEEK(28766) DIV 128)
  ENDCASE
ENDFUNC heading
```

COMAL into FORTRAN Translator

by Solomon Katz

[*Sol is sharing a program with us that will translate a COMAL program into Fortran source code. The program is not complete, but worth looking at by any Fortran programmers out there. Next we need a COMAL to C translator ... especially with Amiga COMAL right around the corner. I know several people who were working on such a translator; maybe one is completed by now.*]

The COMAL to Fortran 77 translator started its life as a quick and dirty automatic text editor to do some minimal level of substitution of specific words. Over the last two years I kept adding "*bells and whistles*" as I found time. I tried to produce "*plain vanilla*" F77 code. COMAL2F77 was written to work on LISTed COMAL files. It was written using COMAL 2.0 on the C64. The only package I used was strings, for the quicksort procedure. Therefore, the program should port easily to other versions of COMAL. The program is not finished, but I decided to share it before it becomes obsolete. User discretion is advised.

Some of the features of the program are:

* replaces COMAL keywords and operators with Fortran equivalents.
* adds parenthesis in IF expressions.
* splits one line WHILEs and FORs to multiple lines.
* splits multiple assignments on a line into separate lines.
* converts WHILE, LOOP and REPEAT structures to IF ... GOTO structures.
* converts COMAL line labels to F77 numeric labels.
* inserts INTEGER declares at the beginning of each subroutine / function.
* converts FOR structures to DO structures.
* converts a:+1 into a=a+1, and the same for a:-1.
* converts (6:) into (6:LENGTH()) for character data [substring specification].
* converts (:6) into (1:6) for character data [substring specification].
* removes $ from character variables and # from integer variables.
* changes IMPORT into COMMON/block/.
* removes PROC and FUNC names from COMMONs.
* splits off in-line comments and writes them to the next line.
* converts CASE structure to IF ... ELSE IF structure.
* inserts CALL before subroutine calls.
* ... and other similar functions.

In order to minimize hand editing of the F77 file, you should follow these guidelines when you program in COMAL. Most of these reflect significant differences between the languages.

* All PROCs and FUNCs should be CLOSED.
* If a variable is used as an integer (ie, counter), mark them with #.
* In substrings, don't use a$(6:) or b$(6).
* Wherever possible, variables should be passed by reference, using REF.
* Wherever possible, pass variables instead of using IMPORT.
* The TRAP structure doesn't translate well.
* All statements not in PROC / FUNCs should be at the top of the file.
* Don't use PEEKs and POKEs.
* Don't use KEY$.
* Put all DATA statements near the top of each PROC / FUNC.
* Don't use RESTORE.
* Don't mix TRUE / FALSE with 0 / 1.
* Lines should be numbered in steps of 2 or greater.
* When using IMPORT, keep PROCs and variables separate.
* Don't use AND THEN.
* DIM all strings.
* Don't use the British pound sign in your code. It is used as a place holder by COMAL2F77.
* All keywords should be UPPER case and all variables lower case.
* Be sure the program works in COMAL before translating it.

Even if you follow each of these guidelines, there are still some fixes you will have to do by hand.

* The F77 LEN function returns the length the string was dimensioned to. COMAL's LEN is converted to LENGTH. You will need to write a F77 LENGTH function.
* If you used abc$(6:) you will have to insert abc in abc(6:LENGTH( )).
* If you pass an array, you will have to dimension it in the subroutine or function. You will also need to remove the ()s and (,,)s from the argument list. I intentionally left them there as a reminder. The same goes for COMMON.
* If you use the VAL function, you will need to write a F77 function to do the same thing, probably by writing to and reading from a buffer.
* If you use COMAL packages, you will need to write your won F77 equivalents for those PROCs and FUNCs.
* If you used SPC$, you will have to declare a variable CHARACTER SPC * 100.

* Sometimes the logic will be odd: .NOT.(.NOT. . These will work but can be rewritten if you wish.
* EOF(3) will have to be changed to EOF3 or something of your own choosing.
* Some I/O commands will need to be fixed to match your version of F77.
* FORMAT statements will have to be inserted, especially in PRINT USING.
* COMMON blocks will have to be reconciled between subroutines / functions. This is the most difficult and time consuming part of the translator.
* DATA statements will have to be changed to F77 format and COMAL READs will have to be removed.
* Find and insert LOGICAL declares.
* Change <u>a$(6)</u> into <u>a$(6:6)</u>.
* Add or delete occasional parentheses.

I originally wrote the program using global variables, open PROCs and FUNCs and all real variables, and tested most of the features to prove they worked. Since then I concluded an example was needed. I rewrote the program following most of the guidelines. I did not re-test all the features! There is a good chance that some closed PROC/FUNCs will need to have former global variables, and/or PROCs or FUNCs imported, especially if it is a COMAL command that isn't in this program. I suggest that you try COMAL2F77 on a listed version of COMAL2F77 and see if the resulting F77 is good enough for your needs. LISTed COMAL2F77 is about 98 blocks. The resultant Fortran program is even larger. Make sure you have enough space available on your disk. It will take at least 15 minutes to process the 98 blocks, with processing time dependent on the destination of the output, ie, screen, disk, printer.

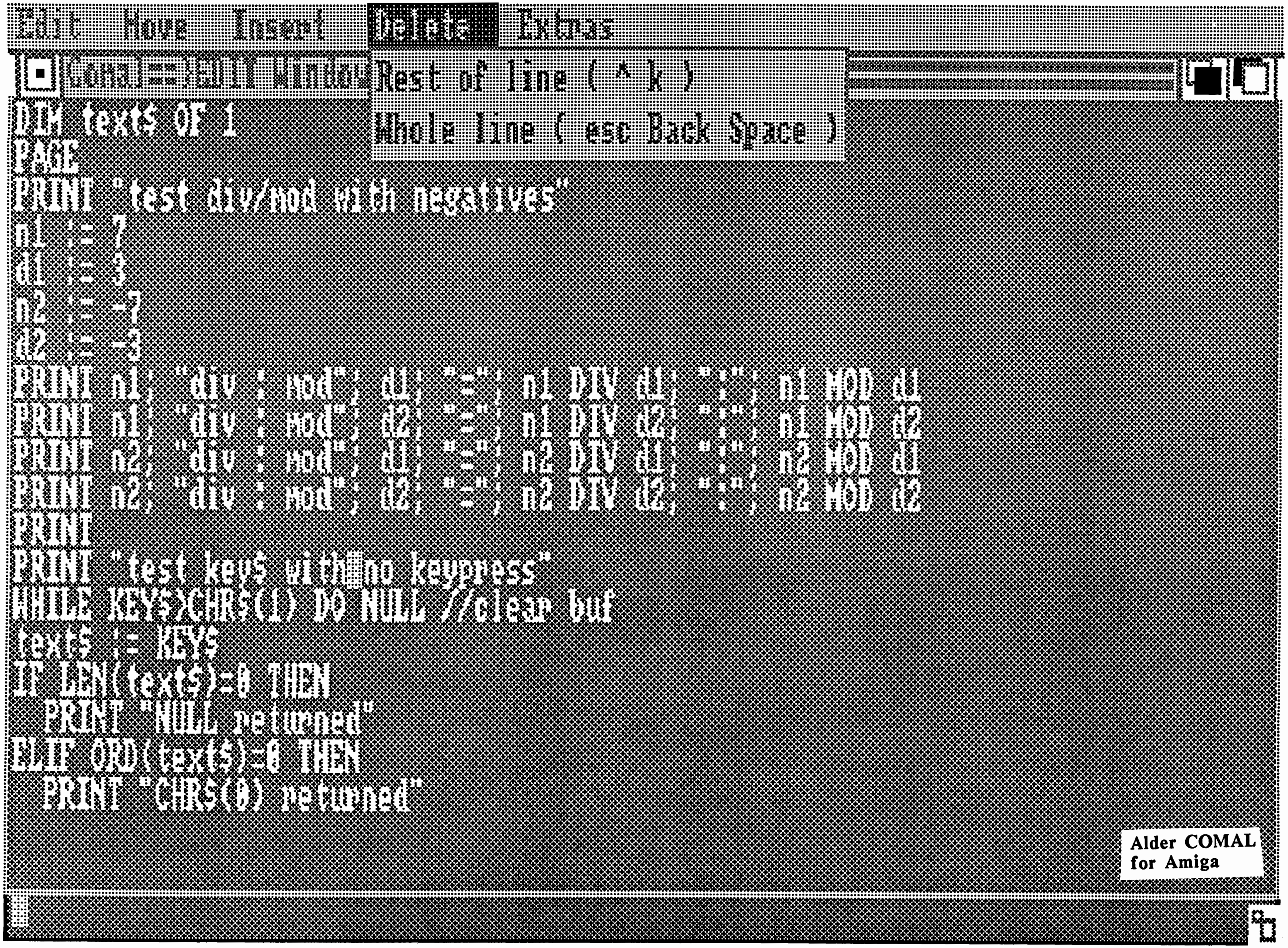
L I S T E R I N E for Power Driver

Original by Will Bow - Modified by R. Hughes

COMAL allows you to store your procedures and functions on disk by LISTing them disk. Later they can be merged into any future program. However, it is time consuming to find and individually list each routine to disk. LISTERINE to the rescue.

LISTERINE splits up a program into its individual procedures and functions, storing each to disk with a filename ending in <u>.f</u> for functions or <u>.p</u> for procedures. First you must LIST the program you wish to be broken up to disk (LIST "name"). Then run LISTERINE and give it the filename of your target program. This version renumbers each routine to start at line 9001 as well as allow you to test it by having it print to screen rather than to disk. (See also *COMAL Today 14* page 22)

```
PAGE
PRINT AT 3,9:"L I S T E R I N E"
set'up
read'listed'file
//
PROC set'up
  DIM txt$ OF 160, ending$ OF 2, tp$(5) OF 6
  DIM infile$ OF 18, outfile$ OF 18
  DIM e1$ OF 40, e2$ OF 40, name$ OF 80
  tp$(1):="proc"; tp$(2):="func"
  tp$(3):="PROC"; tp$(4):="FUNC"
  indev:=PEEK(28446) // cat dev#
  INPUT AT 9,1:"out'drive (8/9) or screen (3): ": outdev
  INPUT AT 13,1:"filename : ": infile$
ENDPROC set'up
//
PROC read'listed'file
  OPEN FILE 10,infile$,UNIT indev,READ
  WHILE NOT EOF(10) DO
    INPUT FILE 10: txt$
    IF "//" IN txt$ THEN //no rems in header
      txt$:=txt$(1:("//" IN txt$)-1)
    ENDIF
    type:=0
    FOR n:=1 TO 4 DO
      IF tp$(n)+" " IN txt$ THEN type:=n
    ENDFOR n
    IF type THEN
      IF NOT """" IN txt$ THEN
        make'name
        list'to'disk
      ENDIF
    ENDIF
  ENDWHILE
  CLOSE FILE 10
ENDPROC read'listed'file
//
PROC make'name
  tp$(5):=tp$(type); ending$:=".f"
  IF (type MOD 2) THEN ending$:=".p"
  p:=(". "+tp$(5)+" " IN txt$)+6
  name$:=txt$(p:LEN(txt$))
  par:="(" IN name$; sp:=" " IN name$
  IF par>0 AND par<sp THEN
    name$:=name$(1:("(" IN name$)-1)
  ELSE
    name$:=name$(1:(" " IN name$)-1)
  ENDIF
  x:=2-(type MOD 2) // lower-case
  e1$:="end"+tp$(x)+" "+name$; e2$:="END"+tp$(x+2)+
  " "+name$ //wrap line
  outfile$:="0:"+name$+ending$
  IF LEN(name$)>14 THEN outfile$:="0:"+name$(1:14)+ending$
  PRINT ">>>>>>>>>";outfile$
ENDPROC make'name
//
PROC list'to'disk
  OPEN FILE 20,outfile$,UNIT outdev,WRITE
  num:=9001; txt$(1:4):=STR$(num)
  PRINT FILE 20: txt$
  REPEAT
    num:+1
    INPUT FILE 10: txt$
    txt$(1:4):=STR$(num)
    PRINT FILE 20: txt$
    txt$:=txt$+" "
  UNTIL e1$ IN txt$ OR e2$ IN txt$
  PRINT FILE 20: STR$(num+1)+" //"
  CLOSE FILE 20
ENDPROC list'to'disk
```



```
DIM text$ OF 1
PAGE
PRINT "test div/mod with negatives"
n1 := 7
d1 := 3
n2 := -7
d2 := -3
PRINT n1; "div : mod"; d1; "="; n1 DIV d1; ":"; n1 MOD d1
PRINT n1; "div : mod"; d2; "="; n1 DIV d2; ":"; n1 MOD d2
PRINT n2; "div : mod"; d1; "="; n2 DIV d1; ":"; n2 MOD d1
PRINT n2; "div : mod"; d2; "="; n2 DIV d2; ":"; n2 MOD d2
PRINT
PRINT "test keys with no keypress"
WHILE KEY$>CHR$(1) DO NULL //clear buf
text$ := KEY$
IF LEN(text$)=0 THEN
  PRINT "NULL returned"
ELIF ORD(text$)=0 THEN
  PRINT "CHR$(0) returned"
```

Alder COMAL
for Amiga

Fractals and Recursion

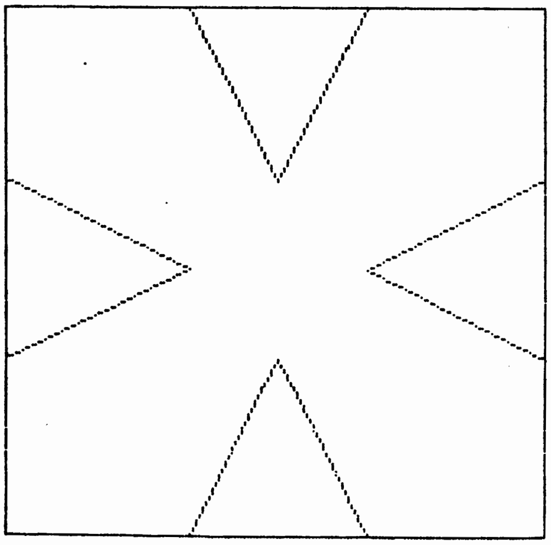
by Bill Inhelder

Fractal drawings which involve the repetition of a basic pattern are particulary appropriate for recursive computer programming techniques. In addition, the set of graphic instructions for each basic pattern provides an excellent opportunity to apply geometric and trigonometric principles. This is especially true when dealling with patterns containing triangular and trapezoidal designs. As an applied high school mathematics project, it combines mathematics concepts, computer graphics and recursive principles producing a final product which is artistically pleasing to the eye.

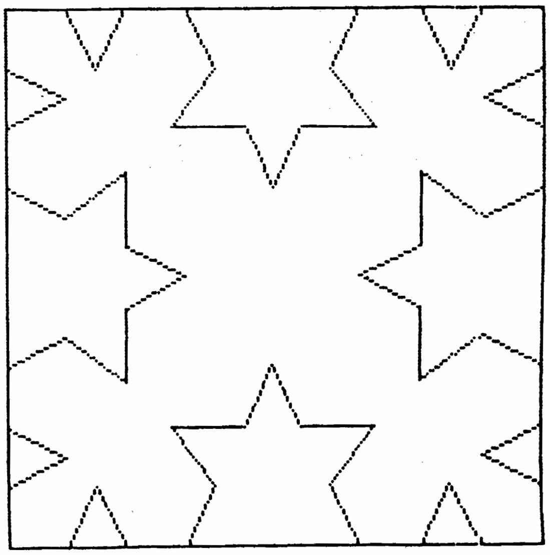
We start with a line segment of fixed length. Divide the segment in thirds and remove the middle segment. Construct a specific shape made up of connected straight line segments bridging the interval; for example, a triangular or rectangular shape. Next repeat the above procedure for each segment in the preceding pattern. Repeat this procedure once again. Finally repeat the entire set of procedures for each side of a large square. The resulting fractal pattern can be quite complex and eye appealing.

Once the initial pattern is established, it is repeated in ever diminishing size by repeated calls to an identical procedure with decreased parameters. Three calls of the basic pattern is usually the greatest definition the pixel density of the screen will allow.

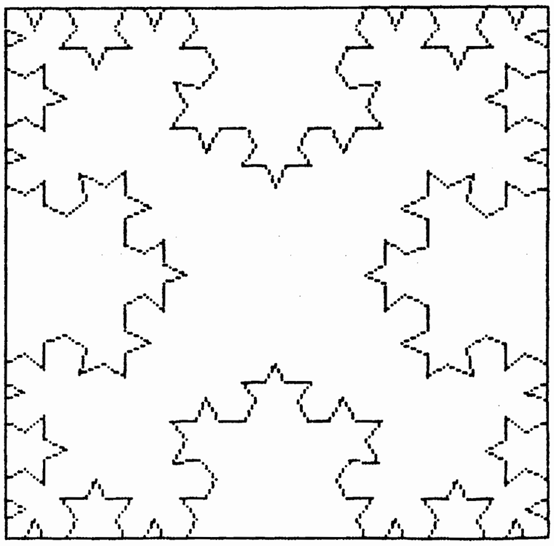
If a simple triangular shape is selected, the first call of PROC pattern produces the following simple shape inscribed in a square.



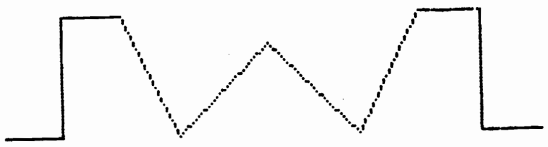
At the second level (call PROC pattern1), the initial shape is repeated at every segment resulting in a more involved fractal picture.



The third level (call PROC pattern2) produces the final fractal picture.

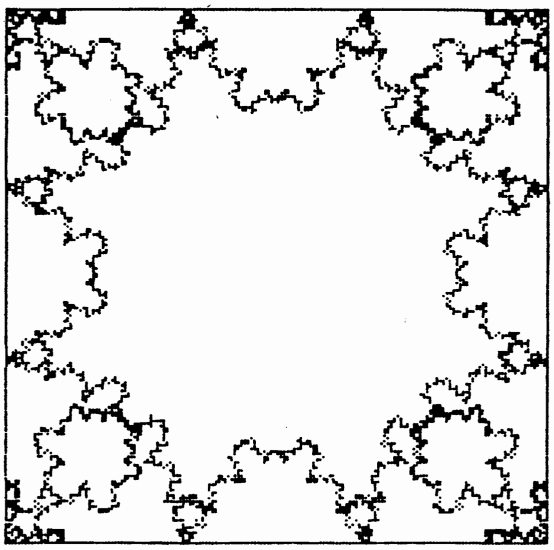


More involved basic patterns produce more complex fractal pictures. If the middle section of the basic pattern is lengthened so that several basic geopmetric types are included, the resulting fractal picture can be especially interesting. A basic pattern which includes a trapezoid, a triangle and another trapezoid (shown below) results in this intricate fractal picture ("*fracpattern7*").



(this is the pattern used ... see resulting fractal below)

Fractals and Recursion



Fracpattern1 begins with a basic rectangular pattern. In the listing below, notice that PROC pattern2(d) is activated which then activates PROC pattern1(d) which in turn activates PROC pattern(d) which executes the basic graphic pattern. By changing the procedure call to either pattern(dist) or pattern1(dist), the simpler fractal patterns are produced.

```
// fracpattern1 by bill inhelder july, 1988
USE graphics
PAGE
PRINT "Enter a size between 120 and 199. 199 "
PRINT "covers the entire screen."
INPUT dist
window(0,230,0,199)
graphicscreen(0)
background(1)
pencolor(0)
penup
setxy(0,0)
pendown
FOR i:=1 TO 4 DO
  forward(dist); right(90)
ENDFOR i
FOR i:=1 TO 4 DO
  pattern2(dist); right(90)
ENDFOR i
pencolor(8)
fill(115,100)
WHILE KEY$="" DO NULL
//
PROC pattern(d)
  forward(d/3); right(90)
  forward(d/6); left(90)
  forward(d/3); left(90)
  forward(d/6); right(90)
  forward(d/3)
ENDPROC pattern
//
PROC pattern1(d)
  pattern(d/3); right(90)
  pattern(d/6); left(90)
  pattern(d/3); left(90)
  pattern(d/6); right(90)
  pattern(d/3)
ENDPROC pattern1
//
PROC pattern2(d)
  pattern1(d/3); right(90)
  pattern1(d/6); left(90)
  pattern1(d/3); left(90)
  pattern1(d/6); right(90)
  pattern1(d/3)
ENDPROC pattern2
```

While the single recursive procedure shortens the program, I feel that it results in a program that is more difficult to follow and understand.

Six other programs are provided on disk: *fracpattern2* through *fracpattern7*. Several use a single recursive procedure rather than three procedures. Some basic patterns are drawn on only one side of the original line while others are drawn on both sides. In the latter case the beginning point of the drawing must be located in the interior of the large square (see *fracpattern3, 5, 6*) rather than on the lower left hand corner. By studying some of the different procedures and using the forward, right and left graphic functions, you can design your own fractal pictures.

Reference

3D Fractals, Kevin Quiggle, *COMAL Today #12*, page 26. (sample on the cover)
Diffusion Limited Aggregation, Jim Frogge, *COMAL Today #23*, page 46.
Fractal Geometry, Ted Groszkiewicz, *COMAL Today #18*, page 53. (sample on the cover)
Mandelbrot Etc, Robert Ross, *COMAL Today #20*, page 36.
Mandelbrot Revisited, Ray Carter, *COMAL Today #19*, page 11.
Recursive Designs, D. Bruce Powell, *COMAL Today #15*, page 35.
Turtle Graphics and Mathematical Induction, Frederick Klotz, *Mathematics Teacher* 8 (1987): 636-39.

Change Drive - Power Driver

This procedure allows you to change a device number via software control.

```
// change diskdrive device#
// from old' to new' ie: drive(8,9)
PROC drive(old',new') CLOSED
  OPEN FILE 15,"",UNIT old',15,READ
  DIM s$ OF 10
  s$="m-w"+chr$(119)+chr$(0)+chr$(2)+chr$(new'+32)+
  chr$(new'+64) //wrap line
  PRINT FILE 15: s$,
  CLOSE FILE 15
ENDPROC drive
```

Disk Drive Direct Access

by R. Hughes

This Power Driver program reads and writes data to disk blocks by direct access to tracks and sectors.

The program reads disk data by opening the disk command channel (15) and a data channel (5) to a diskdrive buffer. Data is read from the specified disk track/sector to the 256 byte drive buffer using the DOS 'U1:' command then transferred from the buffer to the computer through the data channel. The start byte in the buffer is specified by the DOS 'B-P:' (block pointer) command.

The program writes disk data by opening the disk command channel (15) and a data channel (5) to a diskdrive buffer. Data is written from the computer to the drive buffer through the data channel. The start byte in the buffer is specified by the DOS 'B-P:' (block pointer) command. The complete buffer is then written to a specific disk track/sector using the DOS 'U2:' command.

The program provides the following procedures:-

1) READ'DIR
This is automatically called when the program is run and provides a directory listing showing the start track & sector of each program in the directory and in the case of REL files, the record length is also given. The procedure shows each directory block used and uses another procedure called READ'ENTRY(n) to read the data of each of the 8 programs listed in each directory block. The listing is paused while <SHIFT> is pressed.

2) READ'(trk,sct)
This procedure reads the specified track & sector block into a 256 byte variable called TEXT$. The first 2 bytes give the next track and sector linkages.

3) WRITE'(trk,sct)
This procedure is the reverse of READ' and writes the variable TEXT$ to the specified track/sector.

4) READ'ALL(FLAG)
This procedure prompts for a start track/sector of a file then traces the linkages through to the end. The procedure has 2 modes of output depending on the value of FLAG.

* If FLAG=0 the procedure lists a buffer count and the current and next track & sector values.
* If FLAG=1 the procedure prints the contents of TEXT$(3:256) of each block (the first 2 bytes are the next-track and next-sector).

5) ASC(n)
This procedure lists the ascii values of the first 'n' bytes of TEXT$.

The adventurous can now proceed with reading and modifying block data using the READ' and WRITE' procedures and modifying the original data by manipulating the contents of TEXT$.

The disk directory can also be modified (and very easily too) by the following procedure:-

a) use READ'DIR to start a directory listing then pausing the list using <SHIFT> at the end of the appropriate block.

b) use RUN/STOP to stop the procedure then use READ'ENTRY(n) to read the directory parameters of the nth program in the block.

c) The variable 'x' gives the byte in TEXT$ where the directory entry starts. The ASCII values of the bytes are as follows:-

x gives the program type 128 + 1(SEQ), 2(PRG), 3(USR), 4(REL). Use 0 to DEL a file or add 64 to protect a file.

x+1, x+2 gives the start track and sector of the program.

x+3 to x+18 this is the filename padded with shifted spaces chr$(160).

x+21 gives the record size (for REL files only)

x+27, x+28 give the file size in hi-lo format (ie the file size is given by 256*ascii value of TEXT$(x+27) + ascii value of TEXT$(x+28).

Writing this program taught me a lot about disk program and directory structures and the program itself is so easy to understand and modify that I am sure you will be able to modify it to any special purpose of your own.

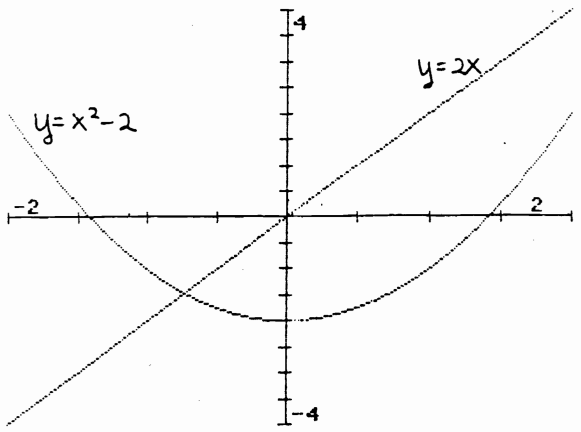
Footnote: Did you know that the last block of a file has a next-track of zero and a next-sector value between 1 and 253 representing the number of valid bytes in the last block. This is because fractional blocks cannot be saved to disk and something has to indicate the border between the true data and the garbage.

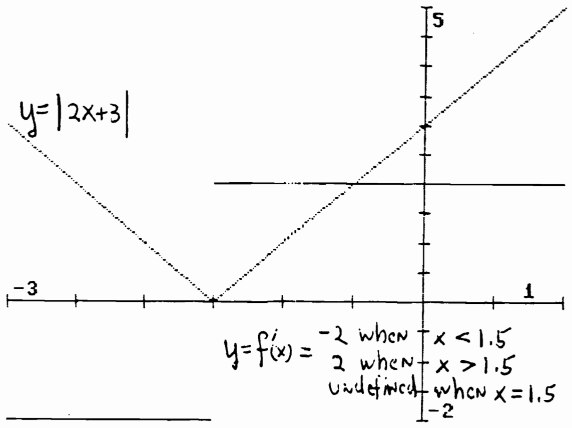
```
DIM text$ OF 256, ask$ OF 1, temp$ OF 1, f'type$ OF 15
f'type$:="***SEQPRGUSRREL"
read'dir
//
PROC read'all(flag)
  INPUT "track,sector : ": nt,ns
  count:=0
  WHILE nt>0 AND nt<36 DO
    read'(nt,ns); count:+1
    IF flag THEN
      IF nt THEN
        PRINT text$(3:256),
```

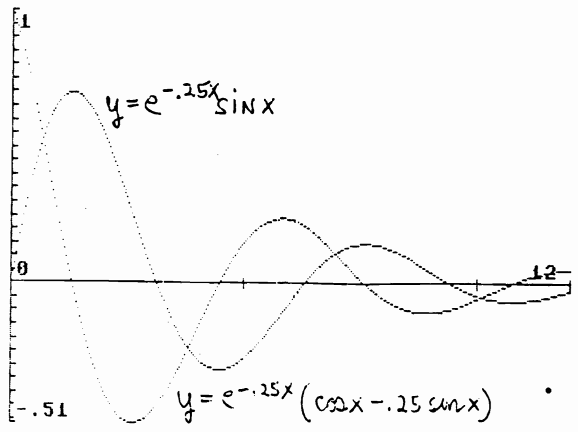
Disk Drive Direct Access

```
   ELSE
    PRINT text$(3:ns)
   ENDIF
  ELSE
   PRINT USING " ###": count,trk,sct,nt,ns
  ENDIF
 ENDWHILE
ENDPROC read'all
//
PROC read'(track,sector)
 trk:=track; sct:=sector
 open'drive
 PRINT FILE 15:"u1: 5 0 "+STR$(track)+" "+
 STR$(sector)+" ", //wrap line
 PRINT FILE 15: "b-p: 5 0 ",
 text$:=GET$(5,256)
 nt:=ORD(text$(1)); ns:=ORD(text$(2))
 CLOSE
 PRINT CHR$(0),
ENDPROC read'
//
PROC write'(track,sector)
 open'drive
 PRINT FILE 15: "b-p 5 0 ",
 PRINT FILE 5: text$,
 PRINT FILE 15: "u2: 5 0 "+STR$(track)+" "+
 STR$(sector)+" ", //wrap line
 CLOSE
ENDPROC write'
//
PROC open'drive
 OPEN FILE 15,"i0",UNIT 8,15,READ
 OPEN FILE 5,"#",UNIT 8,5,READ
ENDPROC open'drive
//
PROC read'dir
 b'free:=664; read'(18,0)
 PRINT "",text$(145:165)," size trk sct rec"
 WHILE nt>0 AND nt<36 DO
  read'(nt,ns)
  PRINT "";trk;sct
  FOR n:=1 TO 8 DO read'entry(n)
  WHILE PEEK(653) DO NULL //shift
 ENDWHILE
 PRINT USING "### blocks free      ": b'free
ENDPROC read'dir
//
PROC read'entry(n)
 IF n<1 OR n>8 THEN RETURN
 x:=(n-1)*32+3; y:=ORD(text$(x))
 type:=y BITAND 7; prot:=y BITAND 64
 size':=ORD(text$(x+28))+256*ORD(text$(x+27))
 IF size'>0 OR type>0 THEN
  trk':=ORD(text$(x+1)); sct':=ORD(text$(x+2))
  IF type>0 THEN b'free:-size'
  PRINT " "+text$(x+3:x+18),TAB(19),f'type$(3*
  type+1:3*type+3), //wrap line
  IF prot THEN PRINT "<",
  PRINT TAB(23),
  PRINT USING " ###": size',trk',sct';
  IF type=4 THEN PRINT USING "###": ORD(text$(x+21)),
  PRINT // ^^^ rec size ^^^
 ENDIF
ENDPROC read'entry
//
PROC asc(n)
 FOR i:=1 TO n DO PRINT USING "### ": ORD(text$(i));
 PRINT
ENDPROC asc
```

Graphing a Function & Derivative





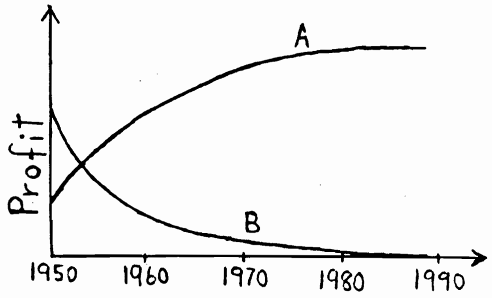


Graphing a Function and its Derivative

by Bill Inhelder

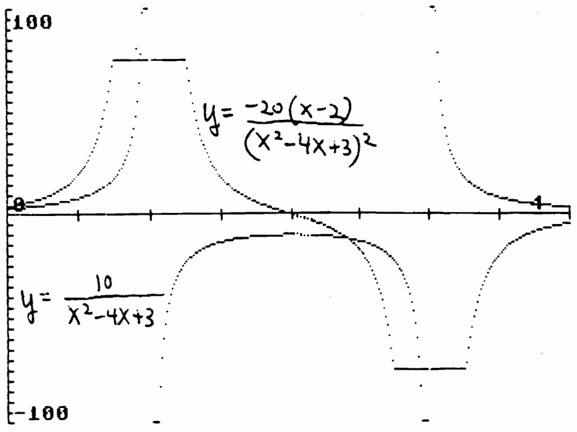
[*See sample output screen dumps on previous page*] In an introductory calculus class, the derivative of a function at a point is usually presented in terms of the slope of the chord joining the given point of the function with another point of the function which is arbitrarily close. Thus we speak of the slope of the graph of a function at a specific point. The collection of all such slope values over the entire function constitutes the slope function which is, by definition, the derivative of the function. In this way the student proceeds from the familiar concept of slope in algebra to an understanding of limits, the slope function and eventually the rules for determining the derivative of functions.

A computer program which produces graphs of both the function and the slope function serves to reinforce an understanding of the basic definition of the derivative before learning the rules of differentiation. An example from economics illustrates this last point. The graph below shows the profits of a company over a period of years. Clearly the company is profitable as shown by curve A; however, notice that while the company's profits continue to increase, the rate of increase (the slope function - curve B) is decreasing. Do you invest money in such a company?



The program fandsgrapher produces a table of 40 sets of values for the function and the slope fuction. For each function a total of 320 points are calculated; however, only every eighth point is displayed in the table. Both graphs are displayed on the same set of axes with automatic scaling which is determined by the min & max values of the 640 points calculated over the interval specified. The function is graphed first, followed by the slope function. Tic marks are automatically set at intervals of 50, 5, .5, .05 and .005 as appropriate.

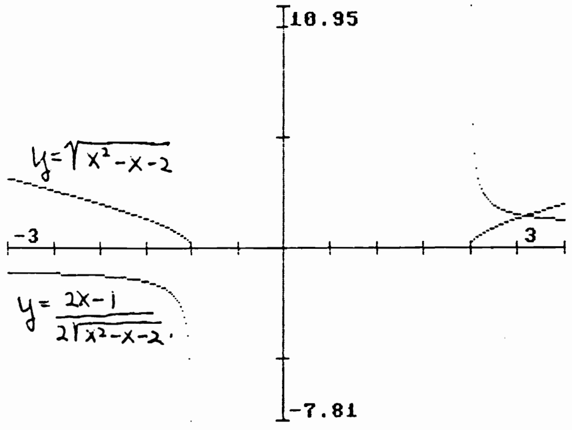
The min / max y values are represented by the extreme tic marks on the y-axis even though the numerical values for these points are not directly opposite those tic marks. Since the screen is used to its maximum size, there is no additional room above and below the y-axis for the maximum and minimum values. The same condition holds for the min / max on the x-axis.



Program requirements and options:

a. enter equation of 67 characters or less
b. type lower/upper bounds separated by a comma
c. after table/graphs are displayed press c to continue
d. press p for printer output of screen; otherwise press RETURN
e. press b for new bounds of original function or
  n for a new function or
  q to quit the program.

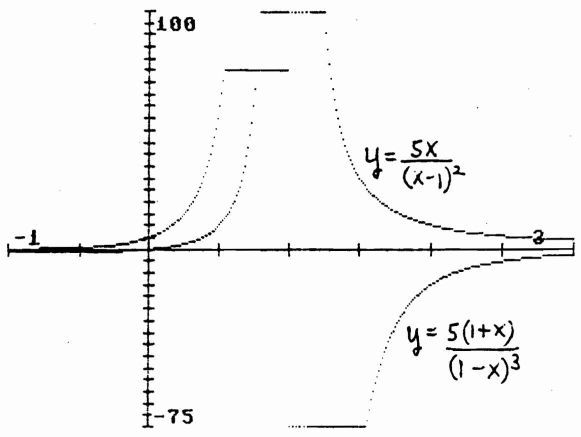
Be sure to enter the equation correct syntactically. Because of the method used to evaluate the equation in a running program, should a syntactic error occur, the program will halt and the program MUST be reloaded (the last 6 lines will have been deleted). See Notes to Programmers for the reasons pkg.meta was not used to evaluate the equation.



For graphic purposes, whenever a calculation for a function results in values greater than 100 or less than -100 those values are set to 100 and -100 respectively. The graphic maximum and minimum for the slope function are 75 and -75. Such limitations are especially necessary in cases of functions with essential discontinuities since the interesting portions of the graph tend to lose all significance. When the limits are exceeded, the condition will be apparent in the table of values and the graph will display horizontal line segments.

Graphing a Function and its Derivative

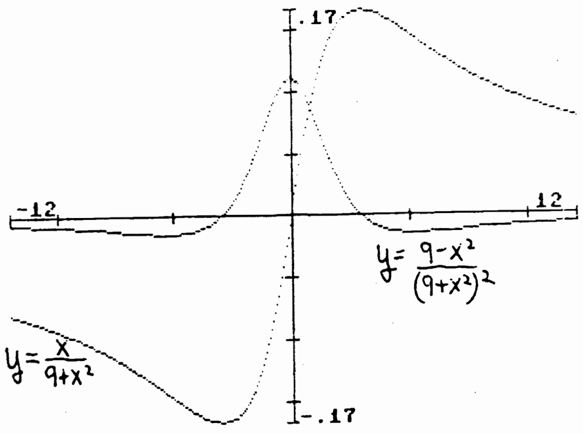
For functions whose values exceed the maximum because of the selection of inappropriate bounds, select the option to enter different bounds.



Error trapping has been provided for cases involving division by zero and invalid function arguments. Thus for a function defined by y=1/((x-1)*(x-3)) when x=1 or 3, the calculation is skipped in the table and the point is not plotted. For a function defined by y=sqr(x*x-x-2), which is not defined over the open interval -1 to 2, such values are omitted from the table and graph.

With error trapping capability and the ability to change minimum and maximum values in lines 2520 to 2550, the program should be able to handle all types of functions.

The algorithm for determining the values of the slope function is the definition of the derivative with delta x=.0001.

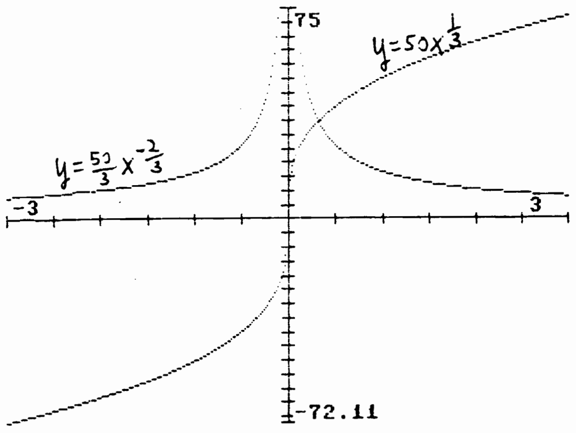


Lines 40 and 2210 contain printer specific graphic screen dumps. If you have a Commodore MPS-801 printer, delete line 40 and substitute PRINTSCREEN("lp:",80) for line 2210. If you own a Gemini-10x, Okidata 93A or Epson RX-80 you may link "pkg.finchutilit" and substitute USE finchutilities for line 40 and one of the following for line 2210:

a. g10xdump(1) for Gemini
b. okidump(1) for Okidata
c. rx80dump(1) for Epson.

This package has the added advantage of being saved with the program. You can select your favorite graphic screen dump package or procedure.

Special problems arise with certain functions because of the way BASIC handles the exponentiation operation (the up arrow). In programming mode this operation requires that the base be greater than 0. This arises because the solution to y=x^b uses the log function which requires positive values of x. As a result, functions like y=50*x^(1/3) over -3 to 3 must be rewritten y=sgn(x)*50*abs(x)^(1/3), while y=20*x^(2/3) is rewritten as y=20*(x*x)^(1/3). To add to the confusion, the statement PRINT (-3.27)^(1/3) when executed in <u>immediate</u> mode produces the correct answer without running an invalid argument error! So much for the idiosyncrasies of BASIC.



<u>Because of the nature of the procedures used to evaluate the equations in a running program, no statement may be placed before the first statement in the program.</u> These procedures are described in *COMAL Today #7*, pg. 32 by Jesse Knight.

To acquaint the reader with the intricacies of the program, it is recommended that the following examples be tried:

1. y=x*x-2 from -2 to 2
2. y=x/(9+x*x) from -12 to 12
3. y=abs(2*x+3) from -3 to 1. Note that the derivative of a continuous function need not be continuous.
4. y=sgn(x)*50*abs(x)^(1/3) from -3 to 3,then from -1 to .25. Same condition as #3.
5. y=5*x/(x-1)^2 from -1 to 3. Note discontinuties and maxima indicated by horizontal line segments.
6. y=exp(-.25*x)*sin(x) from 0 to 12
7. y=sqr(x*x-x-2) from -3 to 3. Note functions are not defined over open interval -1 to 2.

8. y=10/(x*x-4*x+3) from 0 to 4. Note discontinuities at 1 and 3. The slope function has a larger number of points whose slopes exceed 75. Note also that the linear segments have points missing at 1 and 3 since the slope function is also undefined there.

There are times when it may be difficult to decide on the input bounds for a particular function. The table of values can be very helpful in deciding what part of the domain of the function produces interesting results. Consider the function defined by y=2*x*x-113*x+1541. Inputted bounds of -5 to 5 result in function values which exceed the maximum 100. In addition, since the slope is -75 over the domain -5 to 5, this implies that the function is decreasing and the interesting portion of the graph lies to the right of 5. Using values from 10 to 50, confirms this fact and identifies the roots of the quadratic equation (23 and 33.5).

While this program produces graphs of the function and its derivative and illustrates the relationship in terms of slope, it does not provide the user with the equation of the derivative. For that purpose consult the article and program Symbolic Differentiation in *COMAL Today #9* by Tom Kuiper or apply the rules of differentiation.

Notes To Programmers

Generally I prefer to use pkg.meta over the method used in the program. However, two serious shortcomings developed when meta was used. If the program was run with successive options of change of bounds and/or change of equations, it was not possible to exit the program without locking up the computer. Either the run/stop key or the regular exit would lock up the computer. Normal exit did however occur when the program was run only the first time through. By chance I discovered that I could exit without lock-up if the last statement caused an error, such as, PRINT 1/0.

The second problem was more perplexing even when I was able to pinpoint the specific problem. In order to trap function errors when the user specifies a domain for which the function is wholly or partially not defined, strange things happen. The eval procedure is trapped with appropriate handling. The program executes properly for the first 12 consecutive values which trap and handle the invalid function arguments; however, on the 13th the run time error occurs: statement too long or complicated. Furthermore the error is very difficult to break out of.

I was able to use pkg.meta successfully in the program param'graph without the first problem occurring. Even after deleting the printer options and structuring fandsgrapher similar to param'graph, the problem persisted. The reference material for pkg.meta states that adding variable names to the name table or increasing program size will cause lock-up. This was not done in the program. Similar techniques were used in param'graph without ill effect. Any suggestions?

Further Reference

1525 ML Screen Dump from COMAL, Mike Lawrence, *COMAL Today #9*, page 78.
Batch Files From Memory, Jesse Knight, *COMAL Today #7*, page 32.
Code Doctor, Richard Bain, *COMAL Today #13*, page 56.
Differential Equations, Lowel Zabel, *COMAL Today #7*, page 73.
Differentiation, Tom Kuiper, *COMAL Today #11*, page 66.
Epson Package, Green, Rose, Wright, Grainger, *COMAL Today #14*, page 51.
Epson Screendump Package, Dennis Kurnot, *COMAL Today #10*, page 66.
EXEQ - A New Package, Ian MacPhedran, *COMAL Today #10*, page 62.
Expression Evaluator 0.14, Lewis Brown, *COMAL Today #23*, page 28.
Gemini 10X Graphics Dump, *COMAL Today #7*, page 66.
Graphing Parametric Equations, Bill Inhelder, *COMAL Today #23*, page 32.
How To Dump A Graphics Screen To Your Commodore Printer, David Stidolph, *COMAL Today #4*, page 30.
How To Use The META Package, Glen Colbert, *COMAL Today #10*, page 65.
HP LaserJet Screen Dump, David Stidolph, *COMAL Today #10*, page 15.
Multi-function Graphics, Lowell Toms, *COMAL Today #14*, page 24.
Okidata Graphics Dump, Terry Ricketts, *COMAL Today #11*, page 56. Okidata Package, Terry Ricketts, *COMAL Today #12*, page 20.
Packages Library Vol 1, Finchutilities, pg.14.
Packages Library Vol 2, Meta, pg.30.
Perform the Impossible, *COMAL Today #10*, pg.31.
Professional Graphs, Tom Kuiper, *COMAL Today #5*, page 40.
Programming Batch Files, Ian MacPhedran, *COMAL Today #8*, page 50.
Screendump for Oki, Epson, Star, Randy Finch, *COMAL Today #10*, page 72.
Stats For Teachers, Gerard Frey, *COMAL Today #22*, page 66
Symbolic Differentiation, Tom Kuiper, *COMAL Today #9*, pg.53.
Trig Art, Gerald Hobart, *COMAL Today 14*, pg 63

File Master

by Bobby Wallen (BobbyW6 on QLink)

This is a short program that provides the basis of a file maintenance system. It can display a directory, rename or scratch files, or read a file to the screen or printer.

```
DIM reply$ OF 1, name$ OF 16, text$ OF 255
DIM char$ OF 1, nname$ OF 16, cmnd$ OF 30
PAGE
PENCOLOR 3
BORDER 0 //border black
BACKGROUND 0 //background black
PRINT AT 10,15:"FILE"
PRINT AT 12,14:"MASTER"
PRINT AT 18,16:"by"
PRINT AT 20,11:"Bobby Wallen"
REPEAT
  wait
  choice
UNTIL reply$="Q" OR reply$="q"
//
PROC choice
  PAGE
  PRINT AT 10,5:"[R]ead Directory"
  PRINT AT 11,5:"[V]iew File"
  PRINT AT 12,5:"[P]rint file"
  PRINT AT 13,5:"[Q]uit"
  PRINT AT 14,5:"[S]cratch File"
  PRINT AT 15,5:"Re[N]ame file"
  PRINT AT 20,5:"Enter your choice [R,V,P,S,N,Q]"
  REPEAT
    reply$:=KEY$
  UNTIL reply$ IN "rvpqsn"
  CASE reply$ OF
  WHEN "r"
    direc
  WHEN "v"
    view
  WHEN "p"
    prnout
  WHEN "q"
    PAGE
    PRINT "Goodbye."
  WHEN "s"
    scratch
  WHEN "n"
    rename
  OTHERWISE
    NULL //not valid
  ENDCASE
ENDPROC choice
//
PROC wait
  PRINT
  PRINT "press  c  to continue"
  REPEAT
  UNTIL KEY$="c"
ENDPROC wait
//
PROC direc
  PAGE
  DIR
ENDPROC direc
//
PROC view
  PAGE
  INPUT "filename: ": name$
  OPEN FILE 2,name$,READ
  WHILE NOT EOF(2) DO
    text$:=""
    FOR i:=0 TO 254 DO
      char$:=GET$(2,1)
      text$:=text$+char$
    ENDFOR i
    PRINT text$,
  ENDWHILE
  CLOSE FILE 2
ENDPROC view
//
PROC prnout
  PAGE
  INPUT "filename: ": name$
  OPEN FILE 2,name$,READ
  OPEN FILE 3,"",UNIT 4,7,WRITE
  WHILE NOT EOF(2) DO
    text$:=""
    FOR i:=0 TO 254 DO
      char$:=GET$(2,1)
      text$:=text$+char$
    ENDFOR i
    PRINT FILE 3: text$
  ENDWHILE
  CLOSE FILE 2
  CLOSE FILE 3
ENDPROC prnout
//
PROC scratch
  PAGE
  INPUT "Enter name of file:": name$
  cmnd$:="s0:"+name$
  PRINT "",name$
  INPUT "ARE YOU SURE (Y/N):": reply$
  IF reply$="y" THEN
    PASS cmnd$
  ELSE
    choice
  ENDIF
  PRINT STATUS$
ENDPROC scratch
//
PROC rename
  PAGE
  INPUT "Enter present name of file:": name$
  INPUT AT 4,0:"Enter new name of file:": nname$
  cmnd$:="r0:"+nname$+"="+name$
  INPUT AT 20,0:"ARE YOU SURE (Y/N):": reply$
  IF reply$="y" THEN
    PASS cmnd$
  ELSE
    choice
  ENDIF
  PRINT STATUS$
ENDPROC rename
```

Set Printer Device - Power Driver

This procedure allows you to change the default printer device and secondary address.

```
// change printer dev# & 2nd addr
PROC set'printer(dev,sa)
  POKE 27507,dev
  POKE 27509,sa
ENDPROC set'printer
```

CHAOS

by Lewis Brown

This CHAOS program was written after watching the PBS NOVA TV show "*The Strange New Science of Chaos*". The screen dump on the next page shows it running under AmigaCOMAL with the windows adjusted so you can see the listing and results at the same time. It demonstrates that seemingly random points can create a pattern (triangles). If you missed the TV show, here is how it works: Pick 3 points on a plane (P1,P2,P3). Then, pick a 4th point (P4) anywhere on the plane. To determine the next point (Pm), pick a number from 1 to 6 at random. You do this by throwing a die (on the computer you get a random number). If the value is 1-2, draw a line to P1, and locate the midpoint, Pm. If the value is 3-4, draw a line to P2, and locate the midpoint, Pm. If the value is 5-6, draw a line to P3, and locate the midpoint, Pm. Now, repeat the process to locate a new random point, Pm. That is: (1) Get a random number from 1-6, (2) draw a new line to (P1,P2,or P3), and, (3) Determine a new midpoint Pm, then do it again. [*Curious? Try adapting the program for a rectangle instead of triangle base ... I tried it and got christmas trees in the pattern*]

```
dim'var
pick'point
x'y'axis
init'points
plot'point
WHILE KEY$=CHR$(0) DO
  random'point
  plot'random(m)
ENDWHILE
//
PROC x'y'axis
  SETGRAPHIC (0) //clear hi res screen
  HIDETURTLE
  FULLSCREEN
  PLOTTEXT 0,0,"0,0"
  PLOTTEXT 275,0,"x-axis"
  PLOTTEXT 0,195,"y-axis"
  MOVETO 0,0
  DRAWTO 319,0 //x-axis
  MOVETO 0,0
  DRAWTO 0,199 //y-axis
ENDPROC x'y'axis
//
PROC dim'var
  x1:=10; y1:=10; n:=2
  x2:=100; y2:=190
  x3:=310; y3:=100
ENDPROC dim'var
//
PROC init'points
  PLOTTEXT x1,y1,"+"
  PLOTTEXT x2,y2,"+"
  PLOTTEXT x3,y3,"+"
ENDPROC init'points
//
PROC pick'point
  PAGE
  PRINT "Pick the coordinates of the first point, Pm: "
  PRINT "0<x<319  and  0<y<199"
  INPUT "X= ": x
  INPUT "Y= ": y
  IF (x<0 OR x>319 OR y<0 OR y>199) THEN pick'point
ENDPROC pick'point
//
PROC plot'point
  PLOTTEXT x,y,"*"
ENDPROC plot'point
//
PROC random'point
  m:=RND(1,6)
ENDPROC random'point
//
PROC plot'random(m)
  CASE m OF
  WHEN 1,2
    PLOT (x1+x)/n,(y1+y)/n
    x:=(x1+x)/n; y:=(y1+y)/n
  WHEN 3,4
    PLOT (x2+x)/n,(y2+y)/n
    x:=(x2+x)/n; y:=(y2+y)/n
  WHEN 5,6
    PLOT (x3+x)/n,(y3+y)/n
    x:=(x3+x)/n; y:=(y3+y)/n
  ENDCASE
ENDPROC plot'random
```

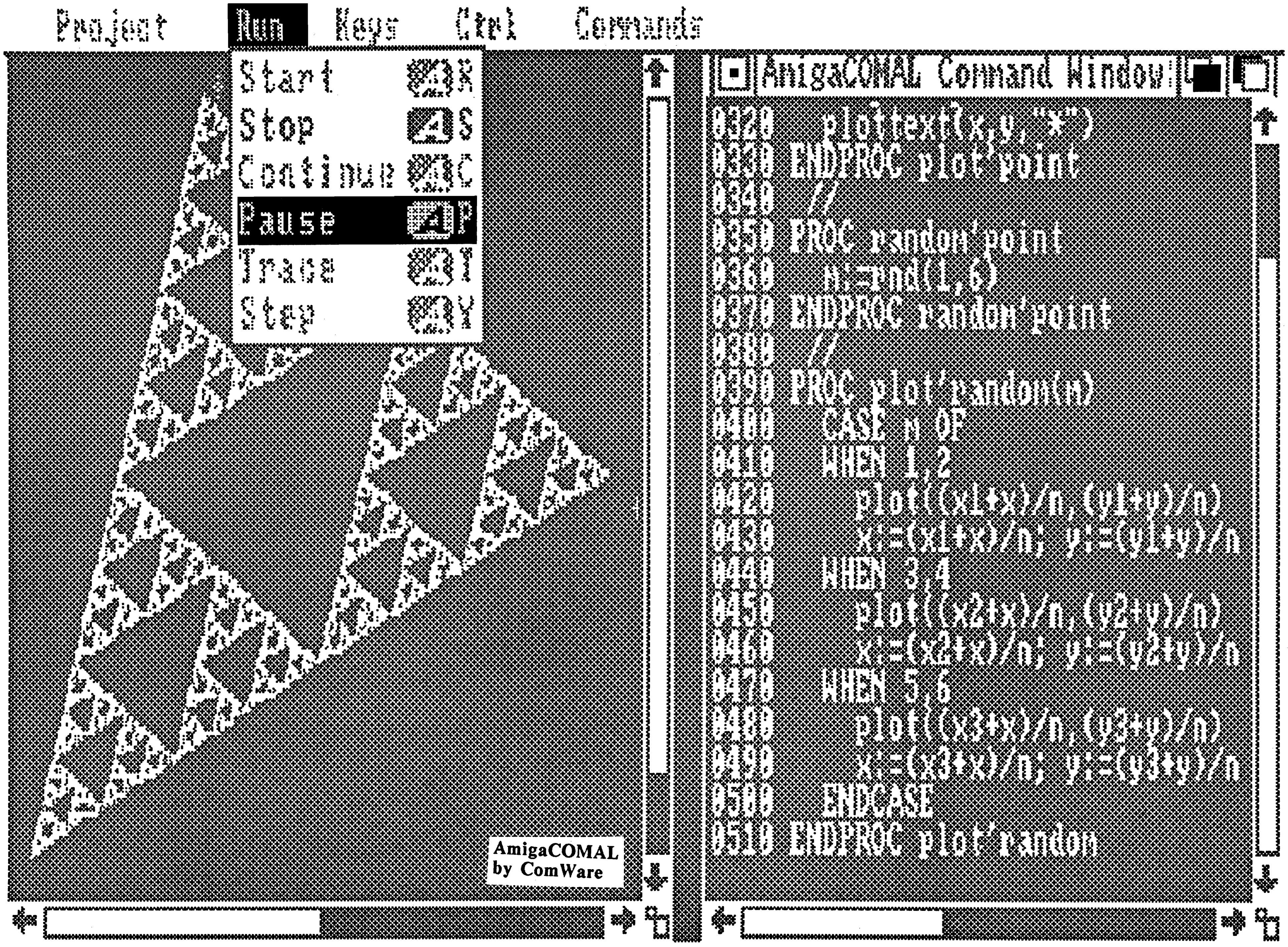
C64 Cartridge Puzzle Program

by Robert Ross

This puzzle program is a small version derived from a concept with a design maximum of 23 x 23 x 23. This version doesn't have any linked packages and has room to store everything in dimensioned string variables but it still isn't directly portable to other COMALs because of POKEs used to control the cursor blink and because of other machine specific techniques used to control the screen image. I probably can't claim this program is "user friendly" by today's standards but it may qualify for "user sociable".

The program ends if there is an error while trying to read the puzzle file. I put more effort into trying to trap errors that might happen after the puzzle has been read. Among other things, an options menu allows toggling the autoclue mode, checking the answers, erasing wrong answer letters, getting some hints, showing the answers and saving a partially completed puzzle. Function keys F1 to F6 provide different views of the puzzle. The cursor keys move the point of typing horizontally and vertically within the visible layer and the up-arrow and shift/up-error keys move through the layers of the puzzle.

The puzzle file itself is my first real attempt at writing a crossword puzzle and my sister the crossword fan thought some of the clues misleading and perhaps unfair. Nevertheless, some people might enjoy it and I hope you do. [*The program is "source protected" on our disk, only can be RUN.*]



```
0320    plottext(x,y,"*")
0330 ENDPROC plot'point
0340  //
0350 PROC random'point
0360    n:=rnd(1,6)
0370 ENDPROC random'point
0380  //
0390 PROC plot'random(n)
0400    CASE n OF
0410    WHEN 1,2
0420      plot((x1+x)/n,(y1+y)/n)
0430      x:=(x1+x)/n; y:=(y1+y)/n
0440    WHEN 3,4
0450      plot((x2+x)/n,(y2+y)/n)
0460      x:=(x2+x)/n; y:=(y2+y)/n
0470    WHEN 5,6
0480      plot((x3+x)/n,(y3+y)/n)
0490      x:=(x3+x)/n; y:=(y3+y)/n
0500    ENDCASE
0510 ENDPROC plot'random
```

How To Submit Material

More and more computer systems now support COMAL. This makes it harder to do this newsletter. Articles and programs are needed, especially relating to the newest COMAL implementations. If you send in a program, put it on your disk twice:

SAVE "name"
LIST "name.lst"

Also, if possible, include a short (or long if you wish) article about the program. Put the article on the same disk. Store it as a SEQ text file (most word processors allow this as an option, such as Control-Z in PaperClip). Also include a printout of the article if you can (no need to send a printout of the program listing though).

Include your name and subscriber number on the disk label as well as in the first line of the program and article. Also put the computer type on the disk label so I know where to start with it. Eventually, all text and listings end up on my IBM PC hard disk. I use Big Blue Reader to transfer disks from C64 to IBM. I have a MatchPoint card in my IBM computer that can read Apple disks, allowing me to transfer Apple SEQ files to IBM. I have Access64 for my Amiga, which allows me to read C64 disks on my Amiga. I also should be able to get the Amiga drive to read the 3.5 IBM format disks. Finally I have a program that transfers IBM files between 5 1/4 and 3 1/2 disks.

Send your submissions to our new address:

COMAL Users Group, U.S.A., Limited
5501 Groveland Terrace
Madison, WI 53716

Material submitted is not returned.

How To Type In COMAL Programs

Line numbers are required for your benefit in editing a program (but are irrelevant to a **running** program). Thus line numbers usually are omitted when listing a COMAL program. It is up to **YOU** to provide the line numbers. Of course, **COMAL can do it for you.** Follow these steps to enter a COMAL program:

1) Enter command: NEW

2) Enter command: AUTO

3) Type in the program. When done:

Power Driver: Hit «*return*» key twice
PET COMAL 0.14: Hit «*return*» key twice
C64 2.0 Cart: Hit «*stop*» key
C128 2.0 Cart: Hit «*stop*» key
CP/M COMAL 2.10: Hit «*esc*» key
AmigaCOMAL: Hit «*esc*» key
IBM PC COMAL: «*control*» + «*break*»
Alder COMAL: «*control*» + C

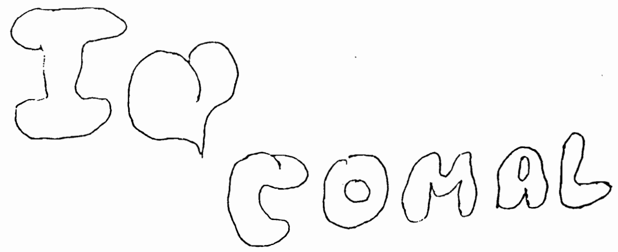
You may use both UPPER and lower case letters while entering a program (except PET COMAL 0.14; C64 COMAL 0.14 users should upgrade to Power Driver). COMAL automatically makes keywords UPPER case and variable names lower case (except PET COMAL 0.14). Also, you don't have to type leading spaces in a line. They are listed only to emphasize structures, and COMAL will insert them for you. You **DO** have to type a space between keywords in the program.

Long program lines: If a complete program line will not fit on one line, we will continue it onto the next line and add //wrapline at the end (and possibly print it in *italic type*). **You** must type it as one continuous line.

Variable names, procedure names, and function names can be a combination of:

abcdefghijklmnopqrstuvwxyz0123456789'_][\

The «*left arrow*» key in the upper left corner of the C64 and C128 keyboard is valid. COMAL 2.0 converts it into an underline. The C64 and C128 computers use a «*British pound*» £ symbol in place of the \ backslash.



ORDER FORM Subscriber #__________

Name:____________________

Street:____________________

City/St/Zip:____________________

Visa/MC#:____________________

Exp Date:________ Signature:____________________

May '89-Prices subject to change without notice

TO ORDER:

- Fill in subscriber# / address (above) (new subscribers write new for subscriber #)
- Check [x] each item you want to order
- Add up items/shipping/handling (fill in below) (shipping fee is often less than the max listed)
- Send check/money order or charge it (above)
- Charge orders may call 608-222-4432
- or Mail to: **COMAL Users Group, U.S.A., Ltd. 5501 Groveland Terrace, Madison, WI 53716**

SUBSCRIPTIONS:

Expired subscribers must renew before they may order at subscriber prices (renewal starts with the issue where you left off). New subscribers can order at the same time as subscribing.

[_____] <=How many issues? $4 per issue
(Canada/APO add $1 per issue, 1st Class)
[_____] <=How many disks? $9 per disk

[_] $4.95 each for backissues; circle ones wanted:
1 2 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20/21 22 23 24
[_] $9.95 each, C64 COMAL Today Disks; circle to order:
1/2 3/4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20/21 22 23 24

NOTICE: **All orders must be prepaid - in US dollars. Minimum order $10. Shipping is extra: $3 minimum shipping (does not apply to subscription only orders); Canada, APO & 1st Class add $1 more per book and newsletter issue. Newsletter is published as time permits (no set schedule); size and format varies. Cancelled subscriptions receive no money back. Orders accepted from Canada and USA only. Prices shown are subscriber prices and reflect a $2 discount. Allow 2 weeks for checks to clear. $15 charge for checks not honored. Wisconsin residents add 5% sales tax and if your county charges a county tax, you must include it as well.**

ENTER TOTALS HERE:

Item Total ($10 minimum):$____________
Ship Total ($3 minimum):$____________
Grand Total enclosed:$____________

BACKISSUES - COMAL Today

Price per issue: $4.95 (no extra shipping other than min) (circle numbers on the left; matching disks are also available)

1 - Original first issues still available (not reprints). Only a few copies left.
2 - PRINT FILE versus WRITE FILE; Recursion; Stack overflow; Functions/Parameters
3 - sold out
4 - Graphics screen dump in COMAL; spirolateral; Dodge 'em; Music; arrays /parameters
Issues 1-4 Spiral bound: (see COMAL Yesterday/book)
5 - How COMAL statements are stored; strings; Target; Inventory; Pitfall Harry
6 - Memory maps; 2.0 & 0.14 compared; Shadow letters; Draw molecule; Function keys
7 - COMAL Standards meeting; Book reviews; Function keys; External procs
8 - Sprites; Soundex; Forest; Sound; Fun print; Recursion; Font sprites; Depreciation
9 - Modems; Ascii codes; TRAP; Function keys; Metamorphose sprite; Bitmaps; Sizzle; C64 cart internal structure & token table; Rod the Roadman; Strings; Icon Editor
10 - Sorts; Font Editor; Draw; Walker; Phone Database; Missing Letters; Designer; Compressed bitmaps; Meta; Compare disk files; Easy Curves
11 - 2.0 keywords chart; Disk directories; Pop Over; Graphics Editor System; Ram disk
12 - Cart package keywords chart; Rabbit; 3D Fractals; Cart schematics; 2 column printing; Pic Finder; Benchmarks; Free form database; Transfer programs from 0.14 to 2.0; Kelly's Beach
Index to issues 1-12, 4,848 entries, 56 pgs; (see books)
13 - Superchip keywords chart; Sprites; Wheel of Fortune; Sets; Benchmarks; BASIC into COMAL; Encryption; C128 package; Kastle
14 - Outliner; Listerine; Scope; Stacks; CASE statement; Calendar; Multi-directory; Modems
15 - Over 60 0.14 Proc/Func listed!; Dr Who database; Program construction; Wheel of Fortune
16 - Smart file reader; Text input window; Magic squares; Easy instructions; Read & Run; Learn subtraction; NIM; Tiny directory; Sorts
17 - COMAL Kernal in full; Ram expander; Sprites; Calculate PI; 0.14 graphic/sprite chart
18 - Reversi; Poetry; Hammurabi; Puzzle; Zip Zone; Fractal geometry / Mandelbrot; stacks
19 - Power Driver keywords chart; Sample book pages; Coloring book; Rotating 3D
20/21 - Files; Black box; Best of 1-19
22 - Walking sprites; Animals; Maze; Music
23 - Message board; Programming; Graphing

SYSTEMS: **ORDER FORM** **SYSTEMS:**

C64 - Disk Loaded

[_] **Power Driver Complete**db - all of the items below, Power Box, Starter Kit, and keyboard overlay. $49.95 + $6 shipping
[_] **Add $5 for Doc Box¤ binder/slipcase**

[_] **Power Driver** interpreter and 20 lesson tutorial (with turtle tutor) on disk. $9.95

[_] **Power Box**db - includes Power Driver interpreter and **compiler**, 3 utility disks, a toolbox of about 250 procedures and functions and Power Driver note pages. $29.95 + $4 ship

[_] **Starter Kit** - 12 issues of *COMAL Today*, 56 page index, 2 books and 5 disks! Over 1,000 pages! $29.95 + $4 ship

[_] **C64 Keyboard Overlay**: excellent condensed command reference. $3.95 + $1 shipping

[_] **Doc Box¤ binder/slipcase** option ... $7.95

C64 - Cartridge

[_] **COMAL 2.0 Cart Complete***
The C64 cartridge plus all the options below (except Superchip) ... Tutorials, references, packages, applications ... $179.95 + $7 ship

[_] **COMAL 2.0 Cart Deluxe***
64K cartridge; 3 reference books: *2.0 Keywords, Cart Graphics & Sound, Common COMAL Reference* plus 4 demo disks. $124.95 + $3 ship

[_] **COMAL 2.0 Cartridge***
64K cartridge with empty socket for up to 32K EPROM. Plain, no documentation. $99.95

[_] **Deluxe Option**db: three books: *2.0 Keywords, Cart Graphics & Sound, Common COMAL Reference*; 4 cart demo disks. $29.95+$4 ship

[_] **Packages Option**db: three books: *COMAL 2.0 Packages, Packages Library 1* (17 packages), *Packages Library 2* (24 packages); Superchip on Disk (9 packages). $36.95 + $4 ship

[_] **Applications/Tutorial Option**db: three books: *COMAL Collage, 3 Programs In Detail, Graph Paper*. $36.95 + $4 ship

[_] **Super Chip**: plug in chip for C64 cart; adds about 100 commands to the cartridge. $24.95

[_] **Doc Box¤ Binder/Slipcase Option** ... $7.95

CP/M Systems (includes C128 CP/M mode)

[_] **CP/M COMAL 2.10**
Full COMAL system disk plus the DEMO disk, packed in a Doc Box¤ with manual.
Works in C128 CP/M mode. $39.95 + $4 ship

[_] **Compiler Option** (CP/M COMAL RUNTIME system). $5.95

[_] **C128 Graphics Option**: Package disk $9.95

[_] **CP/M Package Guide Option**: Reference on how to write packages $14.95 + $1 ship

[_] **Common Comal Reference** option: $16.95 +3 ship

IBM PC & compatibles

[_] **UniComal IBM PC COMAL 2.2***
Full fast system, with extensive reference & tutorial packed in a Doc Box. $495 +$5 ship
$50 discount if prepaid by check, M.O., Visa, MC

[_] **Compiler (PLUS) Option**: adds runtime compiler and Communication Package, with manuals packed in a Doc Box. $295 +$4 ship
$50 discount if prepaid by check, M.O., Visa, MC

[_] **Common Comal Reference** option: $16.95 +3 ship

[_] **Option**: UniDump $45 (for laser printers)
[_] **Option**: UniMatrix $165 (matrix package)
[_] **Option**: Hercules Support $85
[_] **Option**: Btrieve interface $25 /$110 multi
[_] **Option**: XQL interface $110
[_] **Option**: Update 2.1 to 2.2 - $45

Amiga (500, 1000 & 2000 models)

Now in final testing, should be available soon. Estimated price is $99.95 + $4 shipping. Send us a Self Addressed 25¢ Stamped Envelope and we will send you info on ordering as soon as it is available in the USA.

A compiler option will probably also be available.

Previously referred to as German Amiga COMAL, it correctly is AmigaCOMAL from ComWare.

We also are awaiting new info on the Mytech (now called Alder) COMAL for the Amiga.

db = Doc Box pages
* = subject to customs/ship variations/availability
¤ = while supplies last (out of print)

Disks are $9.95 each. Unless the label on the disk you receive specifically states that you may give out copies, our disks may not be copied or placed into club disk libraries. **Choose from the disks below:**

[ ] IBM Special Series Disk #1
[ ] IBM Special Series Disk #2 (Test System)

[ ] CP/M COMAL Demo Disk ($5)

[ ] Beginning COMAL disk §
[ ] Foundations with COMAL disk §
[ ] COMAL Handbook disk §
[ ] New: **Introduction to COMAL 2.0 disk** §
[ ] Today INDEX disk § (2 disks count as 1)
[ ] Games Disk #1 (0.14 & 2.0)
[ ] Modem Disk (0.14 & 2.0)
[ ] Article text files disk

Today Disks:
[ ] Today Disk (one disk type--circle choices):
1 2 3 4 5 20&21
[ ] Today Disk (double sided -- circle choices):
6 7 8 9 10 11 12 13 14 15 16 17 18 19 22 23
[ ] Today Disk 24

0.14 Disks:
[ ] Data Base Disk 0.14
[ ] New: Power Driver Tutorial Disk
[ ] Auto Run Demo Disk
[ ] Paradise Disk
[ ] Best of COMAL
[ ] Bricks Tutorial (2 disks count as 1)
[ ] Utility Disk 1
[ ] Slide Show disk (circle which): 1 2
[ ] Spanish COMAL
[ ] User Group 0.14 disks (circle numbers):
1 2 3 4 5 6 7 8 9 10 12

2.0 Disks:
[ ] Data Base Disk 2.0
[ ] Superchip Programs disk
[ ] Read & Run
[ ] Math & Science
[ ] Typing disk (2 disks count as 1)
[ ] Cart Demo (circle which): 1 2 3 4
[ ] 2.0 user disks (circle choices): 11 13 14 15
§ = these disks assume you have the book

Note: Some disks may be supplied on the back side of another disk. Disk format is Commodore 1541 unless specified otherwise. We replace any defective disk at no charge if you return the disk with a note explaining what is wrong with it. Some disks are being reduplicated and appropriately relabeled.

[_] $10.95 **Sprite Pak**
Two disk set. Huge collection of sprite images, sprite editors, viewers, and other sprite programs. For 0.14 and 2.0.

[_] $12.95 **Font Pak**
Three disk set. Collection of many different character sets (fonts) for use with 0.14 and 2.0 including special font editors!

[_] $14.95 **Graphics Pak**
Five disk set. Picture heaven. Includes Slide Show, Picture Compactor, Graphics Editor and lots of pictures (normal and compacted)

[_] $29.95 **Sprite,Font & Graphics Pak**
All ten disks mentioned above!

[_] $9.95 **C128 CP/M Graphics**
Graphics package on disk for use with CP/M COMAL on the C128. Includes turtle graphics and preliminary Font package.

[_] $10.95 **Guitar Disks**
Three 0.14 disk set. Teaches guitar by playing songs while displaying the chords and words.

[_] $14.95 **Cart Demo Disks**
Four disks full of programs demonstrating the many features of the C64 2.0 cartridge.

[_] $10.95 **Shareware Disks**
Three disk set. Includes a full HazMat system (Hazardous Materials), an Expert System, Finger Print system, Traffic Calc, and a BBS program.

[_] $14.95 **Superchip On Disk**
All the commands of Super Chip (but not the Auto Start feature) disk loaded.

[_] $24.95 **Super Chip Source Code**
Full source code with minimal comments. Customize your own Super Chip. Add commands. Remove the ones you don't need.

[_] $14.95 **2.0 User Group Disks**
Four disks set for the C64 COMAL cart.

[_] $29.95 **0.14 User Group Disks**
Twelve disks (User Group disk 9 is a newsletter on disk system, double disk).

[_] $29.95 **European Disk Set**
Twelve 2.0 disk set. Find out what COMALites in Europe are doing.

[_] $2.95 **C64 COMAL 2.0 Keywords**db
#4 best seller for Feb 88 lists all the keywords built into the cartridge (including all 11 built-in packages) in alphabetic order complete with syntax and example. **+$1 ship**

[ ] $16.95 **Beginning COMAL**¤
#8 best seller by Borge Christensen
333 pages - General Textbook
Beginners text book, elementary school level, written by the founder of COMAL. This book is an easy reading text. You should find Borge has a good writing style, with a definite European flair. **+$3 ship**
[_] **Optional Matching Disk** of programs. $9.95

[_] $3.95 **COMAL From A to Z**¤
#1 all time best seller by Borge Christensen
64 pages - Mini 0.14 Reference book
Written by the founder of COMAL. **+$1 ship**

[_] $3.95 **COMAL Workbook**¤
#1 best seller for Feb 88 by Gordon Shigley
69 pages - 0.14 Tutorial Workbook; Companion to the Tutorial Disk, great for beginners, full sized fill in the blank style. **+$1 ship**
[_] **Tutorial Disk Option**: $9.95

[_] $3.95 **Index - COMAL Today 1-12**¤
#9 best seller by Kevin Quiggle
52 page, 4,848 entry index to COMAL Today. The back issues of COMAL Today are a treasure trove of COMAL information and programs! This index is your key. **+$1 ship**
[_] **Index Disk Option**: $9.95

[_] $16.95 **Common COMAL Reference**db
#3 best seller for Feb 88 by Len Lindsay
238 page detailed cross reference to the COMAL implementations in the USA (formerly COMAL Cross Reference) Covers: C64 COMAL 2.0, C128 COMAL 2.0, CP/M COMAL 2.10, and UniComal IBM PC COMAL. **+$3 ship**

[_] $19.95 **COMAL Yesterday Issues 1-4 of COMAL Today - Spiral bound +$3 ship**

[_] $12.95 **Library of Funcs/Procs**db
#1 best seller Dec 87 by Kevin Quiggle, 80 pgs, over 100 0.14 procs/funcs, with disk. **+$1 ship**

[_] $4.95 **Cart Graphics & Sound**¤db
#6 best seller by Captain COMAL
64 pages - 2.0 packages reference guide to all the commands in the 11 built in cartridge packages. **+$1 ship**

[_] $14.95 **COMAL 2.0 Packages**db
#7 all time best seller by Jesse Knight
108 pgs with disk - package reference. How to write a package in Machine Code; includes C64 comsymb & supermon. **For advanced users. +$2 ship**

[_] $14.95 **Package Library Vol 1**db
compiled by David Stidolph
76 pages with disk - package collection
17 packages ready to use, many with source code, plus the Smooth Scroll Editor! **+$1 ship**

[_] $14.95 **Package Library Vol 2**db
67 pages with disk - package collection
24 example packages ready to use, most with source code, plus Disassembler, Re-Linker, De-Linker, Package Maker, Package Lister, and more. (includes windows). **+$1 ship**

[_] $14.95 **COMAL Collage**db
by Frank and Melody Tymon
168 pages with disk, 2.0 programming guide, including graphics and sprites tutorial with many full sized example programs. **+$2 ship**

[_] $12.95 **3 Programs in Detail**db
82 pages with disk by Doug Bittinger
Three 2.0 application programs explained: Blackbook (name/address system), Home Accountant, and BBS. **+$1 ship**

[_] $12.95 **Graph Paper**db
52 pages with disk by Garrett Hughes
Function graphing system for COMAL 2.0. The program can't be LISTed. Includes a version for the Commodore Mouse. **+$1 ship**

[_] $12.95 **COMAL Quick /Utility 2 & 3**db
#2 best seller Dec 87 by Jesse Knight
20 pgs with 2 disks, fast loading COMAL 0.14, printer programs, utility programs. **+$1 ship**

[_] $14.95 **CP/M COMAL Package Guide**db
The guide to making your own packages for CP/M COMAL by Richard Bain - 76 pages - advanced package reference. **+$1 ship**

[=] **$16.95 Foundations with COMAL**¤ sold out
[=] **$17.95 Introduction to COMAL 2.0**¤ sold out

db = Doc Box pages
* = subject to customs/ship variations/availability
¤ = while supplies last (out of print)

Filename Conventions

We try to include more than just COMAL programs on our disks. To help you determine a file type, we've adopted naming conventions. Use these conventions to avoid unnecessary confusion over files, especially if you are submitting them to us.

All computer systems
NAME can be up to 8 characters:

Name	Meaning (all systems)
NAME	COMAL program (IBM, CP/M, Amiga automatically tack on their own extension)
NAME.LST	Listed program / proc / func
NAME.DSP	Displayed program
NAME.EXT	External procedure
NAME.DAT	Data file
NAME.TXT	Text file
NAME.DOC	Documentation file
NAME.BAS	Basic program
NAME.RAN	Random file

Commodore only
Filename can be up to 16 characters:

Suffixed	Prefixed	Meaning (C64/C128)
NAME.PROC	PROC.NAME	PROC listed to disk
NAME.P		PROC listed to disk
NAME.FUNC	FUNC.NAME	FUNC listed to disk
NAME.F		FUNC listed to disk
NAME.DAT	DAT.NAME	Data file
NAME.TXT	TXT.NAME	Text file
NAME.DOC	DOC.NAME	Documentation file
NAME.EXT	EXT.NAME	External proc/func
	SHAP.NAME	Sprite shape file (for **loadshape**)
	IMAG.NAME	Sprite shape file (for **read** file)
	FONT.NAME	Font file (for **loadfont**)
	SET.NAME	Basic type font file
NAME.BAT	BAT.NAME	Batch file
	SNG.NAME	Song file
	HRG.NAME	Color COMAL picture
NAME.HRG		Black/White bitmap
	CRG.NAME	Compact Color picture
NAME.CRG		Compact B/W picture
	ICON.NAME	Print Shop Icon file
	SCRN.NAME	Text Screen file
	POP.NAME	Mergeable Popover
NAME.POP		Program with Popover
	CHIP.NAME	Super Chip Program

Amiga Filenames
Filename can be quite long, including several periods, which usually are significant:

Name.info	WorkBench Icon Picture
Name.sav	AmigaCOMAL program
Name.sav.info	Icon for AmigaCOMAL program
Name.sav.backup	Auto Backup AmigaCOMAL program

Also, these drawers (subdirectories) are setup:

Programs	AmigaCOMAL programs
Externals	External proc/func for AmigaCOMAL
Packages	Packages for AmigaCOMAL

Amiga Startup-Sequence

I broke my startup-sequence file into three parts. This allows me to copy part of it into RAM first, then execute it. This is more efficient (saves grinding the disk drive). It also gives a choice at boot up for one of two types of systems appropriately named finish-min and finish-max. The back cover lists the contents of my boot disk.

Startup-Sequence

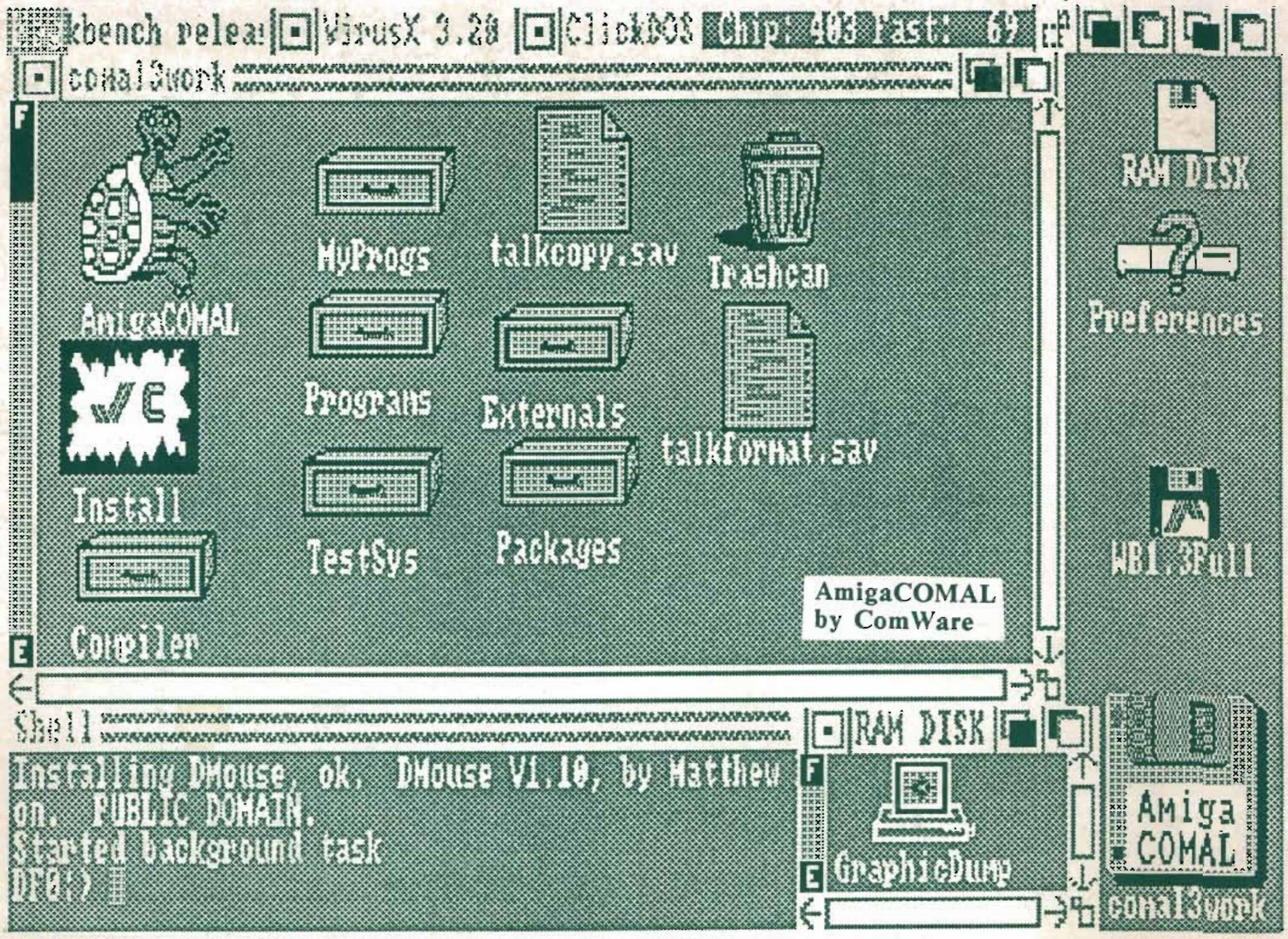
```
Sys: System/FastMemFirst
C:AddBuffers DF0: 30
makedir ram:t
assign t: ram:t
setclock opt load
say full system?
inquire Full System?
If WARN Then
  C:Copy S:Finish-Max TO RAM:Finish-Sequence
Else
  C:Copy S:Finish-Min TO RAM:Finish-Sequence
Endif
Say System Copied.
C:Resident c:Resident pure
C:Residnet CLI L:Shell-Seg SYSTEM pure add
C:path ram: c: sys:system sys:utilities
C:Mount NEWCON:
C:NewShell newcon:0/150/450/50/Shell RAM:Finish-Sequence
C:EndCLI > nil:
```

Finish-Min

```
Say Starting Finish Now.
stack 40000
SYS:System/SetMap usa1
C:SetPatch >nil:
C:Prompt "%S> "
LoadWB delay
```

Finish-Max

```
Say Starting Finish Now.
stack 40000
Resident C:assign
Resident C:info
Resident C:list
Resident C:rename
Resident C:path
Resident C:cd
Resident C:dir
Resident C:run
Resident C:copy
Resident C:delete
Resident C:DiskChange
Copy SYS:System/DiskCopy TO Ram:DiskCopy
C:VirusX
SYS:System/DOS -i ;clickDOS iconized
C:FF >nil: -0 ;speedup text
SYS:System/setmap usa1
C:SetPatch >nil:
C:Dmouse -a2 -s500 -L1 -A2 -C4 -10003
C:Prompt "%S> "
Alias endshell endcli
Alias Ren Execute s:dpat rename []
Alias clear echo "*E[0;0H*E[J"
Alias reverse echo "*E[0;0H*E[41;30m*E[J"
Alias normal echo "*E[0;0H*E[40;31m*E[J"
Alias cat dir
LoadWB delay
c:RunBack -5 sys:system/nag
```



```
Installing DMouse, ok.  DMouse V1.10, by Matthew
on.  PUBLIC DOMAIN.
Started background task
DF0:>
```

WorkBench 1.3
BOOT DISK

```
======================
C (dir)
AddBuffers
AddMem
Ask
Assign
Avail
BindDrivers
Break
CD
ChangeTaskPriority
Copy
Date
Delete
Dir
DiskChange
DiskDoctor
DMouse
Echo
Ed
Else
EndCLI
EndIf
EndSkip
Eval
Execute
Failat
Fault
FF
GetEnv
IconX
If
Info
Inquire
Install
Join
Lab
List
LoadWB
Lock
Makedir
Mount
NewCLI
NewShell
Path
Play
Prompt
Protect
Quit
Relabel
Rename
Resident
Run
RunBack
Say
SetClock
SetDate
SetEnv
SetPatch
Show
Skip
Stack
Status
Type
VirusX
Wait
WhereIs
Which

DEVS (dir)
ClipBoard.Device
Clipboards
DST.device
KeyMaps (subdir)
  usa1
MountList
Narrator.device
Parallel.device
Printer.device
Printers (subdir)
  Epson
  Generic
  HP_LaserJet
RamDrive.device
Serial.device
System-configure
Disk.info
Fonts (dir)
Diamond (subdir)
  12
  20
Diamond.font
Topaz (subdir)
  11
Topaz.font
L (dir)
Aux-Handler
Disk-Validator
DMouse-Handler
FastFileSystem
Newcon-Handler
Pipe-Handler
Port-Handler
Ram-Handler
Shell-Seg
Speak-Handler
Libs (dir)
diskfont.library
icon.library
info.library
mathieeedoubbas.librar
mathieeedoubtrans.libr
mathtrans.library
translator.library
version.library
Prefs (dir)
.info
Pointer.info
Preferences
Preferences.info
Printer.info
Serial.info
Prefs.info
S (dir)
.dosrc
CLI-Startup
DPAT
Finish-Max
Finish-Min
Nag.1988
Nag.1989
Nag.1990
PCD
Shell-Startup
SPAT
Startup-Sequence
ThruDaWindow
Shell
Shell.info

System (dir)
.info
CLI
CLI.info
Diskcopy
Dos
Dos.info
FastMemFirst
FastMemFirst.info
Format
MergeMem
MergeMem.info
Nag
Nag.config
Nag.info
NoFastMem
NoFastMem.info
SetMap
SetMap.info
System.info
T (dir)
Utilities (dir)
.info
CMD
CMD.info
GraphicDump
GraphicDump.info
InstallPrinter
InstallPrinter.info
More
More.info
PrintFiles
PrintFiles.info
Utilities.info
```