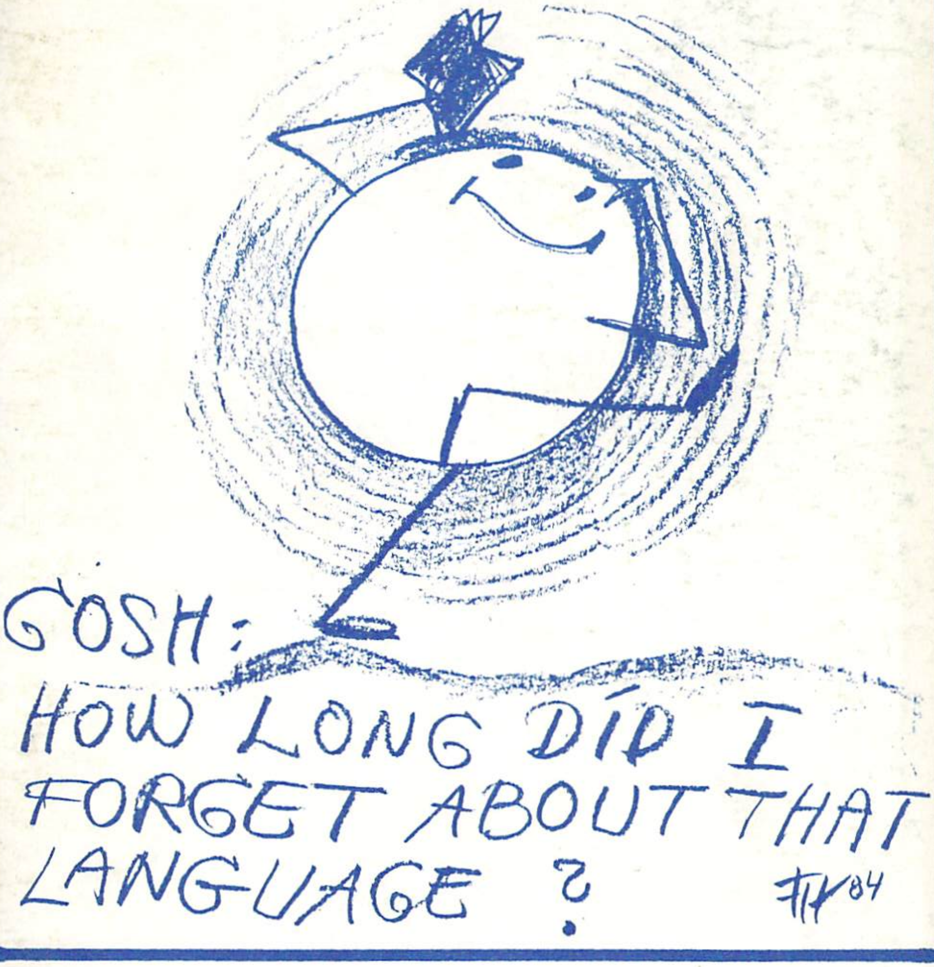
THE AMAZING ADVENTURES OF CAPTAIN COMAL™ BOOK 4

CARTRIDGE GRAPHICS AND SOUND
AND OTHER PACKAGES

$9.95



THE AMAZING ADVENTURES OF CAPTAIN COMAL
Book 4

CARTRIDGE GRAPHICS AND SOUND
and other packages

by Captain COMAL's Friends

Published by COMAL Users Group, U.S.A., Limited
5501 Groveland Terrace
Madison, WI 53716-3251

phone: 608-222-4432

First Edition Cover Art by Frank Hejndorf

This edition is for the C64 COMAL 2.01 Cartridge. It applies to both the EPROM and ROM versions since both are identical except for the start up screen. The cartridges are distributed in North America by COMAL Users Group, U.S.A., Limited.

Your comments on the book are welcome and appreciated. We tried to keep it concise and to the point. COMAL TODAY newsletter will have a special section just for the CARTRIDGE starting in 1985. Make sure you are a subscriber.

ISBN 0-928411-02-8 Printed in U.S.A.

TABLE OF CONTENTS

Running COMAL 0.14 programs in COMAL 2.0

COMAL programs written in disk loaded COMAL 0.14 are upward compatible with the C64 COMAL 2.0 Cartridge. **BUT** they are not SAVE and LOAD compatible (if you try to LOAD a COMAL 0.14 program file into COMAL 2.0 you will get an error message). To transfer a COMAL 0.14 program to run under COMAL 2.0 you must follow these steps:

1) You can keep your COMAL cartridge plugged in. Switch to BASIC mode by issuing the command: BASIC.
2) LOAD in COMAL 0.14 from your disk as usual:
```
    LOAD "BOOT*",8   //   RUN
```
3) While in COMAL 0.14 LIST the program to disk:
LIST "NAME.L"
4) Do the same thing with any other COMAL 0.14 programs you wish to transfer to COMAL 2.0.
5) Now switch back to COMAL 2.0:
Turn off the computer, then back on again.
6) ENTER each program previously LISTed to disk:
```
    ENTER "NAME.L"
```
7) After the program is successfully ENTERed, you may wish to SAVE it to disk with 2.0 SAVE command for future use.

Remember, graphics is not COMAL, it is an environment (now called a package). Plain COMAL programs should transfer fine. Graphics and sprites now need () parentheses in 2.0 and a USE GRAPHICS and USE SPRITES at the beginning of the program.
```
FORWARD 9    becomes    FORWARD(9)
```

More tips will be in COMAL TODAY newsletter.

SOME OF THE CARTRIDGE FEATURES AND HOW TO USE THEM

Professional COMAL is available for the Commodore 64. It is called the C64 COMAL 2.0 Cartridge and contains the full version 2.0 COMAL implementation for the Commodore 64 computer system. Preliminary versions were available for a year before the final version was released. The preliminary versions were called 2.00 and 2.01y. The final version was released as version 2.01 (they did not bump up the number to version 2.02 as expected). In this book, we will refer to it simply as the cartridge or COMAL.

The cartridge contains the complete COMAL system and more. The COMAL system is detailed in the **COMAL HANDBOOK**. In addition, 11 packages (including graphics and sound) are built into the cartridge. These packages are considered 'tack-ons' to COMAL. You must issue a USE command before you can use features in a package. Version 0.14 disk loaded COMAL was more primitive, and just built in the graphics, turtle and sprites as part of COMAL, when they really are not COMAL. Commands inside packages can have parameters, and when parameters are used, they must always be enclosed inside parentheses ( ). Thus note the change in syntax for graphics and spites from version 0.14 to version 2.0:

```
Version 0.14:     FORWARD 10
Cartridge 2.0:    FORWARD(10)
```

CORRECTS SOME COMPUTER PROBLEMS

The COMAL Cartridge even corrects some problems in the C64 itself. For example, colors on the graphics screen are in 8x8 (with a limit of 2 colors) or 8x4 (with a limit of 4 colors) pixel blocks. Once the color limit of 2 or 4 is reached, the C64 does NOT reset the pixels back to 'unused' if the whole block is filled with one color. COMAL resets colors automatically for you. Also the Commodore 1541 Disk Drive problem with RANDOM or RELATIVE files is corrected by the COMAL system - and still gives you faster access times than BASIC.

INPUT STATEMENTS

An INPUT statement now can accept up to 120 characters (on continued screen lines). Also, the input fields are now 'protected' fields. Cursor up and Cursor down are ignored, as are reverse on and off, and there is no 'quote mode'. But the interesting feature is that while in an INPUT request, the CLR (clear screen) key will only clear the input field and HOME (cursor home) key will put the cursor on the first position of the input field. In addition to the normal INPUT statement you now can use the professional INPUT AT specifying the row and column to start at as well as the maximum length of the input field:

```
INPUT NAME$
INPUT "NAME> ": NAME$
INPUT AT 9,1: "NAME> ": NAME$
INPUT AT 9,1,10: "NAME> ": NAME$
```

The above input statements all input characters into the variable NAME$. The first one is the most basic type. Since it includes no prompt, COMAL will use the question mark (?) as the prompt (to have no prompt at all, use a prompt of ""). The next line uses a prompt of "NAME> ". The next example starts the INPUT request at row 9 column 1 (the N in NAME> will be in that position). Finally, the last example includes 10 as the maximum permitted input. COMAL will not let the user type more than 10 characters for the reply! Try this example program:

```
10 PAGE // clear screen
20 INPUT AT 9,1,10: "NAME> ": NAME$
30 PRINT NAME$
```

Also it is important to note that both RETURN and SHIFTed RETURN are the same now. Use the STOP key instead of SHIFTed RETURN.

EASY EDITING

FIND and CHANGE commands are available:

```
FIND " PROC "
```

This will find all program lines with " PROC " in it. COMAL will list one line at a time. Hit RETURN for the next one (you may edit the line first if you wish). Hold the RETURN key down for a continuous list of all lines. Issue a SELECT "lp:" command first and the list of lines will print on the printer.

```
CHANGE "temp","final"
```

This will change all occurances of "temp" into "final". COMAL asks your approval before it makes each change (the line is displayed with the target flashing). Hit N for No change or hit the RETURN key and it will be changed. Use the STOP key to cancel the command. While a line is displayed you may also hit the COMMODORE LOGO key - this allows you to edit that line first and then continue (but the CHANGE will not be made for that line unless you do it yourself).

UPPER and lower case are considered different. Therefore, if you search for "proc" it will not FIND "PROC". Be careful.

UNDERLINE CHARACTER

The left arrow key (top left key on keyboard) can be used as part of a variable name, and if used, it will be listed on the printer and screen as underline. The Commodore key plus the @ key will directly produce the underline character. Underline is CHR$(164) in PETASCII or CHR$(95) in ASCII.

The cartridge also converts all 'control code' character in print statements into the code number with quotes on each side. This allows a program listing to print on any printer even with embedded control codes. For example, the HOME CURSOR control code is 19. In BASIC we would say PRINT "[HOME]". When you press the HOME key in between the quote marks a reverse field S appeared. It represents HOME cursor. The cartridge would automatically convert this line into PRINT ""19"". It looks funny at first - but it makes sense. Simply remove the control character and replace it with a quote mark, the code number, and another quote mark. Of course, if you don't like it this way you always can just use PRINT CHR$(19).

UPPER or lower case

The cartridge will let you use either UPPER or lower case letters when entering COMAL commands and statements. Thus to list the program you can use any of the following (they are all considered the same by COMAL):

```
LIST     list     List     LiST     liST  ...
```

However, when listing a program, keywords will be in UPPER case and variable names in lower case:

```
FOR delay:=start TO finish DO PRINT "Testing"
```

You can change this if you wish, and when LISTing a program to disk to transfer to version 0.14 you MUST change the keywords into lower case. Changing this default is accessed thru the SYSTEM package, so let's get on with the packages...

A NOTE ABOUT PACKAGES

It is possible to extend COMAL by use of PACKAGES. The added commands in each package is not available until COMAL has been instructed to **USE** the package. This gives you control over when the commands are active, and memory is not taken up by them when they are not needed in a program. Eleven packages are built into the cartridge, and others can be loaded from disk.

To use the commands in a package, you must issue the command USE followed by the name of the package:

NAME	ABILITIES
DANSK	Displays message texts in Danish language
ENGLISH	Displays message texts in English language
FONT	Allows user defined character sets
GRAPHICS	Adds 48 different graphic commands
JOYSTICKS	Adds joystick control
LIGHTPEN	adds lightpen control (port #1 only)
PADDLES	Adds paddle control
SOUND	Adds sound and music commands
SPRITES	Adds sprite commands including animation
SYSTEM	Gives control over some of the COMAL system
TURTLE	All GRAPHICS package plus LOGO abbreviations

Examples:
```
USE GRAPHICS
USE JOYSTICKS
USE FONT
```

More than one package can be active at one time, but by adding them into the command table, some memory is used (although this should not be a problem in a 30K system). These added commands are not GLOBAL and do not automatically pass into a CLOSED procedure, so you will have to issue the USE command within that procedure in order to use the commands, or IMPORT the specific ones needed.

The command DISCARD removes all packages from the name table. It is a direct command and cannot be used in a running program.

SYSTEM PACKAGE

The SYSTEM package allows you to change some of the system defaults and gives you some extra commands as well. Before you can use any of this you **MUST** tell COMAL you want to use the SYSTEM package. Just issue this command:

USE SYSTEM

The following pages detail what is available in the SYSTEM package.

BELL

```
BELL(<number>)
BELL(3)
```

This makes the sound heard when you turn on the computer with the COMAL cartridge in place. The sound can be long or short, depending on the number used (3 is a good choice for many uses). The bell will 'ring' the number of times specified (from 0 to 255 times).

CURCOL

```
CURCOL
X:=CURCOL
```

This function returns the cursor's current position on the the screen. This is useful with CURROW to remember where the cursor currently is before you move it someplace else.

CURROW

```
CURROW
R:=CURROW
```

This function returns the row that the cursor is currently on. It is useful with CURCOL to identify the current location of the cursor.

DEFKEY

```
DEFKEY(<function key number>,<string to issue$>)
DEFKEY(1,"LIST"13"")
DEFKEY(1,"LIST"+CHR$(13))
DEFKEY(3,"COPY"+CHR$(13)+","+CHR$(34)+"1:*")
DEFKEY(3,"COPY"13","""1:*")
```

You now have full control over the function keys. DEFKEY allows you to set up any function key to return any string of characters you wish, up to the line length limit. The first example above sets function key 1 (F1) to be the LIST command with a carriage return (note the "13" is the new way of saying CHR$(13) for carriage return). The second example uses the old method. Both achieve the same result.

The third example sets up function key 3 (F3) to be a 'copy' key for a dual disk drive. To copy any file from drive 0 to drive 1 simply do a CAT for drive 0. Then simply put the cursor in front of any program you wish to copy and press F3. The last example is the same, but using the new method of indicating CHR$ via quotes around the CHR value.

Within a running program, the function keys are internally referred to as F11 - F18 (10 more than usual). This allows you to set up function keys inside a program and not affect their settings outside the program. The previous values of F11 - F18 are cleared by the RUN command back to their default values.

FREE

```
FREE
MEM'LEFT:=FREE
PRINT FREE
```

While the SIZE command is part of COMAL and provides more complete details on memory used, it is not accessible from a running COMAL program. FREE is a function that will return the amount of FREE user memory available. This could be useful when dealing with external procedures or large arrays.

GETSCREEN

```
GETSCREEN(<string$>)
GETSCREEN(screen1$)
```

This command assigns the entire text screen, including colors and cursor location to the string variable indicated. The variable must be DIM'ed to 1505 before using it (ie, DIM SCREEN1$ OF 1505).

GETTIME$

GETTIME$	returns the time as a string: HH:MM:SS.T

```
PRINT GETTIME$
THIS'TIME$:=GETTIME$
```

The current time is returned by GETTIME$ as a string in the format: HH:MM:SS.T where HH is the hour, MM is the minutes, SS is the seconds and T is the tenths of seconds. The following program is a complete digital clock program:

```
10 USE SYSTEM
20 PAGE // clear screen
30 WHILE TRUE DO PRINT AT 10,10: GETTIME$(1:8)
```

HARDCOPY

```
HARDCOPY(<filename$>)
HARDCOPY("lp:")
HARDCOPY("0:textscreen")
```

This sends a text screen dump to the location specified. The first example sends it to the printer while the second creates a disk file called TEXTSCREEN and sends it there. Note that CONTROL P is the same as HARDCOPY("lp:") and produces a text screen dump to the printer.

INKEY$

```
INKEY$
CHOICE$:=INKEY$
PRINT INKEY$
```

This is a very useful addition. The keyword KEY$ is part of the COMAL system and simply looks at the keyboard buffer once to see if a key has been pressed. If you want to actually wait for a key to be pressed, you can now use INKEY$. In addition, COMAL will blink a cursor while waiting. INKEY$ is a string function and returns the character matching the key pressed.

KEYWORDS'IN'UPPER'CASE

```
KEYWORDS'IN'UPPER'CASE(<true/false>)
KEYWORDS'IN'UPPER'CASE(false)
```

You will notice that COMAL tries to make your program listings even easier to follow by automatically listing COMAL keywords in UPPER case and your variable names, procedure names, etc in lower case. This is compatible with COMAL systems running on other computers systems (mostly in Europe). Hopefully you will like this default method of listing. However, you can easily change it with the KEYWORDS'IN'UPPER'CASE command.

NOTE: If you are LISTing a program to disk to transfer to version 0.14 you **MUST** first issue this command:

```
     KEYWORDS'IN'UPPER'CASE(false)
```

This is because version 0.14 does not recognize UPPER case keywords.

NAMES'IN'UPPER'CASE

```
NAMES'IN'UPPER'CASE(<true/false>)
NAMES'IN'UPPER'CASE(true)
```

All identifiers (ie, variable names, procedure names,...) are listed in lower case by default. You can have them listed in upper case by issuing the command as shown in the example above.

QUOTE'MODE

```
QUOTE'MODE(<true/false>)
QUOTE'MODE(true)
```

Quote mode is used by Commodore to allow you to insert cursor movements and other special controls inside a string constant. Some people like it. Many hate it. With COMAL you can choose to have it or not. It is disabled by default and the example above illustrates how to turn it back on.

SERIAL

```
SERIAL(<true/false>)
SERIAL(true)
```

Commodore's IEEE interface for the C64 does not allow 'mixed' devices (some on the serial bus and others on IEEE). Thus a serial printer and IEEE disk drive could not both be used by a program at once. COMAL overcomes this drawback with the SERIAL command. Issue the command: SERIAL(true) and all I/O will be on the serial bus. Issue the command: SERIAL(false) and all I/O will be on the IEEE bus. You can change back and forth as often as you need to. Plus, this also allows you to have two printers with device number 4, one on each bus!

SETPAGE

```
SETPAGE(<page number>)
SETPAGE(2)
```

This command is not needed for normal COMAL use. But if you really want to get 'inside' COMAL it comes in handy. Use it to look at the different memory overlays used by COMAL on the C64. Different areas seem to need different overlay numbers. Some areas are:

```
$E000 - $FFFF
  SETPAGE(0) => graphic screen
  SETPAGE(2) => kernal
```

```
$D000 - $DFFF
  SETPAGE(0) => hidden RAM
  SETPAGE(2) => character generator ROM
  SETPAGE(6) => I/O and color RAM
    (when PEEKing color RAM ignore upper 4 bits)

$A000 - $BFFF
  SETPAGE(0) => hidden RAM (used by packages)

$8000 - $9FFF
  SETPAGE(0) => RAM (used by packages)
  SETPAGE(3) => COMAL code
```

SETPRINTER

```
SETPRINTER(<printer specifications$>)
SETPRINTER("u5:/a+/l+/t+/s7/d-")
```

See COMAL HANDBOOK edition 2 APPENDIX N page 441 for details. Briefly:

u5:	sets the unit to device 5
/a+	convert upper/lower case to true ASCII
/l+	a carriage return will also send a line feed
/t+	use IEEE time out conventions used by COMAL
/s7	use secondary address 7
/d-	the file is not a disk file

Whenever a SELECT "LP:" command is encountered, the default printer is opened. SETPRINTER allows you to set up just how the default should be and only needs to be used once, but you are allowed to change it as often as you wish.

SETRECORDDELAY

```
SETRECORDDELAY(<amount delay>)
SETRECORDDELAY(0)
```

This allows you to fine tune the COMAL system to your disk drive. If you are using a Commodore 1541 drive you should not use this command. However, if you are using another drive, such as the MSD Dual Drive, you can get faster disk access by issuing the command:
  SETRECORDDELAY(0).

SETSCREEN

```
SETSCREEN(<string$>)
SETSCREEN(screen1$)
```

This restores a text screen previously stored with a GETSCREEN command. The entire text screen including colors and cursor position is stored as a 1505 character string as follows:

```
  First character is border color of text screen
  Second character is background color of text screen
  Third character is cursor color on text screen
  Fourth character is cursor location, row-1
  Fifth character is cursor location, column-1
  The rest of the string is text and color information
   grouped in sets of 3:
    1: first character
    2: second character
    3: low 4 bits for first character color,
       high 4 bits for second char color
```

Example:

```
  DIM screen$ of 1505
  GETSCREEN(screen$)      save current screen as screen$
  PAGE                    clear screen, ready to test it
  SETSCREEN(screen$)      put screen back again
```

Note: This is useful for HELP menus - save current screen - flip through HELP - then replace original screen.

SETTIME

```
SETTIME(<time string$>)    sets the real time clock
SETTIME("0")
SETTIME("10:30")
SETTIME("5:45:15")
SETTIME("0:0:0.0/50")
```

```
   set time in GETTIME$ format: HH:MM:SS.T
   You may change just the hours like this:
    SETTIME("10")  sets hours to 10
   You set full time (without tenths):
    SETTIME("10:30:0")  sets time to 10:30
```

SETTIME allows you to set the system clock used by COMAL. Time is kept in this format: HH:MM:SS.T with HH the hour, MM the minutes, SS the seconds and T the tenths of a second. To set just the hour it is not necessary to include the rest of the paramenters in the string (the first example sets the hour to 0 and also by default sets minutes, seconds, and tenths to 0).

The second example sets the time to 10:30:00.0. The third example sets the time to 05:45:15.0 and the last example sets the time to 00:00:00.0.

COMAL is designed to work with both 50 and 60 cycle power (European and American). It defaults to the correct type automatically. However, you are allowed to specify the cycle type at the very end of a time setting string - preceded by a slash /. Thus the /50 at the end of the final example would put COMAL time keeping in the 50 cycle mode. This is not practical, but you are allowed the option.

SHOWKEYS

SHOWKEYS

The function keys are active in COMAL. They are set up to perform common functions automatically for you, but you can set them to anything you want with the DEFKEY command. SHOWKEYS will display what each function key is currently set at. The display is done to be compatible with the define key command so that after showing the keys current definitions, you can simply cursor up to the one you wish to change, make the changes in the string, and hit return. The function keys have two settings for each key, one for while a program is running, another while in READY mode (edit mode). Thus a program can set function keys without disturbing the settings outside the running program. To distinguish between them, program mode function keys are internally stored as 10 more than the real key number (ie, F1 would be F11).

Default settings follow:

READY MODE:

```
F1=RENUM + RETURN
F2=MOUNT + RETURN   initializes disk
F3=USE TURTLE + RETURN     switch to turtle graphics
F4=AUTO
F5=EDIT
F6=LIST
F7=RUN             use with DIR listing to RUN from disk
F8=SCAN
```

PROGRAM MODE: (matches their values in BASIC)

```
F11=CHR$(133)
F12=CHR$(137)
F13=CHR$(134)
F14=CHR$(138)
F15=CHR$(135)
F16=CHR$(139)
F17=CHR$(136)
F18=CHR$(140)
```

TEXTCOLORS

```
TEXTCOLORS(<border>,<background>,<cursor>)
TEXTCOLORS(15,1,6)
```

This is used to set all three text colors at once. Colors can be 0-15 plus -1 meaning no change from the current value. Use CONTROL Z to keep the current colors as the default colors.

GRAPHICS PACKAGE

The Commodore 64 graphics screen is made up of 64,000 pixels that can each display one of 16 different colors. The graphics screen has two different modes: Hi-Res and Multi-Color. To access all the commands you must first issue the command: USE GRAPHICS.

Hi-Res: This mode displays 200 pixel lines, each of which is made up of 320 pixels. Due to the structure of the video chip, only one color besides the background color can be used in each 8 X 8 block (these blocks match the character blocks on the text screen). If you try to draw in a block with a new color, then everything drawn in that block is changed to the new color. This mode provides very good high-resolution graphics for smooth lines and drawing.

Multi-Color: This mode displays 200 pixel lines, each made up of 160 pixels. The video chip uses pairs of pixels to tell what color to use. In this mode you have fewer dots per line, but you can have 3 different colors in addition to the background color in each 8 X 8 block. Lines are twice as wide because of the fewer pixels per line, and can appear to be jagged. Once three colors are used in a block, the fourth will change the last drawn color to the new color.

USE GRAPHICS will initialize the graphics screen only if it is not already initialized. After the screen has been initialized, USE GRAPHICS will only add the commands to the name table and set the screen up as follows:

```
    0,199 +---------------+ 319,199
          !               !
          !               !
          !       *       !
          !    160,100    !
          !               !
      0,0 +---------------+ 319,0
```

Drawing on the screen can use a system called "Turtle Graphics". This means you move an imaginary "Turtle", that has a colored pen fastened to it, around the screen with simple commands and, if the pen is down, draw lines. Many of the commands will draw from where the turtle's current position.

Angle degrees on the graphic screen are as follows:

```
          0
      315    45
   270          90
      225   135
         180
```

Straight up is zero degrees, right is 90, and so on.

ARC

```
ARC(<centerX>,<centerY>,<radius>,<start angle>,<size>)
ARC(160,100,50,0,90)
```

This command draws an arc, which is a section of a circle. The first three parameters describe the circle, and the last two, the arc of that circle. The arc will begin on the edge of the circle, starting at <start angle> (0-359). From there it will draw to the left around the circle, for a number of degrees set in <size>. In the example above, an ARC will be drawn on screen starting 50 pixels above the center of the screen, going around to the left outlining a circle 50 pixels in radius, for 1/4 the length of the circle (90/360=1/4). The turtle is moved by this command.

ARCL

```
ARCL(<radius>,<angle size>)
ARCL(50,180)
```

Draws an arc from the current position and HEADING of the turtle counterclockwise for the number of degrees given in <angle size>, using <radius> as the distance from the turtle to the points on the arc. In the example above, half a circle is drawn. If <angle size> is 360, then a full circle is drawn. The turtle is moved by this command.

ARCR

```
ARCR(<radius>,<angle size>)
ARCR(75,10)
```

This command works like ARCL except it draws to the clockwise instead of counterclockwise. The turtle is moved by this command.

BACK

```
BACK(<distance>)
BACK(25)
```

Moves the turtle backwards <distance> units. If the pen is down, a line will be drawn.

BACKGROUND

```
BACKGROUND(<color number>)
BACKGROUND(0)
```

This command sets the background color of the graphics screen. This command will immediatly change the background color of the entire graphics screen (unlike 0.14) and will not work on the text screen. To change the color of the text screen you must now use TEXTBACKGROUND. The colors 0-15 are the only allowable values for <color number>.

BORDER

```
BORDER(<color number>)
BORDER(-1)
```

This command sets the border color of the graphics screen. <color number> can be in the range of -1 to 15. If the number is -1 (as in the above example) the border color of the graphics screen will be the same as the BACKGROUND color. To change the border to the text screen you must now use TEXTBORDER.

CIRCLE

```
CIRCLE(<center X>,<center Y>,<radius>)
CIRCLE(160,100,50)
```

This will draw a circle around the center point with the radius given. The circle is drawn counterclockwise for a full 360 degrees. The turtle is moved by this command.

CLEAR

```
CLEAR
```

This command will clear the part of the screen inside the current VIEWPORT. You can use these two commands to clear any portion of the screen.

CLEARSCREEN

```
CLEARSCREEN
```

This will clear the entire graphics screen, regardless of the VIEWPORT command. It is a good idea to use this command to clear the screen in procedures rather than the CLEAR command, in case some other part of the program does a VIEWPORT command.

DRAW

```
DRAW(<X offset>,<Y offset>)
DRAW(5,-4)
```

This command will draw to a new position set by adding <X offset> to the current X position of the turtle and adding <Y offset> to the current Y position of the turtle. In the example above, the turtle will move to a point 4 units below and 5 units to the right of the current position.

DRAWTO

```
DRAWTO(<X position>,<Y position>)
DRAWTO(100,50)
```

This command will draw a continuous line to the given position regardless if the pen is up or down. The line is drawn in the color set by pencolor.

FILL

```
FILL(<X position>,<Y position>)
FILL(50,50)
```

This command will start at the specified position and fill from that point in the color set by PENCOLOR until a non-background color or the edge of the viewport is encountered. The color of pixel <X position>,<Y position> is considered the background color for this command.

FORWARD

```
FORWARD(<distance>)
FORWARD(10)
```

This command will move the turtle <distance> units forward in the direction it is pointing. If PENDOWN is in effect, a line will be drawn in the color set by PENCOLOR.

FULLSCREEN

```
FULLSCREEN
```

This command will show the full graphics screen. If the text screen is active before this command, then the screen will switch to the graphics screen. If SPLITSCREEN is in effect, the upper four lines will remain as they are. You will have to clear them with the VIEWPORT and CLEAR commands. This mode can setup using the f5 key in direct command mode.

GETCOLOR

```
GETCOLOR(<X coordinate>,<Y coordinate>)
dot'color:=GETCOLOR(160,100)
```

This function returns the color of the specified pixel or -1 if it is a background pixel.

GRAPHICSCREEN

```
GRAPHICSCREEN(<screen type>)
GRAPHICSCREEN(0)
```

This command initializes the graphics screen to either of two types:

```
     Multi-Color = 1
     Hi-Res      = 0
```

The graphics screen is cleared and the graphics screen will be shown.

HEADING

```
HEADING
turtle'heading:=HEADING
```

This function returns the current direction, in degrees, that the turtle is pointing.

HOME

```
HOME
```

This command moves the turtle to the point 0,0 (which is by default the lower left hand corner for USE GRAPHICS and in the center of the screen for USE TURTLE) and points the turtle straight up, SETHEADING(0). If the screen boundries have been changed with WINDOW command, then the turtle will move the point 0,0 -- even if that point is off the screen.

INQ

```
INQ(<type function number>)
num:=INQ(0)
```

This is a multi-purpose function that returns information about the graphics screen and the turtle. This information would be most useful in CLOSED procedures that do graphics and return the turtle and screen to the original state. The parameter determines what information is returned.

NUMBER	FUNCTION TYPE
0	Graphics screen type
1	Text border color
2	Text background color
3	Text cursor color
4	Graphics border color
5	Graphics background color
6	PENCOLOR setting
7	Textstyle Height size
8	Textstyle Width size
9	Textstyle Direction
10	Textstyle Overplot?
11	Turtle visable?
12	Turtle inside VIEWPORT?
13	Textscreen currently displayed
14	Splitscreen active?
15	Wrap mode On?
16	PENDOWN?
17	X-coordinate of turtle
18	Y-coordinate of turtle
19	VIEWPORT X minimum
20	VIEWPORT X maximum
21	VIEWPORT Y minimum
22	VIEWPORT Y maximum
23	WINDOW X minimum
24	WINDOW X maximum
25	WINDOW Y minimum
26	WINDOW Y maximum
27	Cosine of HEADING
28	Sine of HEADING
29	Size of Turtle set by TURTLESIZE

In the example above, the numeric variable (num) is set to the graphic type (0 for Hi-Res, 1 for Multi-Color).

LEFT

```
LEFT(<distance>)
LEFT(45)
```

Turns the turtle to the counterclockwise the number of degrees indicated. By using negative numbers, the turtle is turned the other way.

LOADSCREEN

```
LOADSCREEN(<filename>)
LOADSCREEN("hrg.screen'name")
```

This command will load a color graphics screen from the file <filename>, which was created with SAVESCREEN. To identify graphic screen files, you should precede the file name with "hrg.".

MOVE

```
MOVE(<X offset>,<Y offset>)
MOVE(5,-5)
```

This command works like DRAW, but it just moves the turtle. It will not draw a line even if the pen is down. The offsets are added to the position of the turtle just as for DRAW.

MOVETO

```
MOVETO(<X coordinate>,<Y coordinate>)
MOVETO(100,50)
```

Moves the turtle to the new position without drawing a line.

NOWRAP

```
NOWRAP
```

When the turtle moves off the screen, or VIEWPORT, it continues to move, but does not draw. This is the default for USE GRAPHICS.

PAINT

```
PAINT(<X coordinate>,<Y coordinate>)
PAINT(160,100)
```

This command works like FILL, but it fills with the current pencolor until it meets a pixel of the current color or the edge of the VIEWPORT setting. You can set PENCOLOR to -1 (background), draw the outline, and PAINT the shape to erase it.

PENCOLOR

```
PENCOLOR(<color number>)
PENCOLOR(1)
```

Sets the drawing color. The color numbers range from 0 to 15, plus the number -1 means 'erase' color: then you will erase rather than draw.

PENDOWN

```
PENDOWN
```

This command permits turtle drawing commands to draw lines. This is the default for USE GRAPHICS or USE TURTLE.

PENUP

```
PENUP
```

After this command, commands which are dependent on the Penstate (up/down) will not draw.

PLOT

```
PLOT(<X coordinate>,<Y coordinate>)
PLOT(100,50)
```

This command plots the given point in the current PENCOLOR. The turtle is not moved by this command.

PLOTTEXT

```
PLOTTEXT(<X coordinate>,<Y coordinate>,<text$>)
PLOTTEXT(10,10,"This is the text to print")
```

This plots text on the screen as defined by TEXTSTYLE. Text can be put anywhere on the screen, and it is plotted in the current color. Since the text can be stretched, you can now plot text on the Multi-Color screen. The starting point is the lower left hand corner point of the string's first character.

PRINTSCREEN

```
PRINTSCREEN(<filename>,<offset>)
PRINTSCREEN("lp:",80)
```

This command outputs the graphics screen to the printer (or the disk drive for later printing). It is a smart routine in that it skips over blank areas (resulting in a faster print) and can print Multi-Color pictures (using gray levels). This is only for MPS-801 compatible printers. <offset> is how far from the left the picture should be. The example above would print the picture in the center of the paper on a Commodore MPS-801 printer.

RIGHT

```
RIGHT(<angle>)
RIGHT(45)
```

This command turns the turtle clockwise the number of degrees specified. See also LEFT.

SAVESCREEN

```
SAVESCREEN(<filename>)
SAVESCREEN("hrg.screen'name")
```

This command saves the graphic screen to the disk drive using the name in <filename>. To identify screens saved to disk you should precede the file name with "hrg.". The screen can be retrieved later by LOADSCREEN.

SETHEADING

```
SETHEADING(<angle>)
SETHEADING(90)
```

Sets the HEADING of the turtle to the angle specified. In the example above, the turtle would be pointing to the right.

SETXY

```
SETXY(<X coordinate>,<Y coordinate>)
SETXY(100,50)
```

This command moves the turtle to the point specified, and draws a line if PENDOWN is in effect.

SHOWTURTLE

```
SHOWTURTLE
```

This command turns the turtle image on (the image is a triangle with a line which points the direction the turtle is pointing). The turtle is the color of PENCOLOR.

SPLITSCREEN

```
SPLITSCREEN
```

If the current graphic mode is Hi-Res, then the screen will show the graphic screen, with the top four lines displaying the text screen. The text screen is drawn onto the graphic screen, so if FULLSCREEN is done afterwards, the text remains. This mode can also be enabled in direct command mode by pressing the f3 key.

TEXTBACKGROUND

```
TEXTBACKGROUND(<color number>)
TEXTBACKGROUND(0)
```

This command sets the background color for the text screen. The above example sets the background for the text screen to black.

TEXTBORDER

```
TEXTBORDER(<color number>)
TEXTBORDER(-1)
```

This command sets the border color for the text screen. If color number -1 is selected (as above), then the color is the TEXTBACKGROUND color.

TEXTCOLOR

```
TEXTCOLOR(<color number>)
TEXTCOLOR(1)
```

This works like PENCOLOR, but it is for the color text is displayed on the text screen (not PLOTTEXT). You can use -1 for the color number to print something on the text screen that can not be read by the user (invisible ink), but is accepted by a program (INPUTing from the screen -- maybe for SELECT INPUT "ds:")

TEXTSCREEN

```
TEXTSCREEN
```

This switches the screen to the text screen. To switch back to the graphic screen, you must use the commands FULLSCREEN or SPLITSCREEN. These commands can be called by pressing the f3 or f5 key in direct command mode.

TEXTSTYLE

```
TEXTSTYLE(<height>,<width>,<direction>,<overplot?>)
TEXTSTYLE(1,1,0,1)
```

This command determines how text is ploted on the graphics screen. The first two parameters determine the size of the characters. This is useful for large titles and for plotting text on the Multi-Color screen. Since the Multi-Color screen uses pairs of pixels, COMAL automatically makes sure that the width multiple is an even number: 2,4,6,...,etc. The default is 1,1.

<direction> refers to the direction the text should be printed.

NUMBER	DIRECTION
-1	No change
0	To the right
1	Upwards (rotated 90 degrees to the left)
2	Upside down (to the left)
3	Downwards (rotated 90 degrees to the right)

<overplot?> refers to whether the text should be blended into the graphics, or to simply write over and wipe it out (like it does in 0.14).

NUMBER	STYLE
-1	No change
0	Blend text into graphics
1	Overwrite text onto graphics screen

The example above would print normal sized text on the graphics screen to the right and would overwrite anything on the screen.

TURTLESIZE

```
TURTLESIZE(<size>)
TURTLESIZE(5)
```

Sets the size of the turtle. The turtle can be set to 11 different sizes from 0 (the smallest) to 10 (the largest, and default).

VIEWPORT

```
VIEWPORT(<X minimum>,<X max>,<Y minimum>,<Y max>)
VIEWPORT(0,319,0,199)
```

Sets up a frame, outside of which no drawing will occur. The points specified are not relative and can not be changed or modified by WINDOW. If WRAP is in effect, any drawing which goes off one edge, comes in on the other side.

WINDOW

```
WINDOW(<X minimum>,<X maximum>,<Y minimum>,<Y maximum>)
WINDOW(0,1,0,1)
```

This command sets the scale of the screen (or VIEWPORT) for drawing or positioning sprites. In the above example, points on the screen would be fractional numbers between 0 and 1. One use of this command is to change the scale of the screen so a round CIRCLE would be generated (instead of the usual oval).

WRAP

```
WRAP
```

This command sets up the screen so when you draw off one edge of the screen, or VIEWPORT, you will come in from the other edge (WRAPping around the screen). This is the default mode for USE TURTLE.

XCOR

```
XCOR
num:=XCOR
```

This function returns the X coordinate position of the turtle.

YCOR

```
YCOR
num:=YCOR
```

This function returns the Y coordinate position of the turtle.

TURTLE PACKAGE

This package has all of the commands from the GRAPHICS package, with additions and certain changes. To use this package you must issue the command: USE TURTLE.

The most obvious change is with the screen coordinates. The default range for the X axis is -160 to 159, and the range for the Y axis is -100 to 99. In this case HOME (0,0) is in the center of the screen. Also, if USE TURTLE is entered in direct command mode, the screen will switch to the initialized screen with SPLITSCREEN in effect.

Another change is that WRAP is the default mode. This means if you draw off the screen (or VIEWPORT) you will come back one the screen from the opposite side.

The abbreviations work the same way the regular commands, the names are just shorter.

ABBREVIATION	COMMAND
BG	BACKGROUND
BK	BACK
CS	CLEARSCREEN
FD	FORWARD
HT	HIDETURTLE
LT	LEFT
PC	PENCOLOR
PD	PENDOWN
PU	PENUP
RT	RIGHT
SETH	SETHEADING
ST	SHOWTURTLE
TEXTBG	TEXTBACKGROUND

The abbreviated commands require the same type of parameters as the longer versions of the commands. Both of the following mean the same thing:

```
FORWARD(50)
  and
FD(50)
```

SPRITE PACKAGE

Sprites are video images that can be moved around the screen quickly and easily. The are eight sprites, each of them are 24 pixels wide and 21 pixels high. A 64 byte string defines the sprite image in the following manner:

```
    char  1     char  2     char  3
    char  4     char  5     char  6
    char  7     char  8     char  9
    ....        ....        ....
    ....        ....        ....
    char 61     char 62     char 63
```

The 64th character tells whether the sprite definition is a Hi-Res or Multi-Color sprite. Each character is eight pixels wide (3 X 8 = 24). Whether or not a pixel is on determines what color will be shown.

Hi-Res pixels have only two states, on or off. If the pixel is on then a dot the color of the sprite is turned on. If the pixel is off, whatever is on the graphics screen is shown (also called transparent). Binary numbers can be used in data statements to show a sprite definition:

```
    data %00011000,%00000000,%00000000
    data %00111100,%00000000,%00000000
    data %01100110,%00000000,%00000000
    data %01100110,%00000000,%00000000
    data %01100110,%00000000,%00000000
    data %00111100,%00000000,%00000000
    data %00011000,%00000000,%00000000
    data %00000000,%00000000,%00000000
    ....
    <for a total of 21 lines>
```

Multi-Color sprites are more complex in that they can show three different colors in addition to the fourth transparent. It can do this by using pairs of pixels to choose one of four states. By using this mode the resolution of the sprite becomes 12 X 21 (the dots are streched in size so it will appear the same size as a Hi-Res sprite). The different states (in binary) are:

00	Transparent
01	Multi-Color #1
10	Sprite Color
11	Multi-Color #2

Before you can use any of these commands you must first issue the command: USE SPRITES.

DATACOLLISION

```
DATACOLLISION(<sprite number>,<reset flag?>)
dat'col:=DATACOLLISION(3,FALSE)
```

When a nontransparent part of a sprite overlaps something drawn on the screen, a datacollision occurs and the flag for that sprite is turned on. The status of that flag is returned by this function. The first parameter is the sprite number to check. If the second parameter is true the system clears the datacollision flag for that sprite (making way for a new data collision). If the expression is false then the flag will not be reset (so the status can be checked again). In the above example, the variable dat'col is set to the current status of the data collision flag for sprite number 3, FALSE means that the flag will not be reset.

NOTE: Multi-Color #1 does not cause a collision, so make sure that anything you want visible on the sprite, but not to cause a collision, should be in Multi-Color #1.

DEFINE

```
DEFINE(<image number>,<image string>)
DEFINE(5,sprite'def$)
```

32 different sprite images can be defined in memory at one time (numbered 0-31). The image string is 64 bytes long with the last character telling whether it is Hi-Res or Multi-Color. If the last character value is zero then the sprite is displayed in Hi-Res, otherwise it will be displayed in Multi-Color mode.

The color of a sprite is not contained in the image string -- you must assign it in your program.

HIDESPRITE

```
HIDESPRITE(<sprite number>)
HIDESPRITE(3)
```

This command turns the specified sprite image (0-7) off. It does not change the color of the sprite or its position (although no collision can occur when it is not visible).

The sprite can be turned on again with the SHOWSPRITE command.

IDENTIFY

```
IDENTIFY(<sprite number>,<image number>)
IDENTIFY(0,1)
```

Once a sprite image has been defined with the DEFINE command, any sprite can be displayed with that image. The IDENTIFY command tells a sprite which image definition to use as its shape. It does not turn on the image (SHOWSPRITE does that). In the above example sprite number 0 is set to image number 1 (if sprite number 0 had been on before this command, then it will remain on, but showing the new image).

LINKSHAPE

```
LINKSHAPE(<image number>)
LINKSHAPE(1)
```

This command allows you to 'link' a sprite image, or shape, to the program in memory so when the program is saved and/or loaded the sprite shape will go with it. Once this has been done it is not necessary for the program to DEFINE it. Thus you can include a sprite definition procedure in your program, RUN it, then LINKSHAPE all the images to the program. Then you can delete the definiton procedue as the shapes are available at all times since they then are attached to the proram. Any or all of the 32 different sprite images can be linked to your program. To get rid of the image you can list your program to disk (and ENTER it back in) or use the command DISCARD.

LOADSHAPE

```
LOADSHAPE(<image number>,<file name>)
LOADSHAPE(3,"0:shap.queen")
```

Sprite Images can be saved on disk as a 64 byte string with the SAVESHAPE command. The image number that the shape was in originally is not saved in the file, so you can load the image into any of the 32 areas (0-31) with the LOADSHAPE command. The example above loads a shape from the file "shap.queen" into image number 3.

Shape file names should be preceded by "shap." so that it is obvious what the file contains.

PRIORITY

```
PRIORITY(<sprite number>,<data priority?>)
PRIORITY(2,TRUE)
```

Normally, a sprite moves across the screen passing over (or in front of) anything drawn on the screen. Using PRIORITY, you tell the video chip whether the given sprite should pass in front of, or behind, anything drawn on the graphicscreen. if <data priority> is true, then the sprite will pass behind graphics drawn on the screen.

Sprites have their own, fixed, priority system - in addition to the data priority. A lower numbered sprite is displayed in front of a higher numbered sprite. For example, sprite 3 will pass in front of sprite 4.

SAVESHAPE

```
SAVESHAPE(<image number>,<file name>)
SAVESHAPE(6,"shap.boat")
```

This command saves a image, or shape, to disk. The file is a Sequential file 64 bytes long that consists of the 64 byte definition of the image. The example above writes image number 6 to the disk drive with the file name "shap.boat".

Shape files should have their name preceded by "shap." so they can be easily identified.

SHOWSPRITE

```
SHOWSPRITE(<sprite number>)
SHOWSPRITE(1)
```

This command turns a sprite on. After you define the sprite image with the DEFINE command, and tell the computer what image the sprite should use, you must issue this command to display the sprite. To turn the image off, use the HIDESPRITE command.

SPRITEBACK

```
SPRITEBACK(<multi-color #1>,<multi-color #2>)
SPRITEBACK(1,7)
```

If a sprite has been defined as Multi-Color, then it will use the colors specified with the SPRITEBACK command. The Multi-Colors defined by the command are common to all the eight sprites. The example above sets Multi-Color #1 to white, and Multi-Color #2 to yellow.

SPRITECOLLISION

```
SPRITECOLLISION(<sprite number>,<reset flag?>)
sprite'col:=SPRITECOLLISION(2,TRUE)
```

This command works like the DATACOLLISION command, but it detects whether the specified sprite has collided with another sprite. When two sprites overlap, both flags (one for each sprite) are turned on. The <reset flag> parameter works the same way as it does in DATACOLLISION (see above).

SPRITECOLOR

```
SPRITECOLOR(<sprite number>,<color>)
SPRITECOLOR(3,7)
```

All eight sprites have a unique sprite color register. This command sets the color (0-15) for that sprite.

While SPRITEBACK sets colors common to all sprites, SPRITECOLOR sets the color only for the specified sprite.

SPRITEINQ

```
SPRITEINQ(<sprite number>,<item number>)
this'image:=SPRITEINQ(2,8)
```

This is a general purpose function that returns information on the status of the sprites. There are 11 different functions defined by SPRITEINQ:

ITEM NUMBER	FUNCTION
0	Is sprite visible?
1	Color number of Multi-Color #1
2	Color number of Sprite
3	Color number of Multi-Color #2
4	Expanded in width?
5	Expanded in height?
6	Multi-Color or Hi-Res sprite (64th byte)
7	Data priority
8	Image number
9	Is sprite moving?
10	Sprite-to-sprite collision?
11	Sprite-to-data collision?

This function can by used by general purpose procedures that needs certain information on sprites. The example above sets this'image to the image that was defined in a previous IDENTIFY statement.

SPRITEPOS

```
SPRITEPOS(<sprite number>,<x coordinate>,<y coordinate>)
SPRITEPOS(1,160,100)
```

This command will position a sprite on the graphic screen. Where the sprite is positioned depends on the x coordinate/y coordinate and the WINDOW setting. The reference position is for the upper left hand corner of the sprite.

SPRITESIZE

```
SPRITESIZE(<sprite#>,<expand width?>,<expand height?>)
SPRITESIZE(3,TRUE,FALSE)
```

Any or all of the eight sprites can be exapanded to double their width and/or height. If a sprite is expanded in the X direction, it is expanded to the right. If a sprite is epanded in the Y direction, it is expanded downward. In the above example, sprite number 3 would be double width, but normal height.

SPRITEX

```
SPRITEX(<sprite number>)
x'pos:=SPRITEX(3)
```

This function returns the current X position of a sprite. In the above example x'pos is set to the X coordinate of sprite number 3.

SPRITEY

```
SPRITEY(<sprite number>)
y'pos:=SPRITEY(3)
```

This function returns the current Y position of a sprite. In the above example y'pos is set to the Y coordinate of sprite number 3.

STAMPSPRITE

```
STAMPSPRITE(<sprite number>)
STAMPSPRITE(5)
```

This powerful command stamps the sprite image onto the graphics screen. This is useful for games where you have more than eight images, or to print a screen (stamp all eight sprites so they will show up on a screen dump). Care should be taken in stamping a Multi-Color sprite on a Hi-Res screen or the other way around.

AUTOMATIC SPRITE CONTROL

Almost all games on the Commodore 64 have been written to take advantage of the sprite capability. This can also be done with the standard sprite commands, but timing is usually important, so writing a program to do sprite animation is difficult. Most game programing, therefor, is done in machine code so speed would not be a problem. While machine code is fast, it is not easy to write. The following COMAL commands allow you to do sprite animation at machine code speed.

Sprite animation is like drawing a cartoon. You present a series of drawings that fools the eye into seeing continuous motion.

The animating and moving of sprites presented in this package take place in what is known as "background" mode. This means that the sprites can be animated and moving around the screen while your program continues executing other things. This is similar to the automatic sound system.

ANIMATE

```
ANIMATE(<sprite number>,<sequence$>)
ANIMATE(3,sprite'sequence$)
```

ANIMATE sets up a sequence of commands that can be thought of as a "mini" sprite language. The commands are in pairs, with the first being a command, and the second its parameter. Once started, it runs independently of COMAL. All the commands and their parameters are one byte, or character, long. The commands are:

ANIMATE: IMAGE & DURATION

```
chr$(<image number>)+chr$(<duration>)
Example:    chr$(3)+chr$(30)
     or:    ""3""30""
```

This has the sprite number in the ANIMATE command display the image contained in <image number> for a length of <duration> 60ths of a second. In the above example, the sprite specified in the ANIMATE command would show image number 3 for 1/2 a second (30/60). After the length of <duration> is over, the next command is executed. The duration can vary between 0 and 255 (60ths of a second). If <duration> is zero then the sprite will stop and wait for a "Go" command from another sprite, or another ANIMATE command for this sprite.

ANIMATE: SPRITE COLOR

```
"c"+chr$(<color number>)
Example:    "c"+chr$(1)
     or:    "c"1""
```

This sets the color for the sprite specified in ANIMATE to the color number. In the example above, the sprite color is changed to white.

ANIMATE: GO

```
"g"+chr$(<sprite number>)
Example:    "g"+chr$(4)
     or:    "g"4""
```

Tells sprite <sprite number> to Go. If you give a duration of zero in the first command, it will stop until told to Go. With this you can have a sprite controlled by another sprite. The above example tells sprite number 4 to Go.

ANIMATE: HIDE A SPRITE

```
"h"+chr$(<sprite number>)
Example:    "h"+chr$(3)
     or:    "h"3""
```

Does a HIDESPRITE of the given sprite. The sprite will continue to move and animate, whether it is on or off. When off, no collision can occur.

ANIMATE: PAUSE

```
"p"+chr$(<duration>)
Example:    "p"+chr$(60)
     or:    "p"60""
```

Pauses the sprite given in the ANIMATE command for a length of <duration> in 60ths of a second. The example above will pause the sprite for 1 second (60/60).

ANIMATE: SHOW A SPRITE

```
"s"+chr$(<sprite number>)
Example:     "s"+chr$(2)
     or:     "s"2""
```

Does a SHOWSPRITE of the given sprite. The sprite will continue to move.

ANIMATE: EXPAND WIDTH

```
"x"+chr$(<expand width?>)
Example:    "x"+chr$(TRUE)
     or:    "x"1""
```

Does a SPRITESIZE for the width of the sprite given in ANIMATE. The parameter should be either 0 or 1.

ANIMATE: EXPAND HEIGHT

```
"y"+chr$(<expand height?>)
Example:    "y"+chr$(FALSE)
     or:    "y"0""
```

Does a SPRITESIZE for the height of the sprite given in ANIMATE. The parameter should be either 0 or 1.

ANIMATE: HALT

To stop animation of a sprite, issue a new ANIMATE command with a null string. If an error in the 'syntax' occurs, animation is aborted, and an error generated.

MOVESPRITE

```
MOVESPRITE(<sprite#>,<new x>,<new y>,<speed>,<event>)
MOVESPRITE(2,320,50,400,%00000110)
```

This command will move a sprite from the present position to the new destination at the specified speed.

<speed> is a number from 0-32767 which means how long it takes (in 60ths of a second) to move from the original position to the new destination. A speed of zero means to move there immediatly.

<event> is a number that tell when to start moving the sprite, and if the sprite should stop if it collides with another sprite or with screen data. The number is in the range of 0-7 and is a binary number. The settings are:

NUMBER	SETTING
%0000000X	1=Move Now 0=Wait for STARTSPRITES command
%000000X0	1=Stop if SPRITECOLLISION 0=Continue regardless
%00000X00	1=Stop if DATACOLLISION 0=Continue regardless

The upper 5 bits of <event> are ignored. For example: %00000111 means move now and stop if it collides with either data or another sprite.

MOVING

```
MOVING(<sprite number>)
spr'moving:=MOVING(3)
```

This is a function that returns TRUE if the given sprite is moving (using MOVESPRITE). It will return FALSE if it is not moving, or if no MOVESPRITE has been used.

STARTSPRITES

```
STARTSPRITES
```

This command will start any sprites moving that are waiting because of the setting in a MOVESPRITE command.

STOPSPRITE

```
STOPSPRITE(<sprite number>)
STOPSPRITE(3)
```

Stops the sprite from moving or animating.

See The Programer Reference Guide for more information on the structure of sprites.

SOUND PACKAGE

The Commodore 64 has the most advanced music synthesizer built into any 8 bit personal computer today. Unfortunatly, that also makes it one of the hardest to use. This package will give you complete control over the 64's sound capabilities. It even provides a way to play music independent of your running program. The commands will be broken up into two parts (standard commands and the automatic independent commands) even though you can use both at the same time. As all other packages, before you can use any of the sound commands you must first issue the command:

USE SOUND

This adds the commands to COMAL and initializes the SID chip so that voice 1 sounds like a piano, voice 2 sounds like a violin, and voice 3 sounds like cymbals. Also, the volume is turned on full and the filters are turned off.

<voice number> is a numeric expression equal to 1, 2, or 3 that refers to any of the three available voices.

SETFREQUENCY

```
SETFREQUENCY(<voice number>,<frequency value>)
SETFREQUENCY(1,4291)
```

This command sets the frequency, or pitch, of a given voice. Frequency is how fast the voice vibrates. The higher the frequency, the higher pitched the sound is. Middle C on a piano is 4291. The range <frequency value> is 0-65535.

The example above sets the frequency of voice one to a middle C.

NOTE

```
NOTE(<voice number>,<note$>)
NOTE(1,"c4")
```

Figures the frequency of note$ and sets the given voice to that frequency. Note$ is a two or three character string containing:

The first character is the note name: a,b,c,d,e,f,g
(must be lower case letters)

The second character is the octave number:
0,1,2,3,4,5,6,7

The optional third character is for the sharp indicator.

```
For example, "c4"  means middle C
             "d2#" means D sharp in the second octave
```

Note: The frequency range for the SID chip is the same for a piano, but octave 0 is lower than a piano, and the SID chip cannot go as high as a piano.

The original example sets up the frequency of voice one to a middle C.

ADSR

```
ADSR(<voice#>,<attack>,<decay>,<sustain>,<release>)
ADSR(1,8,8,10,8)
```

ADSR stands for Attack, Decay, Sustain and Release.
ADSR controls the volume of a note while it is playing.

The process starts when the gate is turned on.
Immediately the volume begins to climb to the maximum volume level. The amount of time it takes to get to that level is determined by the attack rate.
Once maximum volume is reached, the volume begins to fall to the sustain volume. The amount of time it takes is set by the decay rate. The note will stay constant at the sustain level until the gate is turned off.

After the gate is turned off the volume will begin to fall to zero at the release rate. This may sound complicated (and it is), but before too long it will seem all to simple.

```
  Default setting for voice 1: ADSR(1,0,12,10)
  Default setting for voice 2: ADSR(2,10,8,10,9)
  Default setting for voice 3: ADSR(3,0,9,0,9)
```

SOUNDTYPE

```
SOUNDTYPE(<voice number>,<waveform>)
SOUNDTYPE(1,2)
```

When a note vibrates, it produces harmonics, which are integer multiples of the base frequency. The overall pitch of the note is the base frequency, with the harmonics anded in. How they are added in is determined by the waveform. For the sake of this discussion we will call the base, or fundemental, frequency harmonic number one (harmonic number 2 is double the frequency of the base, harmonic 3 is thre times the base, etc.). The commodore 64 has 4 different waveforms built in; Triangle, Sawtooth, Pulse, and Noise.

Triangle: This waveform conains only odd harmonics added together by the reciprocal of the square of their harmonic number (1/1 + 1/9 + 1/25 + 1/49 + ...). In other words, harmonic number 3 is 1/9 as quiet as harmonic 1 because 3 squared is 9 and the reciprocal of 9 is 1/9. This type of waveform produces fairly smooth, somewhat hollow, notes, good for organ or flute sounds.

Sawtooth: Contains all harmonics added together by the reciprocal of their base (i.e. 1/1 + 1/2 + 1/3 + 1/4 + ...). This is good for brass and other bright instruments. It is the default for voices 1 and 2.

Pulse: Contains only odd harmonics added together by the reciprocal of their base (1/1 + 1/3 + 1/5 + 1/7 + ...). Furthermore, you can vary the width of the pulse to give an even wider range of control over the sound. This waveform is good for piano and pinging sounds. It is the default for voice 3.

Noise: This waveform produces a hissing type sound produced randomly, commonly referred to as "white noise". It is good for drums and game sounds.

Values for waveforms:

```
 SOUNDTYPE(<voice number>,0)=No sound
 SOUNDTYPE(<voice number>,1)=Triangle
 SOUNDTYPE(<voice number>,2)=Sawtooth
 SOUNDTYPE(<voice number>,3)=Pulse
 SOUNDTYPE(<voice number>,4)=Noise
```

PULSE

```
PULSE(<voice number>,<pulse width 0-4095>)
PULSE(1,1500)
```

If the pulse waveform has been selected, the pulse width controls how it sounds. The closer to the middle width number (2048) the fuller the note sounds. The further away from the middle number the pulse width is, the thinner the note sounds. A reed flute, for instance, produces a very thin, insubstantial, note- so a number far away from the middle would be selected (100-500 or 3600-4095). Low and high numbers are the same to your ear. Only how far from the middle is important.

The original example sets PULSE width for voice one to 1500.

GATE

```
GATE(<voice number>,<on/off>)
GATE(1,TRUE)
```

The GATE command controls the playing of notes. When the gate is turned on the volume of that voice begins to rise at the attack rate until the maximum volume is reached. Once maximum volume is reached the volume for that note begins to fall to the sustain level at the decay rate. When the sustain volume level is reached, the note will remain constant until the gate is turned off. After the gate is turned off the volume for the note will fall to zero at the release rate.

```
  TRUE  = ON   -  Attack, Decay, Sustain
  FALSE = OFF  -  Release
```

In the original example, the GATE for voice one is turned on.

RINGMOD

```
RINGMOD(<voice number>,<on/off>)
RINGMOD(TRUE)
```

If the waveform for <voice number> is set to Triangle, then that voice will be "ring modulated" with another voice. This can be used for bell, or gong sounds. The voices are set to be "ring modulated" in a specific way. If voice one is set then it will be "ring modulated" with voice three. Likewise, voice three with voice two and voice two with voice one.

```
TRUE  = On
FALSE = Off
```

The original example turns the ring modulation for voice number one on.

SYNC

```
SYNC(<voice number>,<on/off>)
SYNC(1,0)
```

The SYNC command will work with any waveform and synchronizes the base frequency of one voice with another. This results in a complex series of harmonics. The frequency of the voice being synchronized with should be lower (but not zero) than the frequency of <voice number>. Voice one will synchronize with voice three, voice two with voice one, and voice three with voice two.

```
TRUE  = On
FALSE = Off
```

In the original example, synchronizing for voice number one is turned off.

VOLUME

```
VOLUME(<level>)
VOLUME(15)
```

This command will set the master (or maximum) volume for all the voices. The range is zero (off) to 15 (maximum).

The example above turns the volume on full. This value is the default for USE SOUND.

FILTERS

FILTERTYPE

```
FILTERTYPE(<low>,<band>,<high>,<voice 3 off?>)
FILTERTYPE(TRUE,FALSE,FALSE,TRUE)
```

The commodore 64 has a built in filter which acts like the tone controls on your stero (but a bit more flexible). First you set the filter's frequency, then you specify the filter type. The different types are Low pass, Band pass, and High pass.

Low Pass: When this is turned on, all frequencies at or below the filter frequency is pass unchanged. Frequencies above the filter frequency are cut out more and more the further above the filter frequency they are. This way you get the deeper tones.

Band Pass: This filter passes freqencies at (or around) the filter frequency and cuts out the frequency above and below. You could eliminate harmonics this way.

High Pass: This filter works like the Low Pass filter, but frequencies at or above the filter frequency are passed. This way you get the higher tones.

These filters are additive and you could select them together. If you turned both the Low pass and the High pass filter on, only frequencies at the filter freqency would be cut out.

<voice three off?>: This allows you to turn off voice three at its output, so you can use it to modify the other two voices.

```
  TRUE  = On
  FALSE = Off
```

The original example sets the filter to Low Pass and turns voice 3 off.

FILTERFREQ

```
FILTERFREQ(<frequency 0-2047>)
FILTERFREQ(729)
```

This command will set the filter frequency. The steps are 5.85 Hz apiece so a middle C is a value of 729. The effective range is 30 Hz to 12 kHz.

The example above sets the filter frequency to a middle C (aproximately).

FILTER

```
FILTER(<voice one>,<voice two>,<voice three>,<external>)
FILTER(TRUE,FALSE,TRUE,FALSE)
```

This command specifies what voices should go through the filter. There is an external input to the sound chip and if you use it, the sounds from it can be modified by the filter.

```
  TRUE  = On
  FALSE = Off
```

In the example above voice one and three will go through the filter while voice two and the external input will not.

RESONANCE

```
RESONANCE(<value 0-15>)
RESONANCE(15)
```

RESONANCE is a peaking effect of freqencies at the filter frequency. The values range from zero (no effect) to 15 (full effect). This will produce sharper sounds.

The example above turns resonance on full for the filter.

VOICE MODIFING

It is possible to modify notes while they are playing. For instance, if you rapidly change the frequency of a note around a small value you would get a vibrato effect. Many different effects can be generated this way (try selecting the Pulse waveform and sliding the pulse width). To help in this, two commands allow you to see what is happening in voice three.

OSC3

```
OSC3
wave'value:=OSC3
```

This function returns the value (0-255) of oscillator 3. The numbers that appear depend on the waveform (although they all change at the FREQUENCY rate).

Triangle will generate a series of numbers going from 0 to 255 and back down to 0.

Sawtooth will generate a series of numbers going from 0 to 255.

Pulse will jump back and forth between 0 and 255.

Noise will generate random numbers.

The example above sets the variable wave'value to the output of oscillator 3.

ENV3

```
ENV3
wave'point:=ENV3
```

This function return the value (0-255) of envelope 3. The numbers returned reflect the ADSR values and how far into the note it is. Voice 3 must be gated in order to use this function.

The example above sets the variable wave'point equal the the value of envelope 3.

AUTOMATIC SOUND CONTROL
(INTERRUPT DRIVEN SOUND)

Every sixtith of a second, an interrupt is generated by the Commodore 64 (for the keyboard, clock, and other system demands). COMAL 2.0 takes advantage of this so you can play music while you do graphics, sprites, or any other programing simultaneously. The only restriction is you cannot Load or Chain other programs, or modify the program in memory. This automatic system is setup by SETSCORE, started by PLAYSCORE, and stopped by STOPPLAY. Waitscore is a built in function that can tell you if any, or all, of the three voices are still playing.

SETSCORE

```
SETSCORE(<#>,<freq array>,<gate=1array>,<gate=0array>)
SETSCORE(1,freq1#(),time'on1#(),time'off1#())
```

<#> is the voice number to be set up.

<freq array> is a global integer array that is filled with the proper sequence of notes (freqencies). The last value should be zero (to turn off the voice).

<gate=1array> is a global integer array that tells how long (in 1/60 seconds) each note should have the gate on.

<gate=0array> is a global integer array the tells how long (in 1/60 seconds) each note should have the gate off (before going to the next note).

The example above sets up voice one to play a series of notes defined in the array freq1#, with the duration of the notes defined in the arrays time'on1# and time'off1#.

STOPPLAY

```
STOPPLAY(<voice one>,<voice two>,<voice three>)
STOPPLAY(TRUE,FALSE,FALSE)
```

This command will stop the given voice(s) from playing if the value given is one (or True), but continue playing if zero (or False).

The example above withdraws voice one from the automatic system, but leaves voices two and three alone.

WAITSCORE

```
WAITSCORE(<voice one>,<voice two>,<voice three>)
WAITSCORE(TRUE,FALSE,FALSE)
```

If any of the voices you have put TRUE for are still playing, then this function will return TRUE, otherwise it will return false.

Example: `WHILE WAITSCORE(TRUE,TRUE,TRUE) DO NULL`

This program line will wait and do nothing until all three voices have stoped playing.

JOYSTICK PACKAGE

This package allows you to read the direction and fire button status of a joystick plugged into one of the two control ports. All brands of joysticks that are for the Commodore 64 can be read with the following command after a USE JOYSTICK command has been issued.

JOYSTICK

```
JOYSTICK(<port>,<direction>,<button>)
JOYSTICK(1,joydir,f'button)
```

The C64 has two joystick ports (1 and 2) which can be checked with this command. The direction of the joystick and status of the fire button of the control port specified are put into the variables for direction and for button. Since 'direction' and 'button' are changed, they must be variables. The direction and fire button status returned is:

```
      1          Fire Button Pressed
    8   2           True
  7   0   3
    6   4        Fire Button Not Pressed
      5             False
```

PADDLES PACKAGE

This package has one command which lets you read the position and fire button status of a pair of paddles plugged into one of the two control ports. The sound chip can read the paddles and return a number 0-255 to represent how far around the to the right the paddle nobs are turned. You must issue the command: USE PADDLES to initialize the PADDLE command.

PADDLE

```
PADDLE(<port>,<paddle1>,<paddle2>,<button1>,<button2>)
PADDLE(1,pad1,pad2,fire1,fire2)
```

The C64 has two control ports (1 and 2), each of which can have a pair of paddles plugged in (for a total of four paddles) and can be checked with this command. The position of the paddles and status of their fire buttons of the control port specified are put into the variables for position and for button. When the paddle is turned to the extreme left a value of zero is returned. When the paddle is turned to the extreme right a value of 255 is returned.

The Koala Pad from Koala Technologies, or any other X-Y reading device, can be used with COMAL with this command. Just plug the device into one of the control ports and use this command to read the position of the device.

If you plan to draw graphics with paddles or any X-Y drawing device, remember that the range of the paddles is 0-255, while the graphics screen (after USE GRAPHICS) set up for 0-319 (X-direction), 0-199 (Y-direction). This can be corrected by the use of the WINDOW command (i.e. WINDOW(0,255,0,255) -- now you can plot directly to the graphics screen).

LIGHTPEN PACKAGE

The LIGHTPEN package allows you to easily read and use a lightpen with your Commodore 64. The lightpen can only be read if it is plugged into control port #1. The quality of your lightpen will control how fast and accurate it is.

ACCURACY

```
ACCURACY(<X range>,<Y range>)
ACCURACY(2,1)
```

This command sets how accurate the pen will read. The more accurate the setting, the longer it takes to read the position. For fast drawing a high value of 5-7 should be used. For slow, accurate drawing, a low value of 1-2 should be used.

DELAY

```
DELAY(<time>)
DELAY(5)
```

This command sets how many time the light pen has to return the same number to be accurate. A high value of 5-10 should be used for menu selection, while a low value 2-3 should be used for drawing.

OFFSET

```
OFFSET(<X correction>,<Y correction>)
OFFSET(5,0)
```

This command corrects for errors that develop on different sizes of televisions. The idea is to touch the lightpen to a known point on the screen, read the position returned, calculate the difference between the point returned and what the point should have been. Then set the OFFSET to that difference.

PENON

PENON

This function returns TRUE if the lightpen senses light, and FALSE if the lightpen does not.

READPEN

```
READPEN(<X coordinate>,<Y coordinate>,<penon?>)
READPEN(xpos,ypos,pen'status)
```

This command reads the position of the lightpen, corrects for the OFFSET command, and sets variables <X coordinate>,<Y coordinate> to the lightpen position. If the lightpen senses light, the variable <penon?> will be set to TRUE, otherwise it will be set to FALSE.

Variables have to be used because they are set to the lightpen settings. If you use numbers instead of variables, an error will result.

TIMEON

```
TIMEON(<time>)
TIMEON(3)
```

This command sets how long the lightpen must be taken away from the screen before it is not recognized.

FONT PACKAGE

The Commodore 64 has two character sets built into it, upper/graphics and lower/upper case. COMAL 2.0 allows you to define your own character sets and to use them in your own programs. This is a very powerfull feature. One way to use this feature is for custom game pieces on the text screen. Another use is to have a Danish character set. While you must spell the command words the same way they are spelled in English, variable names can be spelled any way you wish.

Characters are made up of dots, or pixels, eight wide and eight tall. Due to limitation of the video chip and your video screen, vertical lines should be two dots wide to make them more visable.

Besides the two character sets built into the computer, you can have two of your own defined character sets in memory at one time. These fonts are numbered for the FONT commands so you can access any of them:

```
FONT 0: User defined,    stored in RAM, read/write
FONT 1: User defined,    stored in RAM, read/write
FONT 2: UPPER/graphics, stored in ROM, read only
FONT 3: lower/UPPER,     stored in ROM, read only
```

SHIFT/Commmodore key toggles between FONTS 0 and 1, just as it normally does with fonts 2 and 3. To prevent this, printing chr$(8) will cause the system to ignore the SHIFT/Commodore key. To restore the keys, print chr$(9).

PLOTTEXT uses whatever FONT is currently active (normally FONT 2 or 3). This way you can plot user defined characters on the graphic screen.

Whenever user fonts are used, 5K (5120) bytes of memory is used. 4K (4096) for FONT 0 and 1, and 1K (1024) for the text screen. The built in FONTs, 2 and 3, and textscreen still exist, unchanged.

LINKFONT

LINKFONT

The first time LINKFONT is used it overwrites memory (it will copy the built in FONTs 2 and 3 into FONTs 0 and 1) and links the user defined FONTs to the program so that when you save the program, the user defined FONTs, 0 and 1, will be saved with the program. When LINKFONT is used for the first time it will also stop a running program (because it overwrites memory) so this command should be used in the direct mode the first time.

KEEPFONT

KEEPFONT

This command makes the user defined FONTs 0 and 1 the default FONTs (like FREEZING the FONTs into the system). After this command is issued, programs will be saved without the user defined FONTs. The only way to get back out of them is via a LOADFONT command or by turning the computer off and back on again. If no user font is active then KEEPFONT has no meaning

LOADFONT

```
LOADFONT(<filename>)
LOADFONT("font.russian")
```

This command will load a user defined character FONT into memory from the disk drive or tape machine. In essence it will do the following:

(1) Does a LINKFONT
(2) Loads a 4K (4096) byte SEQ file into the user defined FONTs
(3) Switches to FONTS 0 and 1 as current FONTs

This command will stop a running program so it must be used in direct mode or by use of a batch file with

SELECT INPUT.

A second or third LOADFONT will overwrite FONTs 0 and 1.

SAVEFONT

```
SAVEFONT(<filename>)
SAVEFONT("font.standard")
```

This command will write a 4K (4096) byte SEQ file to disk or tape from the two user defined FONTs. This command may be used in a running program without problem.

GETCHARACTER

```
GETCHARACTER(<font#>,<character#>,<string variable>)
GETCHARACTER(3,1,char$)
```

Each character in a FONT is an 8 byte long string, identifying the pixels as displayed on the screen. The first character of the string is the top row of pixels and the eighth character is the bottom row of pixels. If the pixel is on then it equals a '1' and if it is off then it will equal '0'. Each character is a binary number (0-255) representation of a row of the character. With this command you can look at any of the 256 characters (0-255) in any FONT (0-3).

PUTCHARACTER

```
PUTCHARACTER(<font#>,<character#>,<string variable>)
PUTCHARACTER(0,1,char$)
```

This command allows you to define a character with an eight byte string. Fortunately, COMAL 2.0 allows the use of binary numbers, so creating characters is easy. The following program demonstrates the use of this command:

```
NEW
USE FONT
LINKFONT
AUTO 1000
```

```
1000 USE FONT
1010 DIM char$ OF 8
1020 FOR x:=1 TO 8 DO
1030  READ a
1040  char$(x):=CHR$(a)
1050 ENDFOR x
1060 PUTCHARACTER(0,0,char$)
1070 // repelaces '@' in UPPER/graphics
1080 // with Commodore logo.
1090 data %00111100 // ..####..
1100 data %01100110 // .##..##.
1110 data %01100100 // .##..#..
1120 data %01100000 // .##.....
1130 data %01100100 // .##..#..
1140 data %01100110 // .##..##.
1150 data %00111100 // ..####..
1160 data %00000000 // ........
<Press STOP Key to end AUTO mode>

RUN
```

NATIONAL LANGUAGE MESSAGES

COMAL is designed to be used internationally. With that in mind, the capability to have the error messages displayed in many different languages was incorporated into COMAL.

When you start up the cartridge, it comes up in English. However, Danish messages are built into the cartridge as well and disk loaded packages for other languages will be available soon. To see messages in Danish just issue the command:

```
  USE DANSK
```

Now try this:

```
s       <hit RETURN key>
```

The computer responds:

```
s: ukendt saetning eller procedure
```

Don't worry. If you don't know Danish, just issue the command: USE ENGLISH and try it again. This time the computer responds:

```
s: unknown statement or procedure
```

CONTROL KEY USES

Use the following keys along with the CONTROL key to do special things:

```
A - remove indentations of a listed line
  - oops! return line to original state and
    remove changes - only before RETURN
  - show line - to see any line just type the line#
    then CONTROL A
B - move cursor one word back
C
D - Dump graphics screen (does PRINTSCREEN("lp:",79) )
    (causes an error if printer is not present)
E
F - move cursor one word forward
G
H
I
J
K - erase to end of logical line
L - goto end of line (last non blank character)
M
N
O
P - Textscreen dump (does a HARDCOPY("lp:") )
    (causes an error if printer is not present)
Q
R
S
T
U - Restore Function keys toggle
V - Sets textcolors to (6,6,1) - startup colors
W - Sets textcolors to (11,15,0)
    for black/white monitors
X - Changes text border color
    CNTRL X then CNTRL <color key>
Y - Changes text background
    CNTRL Y then CNTRL <color key>
Z - Makes the current textscreen colors the default
```

FOR MORE INFORMATION

The COMAL 2.0 Cartridge is a very powerful tool. We have just barely scratched the surface of its potential and capabilities. The COMAL HANDBOOK documents the COMAL part of the cartridge and complements this book very well.

COMAL TODAY newsletter is the BEST source of ongoing information and programs for COMAL. Starting in January 1985 it will have a complete section devoted to using the COMAL 2.0 Cartridge. If you aren't already you will want to subscribe to COMAL TODAY - $14.95 per year. (NOTE: Subscribe before January 1, 1985 and get 6 issues per subscription year - after that a subscription will be for 5 issues).

One of the best ways to uncover the power in your cartridge is to look at programs already written. Two disks come with the Deluxe Cartridge Package and provide a good start. But there are two more available December 1984: Cartridge Demo Disk #3 and #4. Demo Disk #3 will be the super demo disk with some useful and amazing programs on it. A special order form should have been packed with your cartridge providing you a special discount on Demo Disks #3 and #4. Take advantage of it!

If you need a tutorial about COMAL programming try these books:

FOUNDATIONS IN COMPUTER STUDIES WITH COMAL
by John Kelly
book=$19.95 matching disk=$19.95
(book and disk only $24.95 to COMAL TODAY subscribers)

STRUCTURED PROGRAMMING WITH COMAL
by Roy Atherton
book=$26.95 matching disk=$19.95
(book and disk only $32.95 to COMAL TODAY subscribers)

All COMAL materials are available directly from:
COMAL USERS GROUP, U.S.A., LIMITED
5501 Groveland Terrace
Madison, WI 53716-3251
(608) 222-4432

TRACE and BATCH FILES

TRACE is very useful as an aid to locating a problem in the program you are writing. If your running program begins doing something unexpected just hit the STOP key. Once the program is stopped enter the command: TRACE. The system will then tell you how it got to the point the program was stopped at. Another use for TRACE is when your program abends (abnormally ends) with an error message. Just enter the command: TRACE and the system will tell you how it got to the point of error.

BATCH FILES are extremely useful to the serious programmer, but even a casual programmer can take advantage of them. The BATCH FILE is a type of COMMAND FILE, a sequential ASCII file containing COMAL commands. The command SELECT INPUT "0:NAME" transfer control to the file named. All commands read from that file are treated as if they had been typed in from the keyboard. When the end of file is reached, control automatically returns to the keyboard. You can have as many batch files as you wish.

Programs that use batch files include: VIEW'FONTS and BATCHFILE'EDITOR (Demo Disk #3) and BATCH'COPIER (Demo Disk #2). Batch files can set up your system. For example, you could have a batch file set up the screen colors and function keys as you like them, and then give you a directory of just the PRG files on the disk. Let's say you called the file "BAT.MINE". Anytime you issued the command SELECT INPUT "BAT.MINE" it would do all those things. For an easy example of BATCH FILES in action type in the following lines from direct mode:

```
OPEN FILE 2,"0:BAT.MINE",WRITE
PRINT FILE 2: "CAT"
PRINT FILE 2: "SIZE"
CLOSE
SELECT INPUT "BAT.MINE"
```

Send us listings of what you put in your batch files, or send us a disk with the files on it. Then watch COMAL TODAY for tips on BATCH FILES shared by other users.

ISBN 0-928411-03-6